

Algoritmos
Lógica para Desenvolvimento
de Programação de Computadores



EDITORA AFILIADA

Seja Nosso Parceiro no Combate à Cópia Ilegal

A cópia ilegal é crime. Ao efetuá-la, o infrator estará cometendo um grave erro, que é inibir a produção de obras literárias, prejudicando profissionais que serão atingidos pelo crime praticado.

Junte-se a nós nesta corrente contra a pirataria. Diga não à cópia ilegal.

Seu Cadastro É Muito Importante para Nós

Se você não comprou o livro pela Internet, ao preencher a ficha de cadastro em nosso site, você passará a receber informações sobre nossos lançamentos em sua área de preferência.

Conhecendo melhor nossos leitores e suas preferências, vamos produzir títulos que atendam suas necessidades.

Obrigado pela sua escolha.

Fale Conosco!

Eventuais problemas referentes ao conteúdo deste livro serão encaminhados ao(s) respectivo(s) autor(es) para esclarecimento, excetuando-se as dúvidas que dizem respeito a pacotes de softwares, as quais sugerimos que sejam encaminhadas aos distribuidores e revendedores desses produtos, que estão habilitados a prestar todos os esclarecimentos.

Os problemas só podem ser enviados por:

1. E-mail: producao@erica.com.br
2. Fax: (11) 2097.4060
3. Carta: Rua São Gil, 159 - Tatuapé - CEP 03401-030 - São Paulo - SP



José Augusto N. G. Manzano
Jayr Figueiredo de Oliveira

Algoritmos
Lógica para Desenvolvimento
de Programação de Computadores

24ª Edição Revisada

2ª Reimpressão

São Paulo
2011 - Editora Érica Ltda.

Copyright © 2000 (edição original) 2009 (edição revisada e atualizada) da Editora Érica Ltda.

Todos os direitos reservados. Proibida a reprodução total ou parcial, por qualquer meio ou processo, especialmente por sistemas gráficos, microfilmicos, fotográficos, reprográficos, fonográficos, videográficos, internet, e-books. Vedada a memorização e/ou recuperação total ou parcial em qualquer sistema de processamento de dados e a inclusão de qualquer parte da obra em qualquer programa juscibernético. Essas proibições aplicam-se também às características gráficas da obra e à sua editoração. A violação dos direitos autorais é punível como crime (art. 184 e parágrafos, do Código Penal, conforme Lei nº 10.695, de 07.01.2003) com pena de reclusão, de dois a quatro anos, e multa, conjuntamente com busca e apreensão e indenizações diversas (artigos 102, 103 parágrafo único, 104, 105, 106 e 107 itens 1, 2 e 3 da Lei nº 9.610, de 19.06.1998, Lei dos Direitos Autorais).

Os Autores e a Editora acreditam que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia, explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Os nomes de sites e empresas, porventura mencionados, foram utilizados apenas para ilustrar os exemplos, não tendo vínculo nenhum com o livro, não garantindo a sua existência nem divulgação. Eventuais erratas estarão disponíveis para download no site da Editora Érica.

Conteúdo adaptado ao Novo Acordo Ortográfico da Língua Portuguesa, em execução desde 1º de janeiro de 2009.

*Algumas imagens utilizadas neste livro foram obtidas a partir do CorelDRAW 12, X3 e X4 e da Coleção do MasterClips/MasterPhotos® da IMSI, 100 Rowland Way, 3rd floor Novato, CA 94945, USA."

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Manzano, José Augusto N. G.

Algoritmos: Lógica para Desenvolvimento de Programação de Computadores / José Augusto N. G. Manzano, Jayr Figueiredo de Oliveira. -- 24. ed. rev. -- São Paulo : Érica, 2010.

Bibliografia.

ISBN: 978-85-365-0221-2

1. Algoritmos 2. Dados - Estruturas (Ciência da Computação)
I. Oliveira, Jayr Figueiredo de. II. Título.

10-11961

CDD-005.1

Índices para catálogo sistemático

1. Algoritmos: Computadores: Programação: Processamento de dados 005.1

Coordenação Editorial:	Rosana Arruda da Silva
Capa:	Maurício S. de França
Editoração e Finalização:	Rosana Ap. A. dos Santos
	Flávio Eugenio de Lima
	Carla de Oliveira Morais
	Marlene T. Santin Alves

Editora Érica Ltda.

Rua São Gil, 159 - Tatuapé

CEP: 03401-030 - São Paulo - SP

Fone: (11) 2295-3066 - Fax: (11) 2097-4060

www.editoraerica.com.br

Dedicatória

À minha esposa Sandra, Sol do meu dia, Lua da minha noite, vento que me abraça em seu amor, brisa que refresca minha alma, chuva que acalenta o meu ser. Te amo, te amo, te amo.

À minha filha Audrey, encanto de pessoa, flor que enfeita meu caminho e o de minha esposa.

Aos meus alunos que, com suas dúvidas, perguntas e questionamentos, fazem com que eu me aprimore cada vez mais. Um grande abraço a todos.

Augusto Manzano

Aos meus amores, dedico este trabalho:

À minha mãe Dorothy com todo o meu carinho, admiração e respeito.

À Marcella pelo companheirismo, paciência e motivação.

Aos meus filhos Isabella e Raphael, meus tesouros.

Ao meu único e fiel amigo Don.

Lembrem-se: minha família é minha vida.

Jayr Figueiredo

"Amarás o Senhor teu Deus, com todo teu coração,
com toda tua alma e com toda tua mente."

Mateus - 22, 37

Agradecimentos

Ao amigo e editor Antonio Marco, que desde o nosso primeiro contato confiou no meu trabalho, principalmente neste que se concretiza;

Aos amigos e professores Wilson Yamatumi, Roberto Affonso, Paulo Giovan, Avelino Bazanel e André Zanin pela constante troca de ideias e discussões técnicas.

Augusto Manzano

Aos meus alunos cujas exigências e anseios cobram um aprofundamento maior do meu universo de conhecimentos.

Jayr Figueiredo

Sumário

Capítulo 1 - Abordagem Contextual	15
1.1 - Introdução à Computação.....	15
1.2 - Mercado Computacional.....	20
1.3 - Linguagens de Programação.....	23
1.4 - Paradigmas de Programação.....	24
Capítulo 2 - Algoritmos e a Lógica de Programação	25
2.1 - Algoritmos Computacionais.....	25
2.2 - Cozinha x Computador.....	26
2.3 - Lógica de Programação de Computadores.....	27
2.4 - Compiladores, Interpretadores e Tradutores.....	39
Capítulo 3 - Programação com Sequência	41
3.1 - Etapas de Ação de um Computador.....	41
3.2 - Tipos de Dados Primitivos ou Dados Básicos.....	42
3.3 - O Uso de Variáveis.....	43
3.4 - O Uso de Constantes.....	44
3.5 - Os Operadores Aritméticos.....	45
3.6 - Expressões Aritméticas.....	46
3.7 - Instruções e Comandos.....	47
3.8 - Exercício de Aprendizagem.....	48
3.9 - Exercícios de Fixação.....	54
Capítulo 4 - Programação com Decisão	57
4.1 - Ser Programador.....	57
4.2 - Decisões, Condições e Operadores Relacionais.....	58
4.3 - Desvio Condicional Simples.....	59
4.4 - Desvio Condicional Composto.....	61
4.5 - Outras Formas de Desvios Condicionais.....	63
4.6 - Operadores Lógicos.....	74
4.7 - Divisibilidade: Múltiplos e Divisores.....	86
4.8 - Exercício de Aprendizagem.....	90
4.9 - Exercícios de Fixação.....	95
Capítulo 5 - Programação com Laços	99
5.1 - Ser Programador.....	99
5.2 - Laços ou Malhas de Repetição (Loopings ou Loops).....	100

5.3 - Laço de Repetição Condicional Pré-Teste	101
5.4 - Laço de Repetição Condicional Pós-Teste	108
5.5 - Laço de Repetição Condicional Seletivo	116
5.6 - Laço de Repetição Incondicional	118
5.7 - Considerações entre Tipos de Laços	120
5.8 - Exercício de Aprendizagem	121
5.9 - Exercícios de Fixação	129
Capítulo 6 - Estruturas de Dados Homogêneas de Uma Dimensão	131
6.1 - Ser Programador	131
6.2 - Matrizes de Uma Dimensão	132
6.3 - Exercício de Aprendizagem	137
6.4 - Exercícios de Fixação	140
Capítulo 7 - Aplicações Básicas com Matrizes de Uma Dimensão	143
7.1 - Ser Programador	143
7.2 - Classificação de Elementos	144
7.3 - Métodos de Pesquisa de Elementos	151
7.4 - Utilização de Matrizes Dinâmicas	161
7.5 - Exercício de Aprendizagem	162
7.6 - Exercícios de Fixação	170
Capítulo 8 - Estruturas de Dados Homogêneas de Duas Dimensões	172
8.1 - Ser Programador	172
8.2 - Matrizes com Mais de Uma Dimensão	173
8.3 - Matrizes de Duas Dimensões	174
8.4 - Exercício de Aprendizagem	177
8.5 - Exercícios de Fixação	184
Capítulo 9 - Estruturas de Dados Heterogêneas	187
9.1 - Ser Programador	187
9.2 - Tipo de Dado Derivado: Estrutura de Registro	188
9.3 - Estrutura de Registro de Matriz	192
9.4 - Estrutura de Matriz de Registros	196
9.5 - Exercício de Aprendizagem	198
9.6 - Exercícios de Fixação	205
Capítulo 10 - Subprogramas	206
10.1 - Ser Programador	206
10.2 - Modularidade	207

10.3 - Métodos Top-Down e Bottom-Up.....	207
10.4 - Procedimentos	209
10.5 - Escopo de Variáveis	216
10.6 - Passagens de Parâmetros.....	219
10.7 - Funções e Recursividade.....	223
10.8 - Exercício de Aprendizagem	228
10.9 - Exercícios de Fixação	256
Capítulo 11 - Programação Orientada a Objetos	260
11.1 - Origem	260
11.2 - PE <i>versus</i> POO	262
11.3 - Fundamentação	263
11.4 - Polimorfismo ou Poliformismo.....	266
11.5 - Resumo dos Termos Empregados na POO	267
Capítulo 12 - Aplicação Básica de POO	269
12.1 - Fundamentação	269
12.2 - Classe e Objeto.....	270
12.3 - Atributo e Método Externo	272
12.4 - Atributo e Método Interno.....	275
12.5 - Herança.....	278
12.6 - Encapsulamento	284
12.7 - Poliformismo	291
Apêndice A - Resolução de Alguns Exercícios de Fixação	302
Apêndice B - Exemplos de Codificação	306
Apêndice C - Comparativo de Instruções entre Linguagens de Programação	316
Bibliografia.....	318

Prefácio

Este livro surgiu da necessidade de conciliar o ensino de técnicas de programação com os programas curriculares das instituições de ensino em que trabalhamos e também de outras escolas, principalmente dos cursos técnicos e tecnológicos. No decorrer dos vários anos, como professores sentimos falta de um material que atendesse às nossas necessidades básicas e também às dos educandos e alunos.

Ele traz o máximo de detalhes para o programador iniciante no estudo da lógica de programação de computadores, desde o paradigma da programação estruturada até o paradigma da programação orientada a objetos, de tal forma que não necessite buscar, em um primeiro momento, informações complementares em outras obras. Com relação ao estudo de algoritmos, o enfoque central é feito em três etapas: primeiramente a fase de entendimento, descrita muitas vezes em passos numerados, chegando a parecer uma receita culinária, uma segunda forma de algoritmo baseada na representação gráfica do raciocínio lógico por meio de diagrama de blocos a partir do uso da norma ISO 5807:1985 e, por último, o algoritmo textual codificado em português estruturado, considerado uma pseudolinguagem de programação.

Esta obra está dividida em cinco partes e abrange um programa de estudo de técnicas de programação. A primeira parte traz informações introdutórias e conceituais a respeito de lógica de programação, mercado de trabalho, organização de computadores, entre outros assuntos. A segunda enfatiza a prática do estudo de lógica, começando dos pontos básicos, passa pelo estudo de técnicas de programação com decisões e laços de repetição. A terceira parte apresenta algumas técnicas auxiliares para facilitar a organização de dados em memória. A quarta destaca a programação de forma estruturada, com o uso de sub-rotinas e orientação a objetos. A quinta parte (apêndices) fornece a resolução de alguns exercícios de fixação e também alguns exemplos de codificação de programas escritos nas linguagens formais de programação PASCAL, BASIC (em modo estruturado), C e C++, além da codificação em português estruturado.

Esperamos, sinceramente, que este trabalho seja bastante útil não só ao aluno, mas também ao amigo professor, pois foi elaborado com base em experiências adquiridas em sala de aula, que resultou um livro bastante didático. Um abraço a todos!

Os autores

Sobre os Autores

José Augusto N. G. Manzano

Brasileiro, nascido no Estado de São Paulo, capital, em 26/04/1965, é professor e mestre *intra corporis* pelo Centro Universitário Sant'Anna. Atua na área de Tecnologia da Informação (Desenvolvimento de Software e Treinamento) desde 1986, tendo participado do desenvolvimento de aplicações computacionais para as áreas de telecomunicações e comércio. Na carreira docente iniciou sua atividade em cursos livres, passando por empresas de treinamento e em seguida atuando nos ensinos técnico e universitário. Atuou em empresas na área de computação como SERVIMEC S.A. (funcionário), CEBEL, SPCI, BEPE, ORIGIN, OpenClass, entre outras, como consultor e instrutor free-lancer.

Atualmente é professor do IFSP (Instituto Federal de São Paulo, antiga Escola Técnica Federal). Em sua carreira docente, devido à experiência acumulada, possui condições técnicas de ministrar componentes curriculares de Algoritmos, Lógica de Programação, Estrutura de Dados, Técnicas de Programação, Microinformática, Informática Básica, Linguagens de Programação Estruturada, Linguagens de Programação Orientada a Objetos, Engenharia de Software, Tópicos Avançados em Processamento de Dados, Sistemas de Informação, Engenharia da Informação, Arquitetura de Computadores e Tecnologia Web. Possui conhecimento de uso e aplicação das linguagens de programação para computadores Assembly, BASIC (padrão 60), LOGO, PASCAL, FORTRAN, C, C++, JAVA, MODULA-2, STRUCTURED BASIC, HTML, XHTML, JavaScript, VBA, C#, Lua e ADA. Possui vários livros publicados pela Editora Érica, além de artigos técnicos publicados no Brasil e no exterior.

Jayr Figueiredo de Oliveira

Pós-doutorado em Administração na área de Sistemas e Tecnologias da Informação (Universidade de São Paulo - FEA/USP), doutorado em Educação na área de Novas Tecnologias, mestrado em Administração e Planejamento (Pontifícia Universidade Católica - PUC/SP); especializações em Administração de Sistemas (FECAP), Didática do Ensino Superior e Ciência da Computação (Universidade Presbiteriana Mackenzie), MBA em Inovação, Tecnologia e Conhecimento (FEA/ USP), bacharel em Administração de Empresas.

Atua desde 1977 como profissional em Administração de Sistemas de Informação e desde 1985 como docente em Educação Superior, tendo ocupado, em ambas as atividades, inúmeros cargos de chefia. Publicou mais de 15 livros e inúmeros artigos nas áreas de Sistemas e Tecnologias da Informação, Liderança, Gestão Organizacional e Negócios.

Carta ao Professor

Prezado amigo educador e professor,

Não é à toa que uma obra literária chega até a edição que esta contempla, principalmente por se tratar de um livro técnico para a área de desenvolvimento de software. Neste contexto, esta obra pode ser considerada um *best-seller*, e por isso temos muito a agradecer aos leitores, principalmente ao amigo educador que, nesses anos de jornada, legitimou nosso trabalho, indicando a seus educandos e alunos, confiando-nos a tarefa de vetores na transmissão de conhecimento.

A primeira edição deste trabalho foi publicada no primeiro semestre de 1996 para atender, principalmente, a uma necessidade particular. No mesmo ano tivemos a surpresa de sair a segunda edição. Este foi o sinal de que a obra teria futuro, e desde então são publicadas duas a três edições por ano, até que, no segundo semestre de 2000, surgiu a décima edição, quatro anos depois do lançamento da primeira edição. Só para ter uma ideia, no primeiro semestre de 2000 foram publicadas duas edições, sendo a oitava e a nona. A vigésima edição desta obra foi publicada no ano de 2006, ou seja, seis anos após a décima edição. Apesar da aparente queda na quantidade de edições entre a décima e a vigésima, continuamos otimistas, pois, na verdade, as condições de mercado levaram a essa queda, e não o nosso trabalho.

Até a nona edição a obra amadureceu, foram feitas algumas mudanças, melhorias e pequenas correções implementadas. Da sétima para a oitava edição foram acrescentados novos exercícios. Da décima até a décima quinta edição ocorreram alguns novos ajustes. A partir da décima sexta edição o livro conta com mais de cento e quarenta exercícios de lógica de programação de computador, enquanto a primeira edição traz em torno de setenta exercícios. Após a décima sexta e até a vigésima primeira edição foram feitos pequenos ajustes e algumas correções necessárias, porém mantendo a obra praticamente com a mesma estrutura.

A partir da vigésima segunda edição (2009) este livro sofre reforma e atualização na estrutura do texto. Foram acrescentados novos detalhes, alguns pontos foram ajustados e realinhados. Há dois capítulos de introdução à programação orientada a objetos apresentados de forma inédita; um aborda os conceitos em si e o outro mostra alguns exemplos de aplicação da programação orientada a objetos por meio do uso de diagramas de UML (diagrama de classe e objeto) e de diagramas de blocos, segundo a norma internacional ISO 5807:1985.

Acreditamos que elevamos a qualidade da obra, tornando-a mais confortável para o trabalho tanto dos colegas educadores quanto de nossos alunos e educandos.

À medida que as edições deste livro avançam, vemos a necessidade de inserir novos exercícios de fixação e outros detalhes que em certo momento não tínhamos notado serem importantes. Sabemos que em breve essa atitude não mais ocorrerá, pois os espaços existentes para essa estratégia estão sendo preenchidos. Nossa intenção é ampliar ao máximo a possibilidade de o educador selecionar a bateria de exercícios que esteja de acordo com o nível do curso ministrado.

Como de costume, esperamos que nosso trabalho (por mais simples que seja, por mais básico que possa parecer) facilite a tarefa do amigo educador na preparação e condução de suas aulas.

Cordialmente, um grande abraço!

Carta ao Aluno

Prezado aluno,

Normalmente, alguns profissionais (a partir da década de 1980) e mesmo alguns alunos dizem que é perda de tempo desenvolver o algoritmo (diagrama de blocos e pseudocódigo) de um programa. Que é bem mais prático e rápido desenvolvê-lo diretamente em um computador, porém geralmente não conseguem entregar seus programas ou sistemas nos prazos estipulados. Quando julgam estar com seus programas prontos, acabam por descobrir uma série de erros de lógica e de sintaxe. Esses erros fazem com que o programador fique sentado horas ou mesmo dias em frente ao computador, tentando localizar o que está errado, e assim acaba ultrapassando os prazos.

O excesso de erros de sintaxe denota um "profissional" com pouca experiência na utilização de uma determinada linguagem de programação, enquanto erros de lógica mostram grande despreparo na "arte" de programar um computador. Acreditando que os erros podem ser retirados no momento da compilação de um programa, as pessoas perdem muito mais tempo do que imaginam, e assim entram em um círculo vicioso de querer resolver o problema (erros de sintaxe e de lógica) por tentativa e erro, o que é abominável do ponto de vista profissional. Imagine se outros profissionais, como engenheiros, comandantes de aeronaves, médicos, advogados, entre outros, pensassem desta forma. Certamente a humanidade estaria com sérios problemas, se é que ainda estaria habitando este planeta.

Até o final da década de 1970, os programadores e demais profissionais da então denominada área de informática, hoje área de tecnologia da informação, seguiam rigorosos procedimentos para o desenvolvimento de programas e sistemas, o que resultava em produtos de excelente qualidade, dentro dos prazos estabelecidos. Com o surgimento dos microcomputadores e sua popular utilização a partir de meados da década de 1970, alguns novos "profissionais" da área passaram a deixar de lado os procedimentos adequados e adquiriram vícios e defeitos graves.

Muitos desses "profissionais", sem o devido preparo, passaram a achar que os conceitos até então utilizados eram ultrapassados e abrir mão deles traria maior velocidade na entrega de trabalhos. Infelizmente essa mentalidade espalhou-se por praticamente todo o mundo, apesar de alguns profissionais verdadeiramente sérios tentarem orientar de forma contrária.

Imagine uma prova de maratona com vários corredores. Assim que é dada a largada, alguns saem correndo desesperados para ficar à frente dos demais corredores. Algum tempo depois, sem fôlego, acabam perdendo o ritmo e também sua posição, normalmente chegando nos últimos lugares. Enquanto alguns saem correndo de forma desesperada, outros procuram poupar-se e quando estão quase no final, usam o fôlego para disputar os primeiros lugares.

Assim como um corredor experiente, um programador de computadores precisa saber poupar-se. É melhor ir devagar no início do desenvolvimento do que achar que, ao pular algumas etapas, conseguirá ganhar a corrida. Fazer os algoritmos preliminares de um programa pode parecer demorado, porém essa atitude proporciona uma visão adequada de todo o problema, garantindo ausência de erros de lógica no programa final.

Esperamos, sinceramente, que este trabalho seja muito útil ao seu aprendizado e proporcione as condições básicas necessárias para que consiga alçar voos mais altos. Pedimos que preste atenção ao que lhe é ensinado pelo professor em sala de aula e faça todos os exercícios de fixação solicitados, consultando o professor sempre que surgirem dúvidas.

Cordialmente, um grande abraço!

Os autores



Este capítulo apresenta os detalhes gerais que norteiam a área da computação. Traz uma introdução aos elementos conceituais da área e de seu desenvolvimento. Mostra a organização de um computador eletrônico e indica as unidades de medidas utilizadas. Aborda detalhes da tabela ASCII, apresenta informações do mercado de computação, das linguagens de programação e sua classificação.

1.1 - Introdução à Computação

A necessidade de desenvolver ferramentas que facilitassem o uso e o manuseio de operações de cálculos fez com que o ser humano chegasse ao estágio atual de desenvolvimento da área da computação. Esse processo teve início há muito tempo com o surgimento do primeiro computador denominado *ábaco*, Figura 1.1, por volta de 3.500 a.C. na região da Mesopotâmia, passando por China e Japão (ALCALDE, GARCIA & PENUELAS, 1991, P. 8) e seu uso estende-se até o século XXI.

Não é objetivo desta obra estender-se em méritos históricos ou tecer explicações sobre a arquitetura dos computadores, pois existem trabalhos muito bons sobre este assunto já publicados. No entanto, faz-se necessário passar ao iniciante no aprendizado da programação de computadores uma rápida visão da estrutura de um computador e sua funcionalidade. Assim sendo, o primeiro item a ser entendido é o termo *computador*.

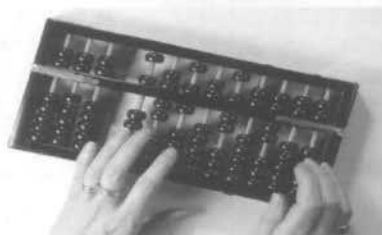


Figura 1.1 - Ábaco.¹

A palavra *computador* origina-se do latim *computatore*, um substantivo masculino que significa "aquele que efetua cálculos". No contexto desta obra, o termo *computador* está associado a um equipamento eletrônico capaz de executar algumas etapas de trabalho, como receber, armazenar, processar lógica e aritmeticamente dados com o objetivo principal de resolver problemas.

¹ Imagem obtida no sítio <http://ummundomagico2.blogs.sapo.pt/arquivo/860885.html> em outubro de 2010.

O computador eletrônico como se conhece atualmente origina-se das ideias estabelecidas pelo cientista, matemático e filósofo inglês Charles Babbage, Figura 1.2, que no ano de 1834 apresentou as ideias da *máquina analítica*, considerada precursora dos computadores eletrônicos mais modernos, apesar de, nessa ocasião, a área de eletrônica ainda não ter sido desenvolvida. Charles Babbage nasceu em 26 de dezembro de 1791 e faleceu em 18 de outubro de 1871.



Figura 1.2 - Charles Babbage.²

1.1.1 - Organização de um Computador

O computador eletrônico é uma coleção de componentes interligados com o objetivo de efetuar (processar) operações aritméticas e lógicas de grandes quantidades de dados. A Figura 1.3 mostra de forma esquemática os componentes de um computador eletrônico.

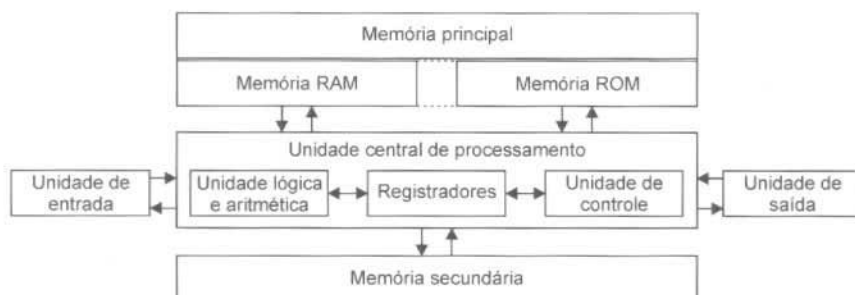


Figura 1.3 - Estrutura dos componentes de um computador eletrônico.

O componente denominado *unidade de entrada* é responsável pela entrada de dados no computador. Os dados inseridos em um computador podem ser armazenados no componente *memória secundária* ou processados no componente *memória principal*, mais precisamente na *memória RAM* (*Random Access Memory* - Memória de Acesso Randômico). Esse tipo de componente pode ser representado pelos periféricos *teclado*, *scanner*, *mouse*, *câmeras de vídeo*, *arquivos*, *sensores de movimento*, entre outros componentes.

O componente *unidade de saída* é responsável pela apresentação de dados e/ou informações que tenham sido processados na memória principal ou que estejam armazenados na memória secundária do computador. Esse tipo de componente pode ser representado pelos periféricos *monitores de vídeo*, *impressoras*, *arquivos*, entre outros.

O componente denominado *unidade central de processamento* é responsável pelo controle das operações de entrada e de saída de um computador. Além disso, esse componente é também responsável por todo o controle operacional, sendo o "cérebro" e o "sistema nervoso" de um computador. A unidade central de processamento é, grosso modo, subdividida em três componentes auxiliares, a saber:

² Imagem obtida no sítio <http://pt.wikipedia.org/wiki/Imagem:CharlesBabbage.jpg> em outubro de 2010.

- ▶ A *unidade lógica e aritmética* é responsável por processar operações matemáticas (unidade aritmética) e/ou operações lógicas (unidade lógica). Esse componente é sem dúvida o mais importante da CPU.
- ▶ Os *registradores* são componentes de memória que apresentam alta performance de velocidade na execução de instruções de processamento aritmético ou lógico.
- ▶ A *unidade de controle* executa instruções de controle do computador. Dependendo da instrução executada, esse componente faz o desvio do controle para a unidade lógica ou unidade aritmética ou, ainda, envia dados para componentes externos à CPU.

A *memória principal* é formada principalmente pelos componentes de memórias RAM e ROM (*Read Only Memory* - Memória Somente de Leitura).

- ▶ A *memória RAM* é utilizada pela CPU para manter temporariamente dados e instruções que são usados no processamento. O controle operacional da execução de instruções e armazenamento de dados nessa memória é realizado por um conjunto de circuitos lógicos. Por esta razão, esse tipo de operação é muito rápido. Apesar de possuir vantagem na velocidade de execução, essa memória é volátil, ou seja, os dados nela armazenados são perdidos quando o computador é desligado ou tem seu sistema reinicializado.
- ▶ A *memória ROM* é utilizada pela CPU para inicializar o computador quando é ligado e faz a busca do sistema operacional instalado em *memória secundária*, a qual gerencia as funções de trabalho e permite usar o computador de forma mais fácil. A memória ROM não pode ser alterada ou regravada como ocorre com os dados e instruções da memória RAM, pois nela estão gravadas as características definidas pelo fabricante do computador em uso.

A *memória secundária*, também conhecida como memória de massa, tem por finalidade armazenar dados a longo prazo, pois os dados armazenados nesse tipo de memória são preservados mesmo quando o computador estiver desligado. Essa memória possui, normalmente, acesso lento. São exemplos os periféricos de armazenamento: *disquetes*, *discos rígidos (winchesters)*, *pen drives*, *cartões de memória*, entre outros.

1.1.2 - Unidades de Medidas Computacionais

A menor informação de um computador eletrônico chama-se *bit* (*binary digit* - dígito binário), o qual é representado pelos valores numéricos "1" (um) e "0" (zero). Os valores binários são eletronicamente usados pelo computador para representar estados eletromagnéticos dos circuitos que compõem a sua estrutura funcional e assim possibilitam representar dados do mundo exterior, além de estabelecer as bases de funcionamento da máquina. O estado eletrônico "1" representa a ativação (ligamento) de um determinado recurso ou circuito interno, enquanto o estado eletrônico "0" representa a desativação (desligamento) de um determinado recurso ou circuito interno. Note que um computador eletrônico opera com dados binários, ou seja, opera na base dois.

O computador tem a capacidade de utilizar dados e informações do mundo exterior e representá-los de forma binária em seus circuitos e memórias. Assim sendo, os dados básicos existentes no mundo exterior, como dados numéricos e alfabéticos, incluindo os símbolos de pontuação, possuem um valor binário particular para representá-los no computador.

A representação de um dado externo na memória de um computador usa um octeto de *bits*, denominado, em inglês, *byte*. Desta forma, um *byte* (que representa um caractere qualquer) é um conjunto de oito *bits*.

Tomando por base o valor numérico dois referente à base de operação interna de um computador eletrônico (o *bit*) e elevando esse valor ao expoente oito referente à quantidade de *bits* de um *byte* (2^8), obtém-se o valor 256 que é a quantidade máxima de caracteres que podem ser usados em um computador eletrônico.

A estratégia de uso de dígitos binários pelos computadores permite maior armazenamento de dados em memória do que se essa mesma tarefa utilizasse dígitos decimais. Se fossem usados dígitos decimais, haveria a necessidade de operar dez símbolos numéricos, o que seria um desperdício de memória e exigiria processadores mais complexos.

Por meio dos dígitos binários, consegue-se fazer uso de 256 caracteres diferentes que são mais do que suficientes para representar caracteres alfabéticos, numéricos e de pontuação que, na prática, utilizam apenas 128 das 256 possibilidades, sobrando ainda um conjunto de 128 *bytes* para a representação de outros caracteres particulares e de uso específico dos fabricantes de computadores (*hardware*) e dos desenvolvedores de programas (*software*).

A partir da definição da base binária é estabelecido um padrão de mensuração da quantidade de caracteres que um computador pode usar tanto na memória principal quanto na memória secundária. É também pela quantidade de *bits* que se determina o padrão de processamento de um computador.

Os primeiros microcomputadores Altair 8800 (1975), Apple II (1976) e TRS-80 Model I (1977) efetuavam seu processamento à taxa de oito *bits*. Esses equipamentos conseguiam processar apenas um *byte* por vez, ou seja, uma palavra por vez, uma *word*.

Em 1980, a empresa IBM, que operava somente no mercado de computadores de grande porte (*mainframes*), lança seu primeiro microcomputador de 16 *bits*, chamado IBM-PC. A sigla PC vem de *Personal Computer* (computador pessoal) e é usada para classificar qualquer computador de pequeno porte.

O IBM-PC tinha a capacidade de processar duas palavras por vez, ou seja, processava uma *double word* (palavra dupla). Depois, em 1984, a empresa Apple lança o microcomputador Macintosh equipado com um microprocessador que operava à taxa de 32 *bits*, processava quatro palavras por vez, ou seja, uma *quadword* (palavra quádrupla). O padrão 32 *bits* foi seguido pela empresa Intel, a partir de 1985, e depois por seus concorrentes. A partir de 2003 a empresa AMD lança seu microprocessador que trabalha à taxa de 64 *bits*, processando assim oito palavras por vez, ou seja, uma *double quadword* (palavra quádrupla dupla).

O volume de dados a ser processado numa memória principal ou armazenado numa memória secundária é medido em relação à quantidade de *bytes* manipulados. A tabela seguinte apresenta algumas unidades de medida utilizadas.

Unidade	Quantidade de caracteres
Bit (b)	Conjunto de dois <i>bits</i> que possibilita a ativação e a desativação de recursos e circuitos eletrônicos.
Byte (B)	Conjunto de oito <i>bits</i> que possibilita a definição e o uso de duzentos e cinquenta e seis (2^8) símbolos para representação de caracteres numéricos, alfabéticos, de pontuação e gráficos (opcional).
Kbyte (KB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Kbyte (<i>kilobyte</i>) equivale a 1.024 caracteres, sendo obtido a partir de 2^{10} .
Mbyte (MB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Mbyte (<i>megabyte</i>) equivale a 1.048.576 caracteres (1.024 Kbytes), sendo obtido a partir de 2^{20} .
Gbyte (GB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Gbyte (<i>gigabyte</i>) equivale a 1.073.741.824 caracteres (1.024 Mbytes), sendo obtido a partir de 2^{30} .

Unidade	Quantidade de caracteres
Tbyte (TB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Tbyte (<i>terabyte</i>) equivale a 1.099.511.627.776 caracteres (1.024 Gbytes), sendo obtido a partir de 2^{40} .
Pbyte (PB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Pbyte (<i>petabyte</i>) equivale a 1.125.899.906.842.624 caracteres (1.024 Tbytes), sendo obtido a partir de 2^{50} .
Ebyte (EB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Ebyte (<i>exabyte</i>) equivale a 1.152.921.504.606.846.976 caracteres (1.024 Pbytes), sendo obtido a partir de 2^{60} .
Zbyte (ZB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Zbyte (<i>zettabyte</i>) equivale a 1.180.591.620.717.411.303.424 caracteres (1.024 Ebytes), sendo obtido a partir de 2^{70} .
Ybyte (YB)	Definição da quantidade de caracteres a ser utilizada e armazenada em memórias: principal e secundária. 1 Ybyte (<i>yottabyte</i>) equivale a 1.208.925.819.614.629.174.706.176 caracteres (1.024 Zbytes), sendo obtido a partir de 2^{80} .

1.1.3 - Tabela ASCII

A representação dos 256 caracteres utilizados por computadores eletrônicos obedece à estrutura de uma tabela de caracteres oficial chamada ASCII (*American Standard Code for Information Interchange* - Código Americano Padrão para Intercâmbio de Informações), a qual deve ser pronunciada como *asqui* e não *asqui dois*, como algumas pessoas inadvertidamente falam.

A tabela ASCII, Figura 1.4, foi desenvolvida entre os anos de 1963 e 1968 com a participação e a colaboração de várias companhias de comunicação norte-americanas, com o objetivo de substituir o até então utilizado código de *Baudot*, o qual usava apenas cinco *bits* e possibilitava a obtenção de trinta e duas combinações diferentes. Muito útil para manter a comunicação entre dois teleimpressores (telegrafia), conhecidos no Brasil como rede TELEX operacionalizada pelos CORREIOS para envio de telegramas. O código de *Baudot* utilizava apenas os símbolos numéricos e letras maiúsculas, tornando-o impróprio para o intercâmbio de informações entre computadores.

O código ASCII padrão permite a utilização de 128 símbolos diferentes (representados pelos códigos decimais de 0 até 127). Nesse conjunto de símbolos estão previstos o uso de 96 caracteres imprimíveis, tais como números, letras minúsculas, letras maiúsculas, símbolos de pontuação e caracteres não imprimíveis, como retorno do carro de impressão (*carriage return* - tecla <Enter>), retrocesso (*backspace*), salto de linha (*line feed*), entre outros.

Da possibilidade de uso dos 256 caracteres, faz-se uso apenas dos 128 primeiros, o que deixa um espaço para outros 128 caracteres estendidos endereçados pelos códigos decimais de 128 até 255. A parte da tabela endereçada de 128 até 255 é reservada para que os fabricantes de computadores e de programas de computador possam definir seus próprios símbolos.

A Figura 1.4 mostra, respectivamente, a tabela ASCII padrão, códigos de 0 até 127, e a parte estendida, códigos de 128 até 255, utilizada para a representação de caracteres especiais pela empresa IBM quando da criação do seu padrão de microcomputadores durante a década de 1980, denominado padrão IBM-PC.

0		24	↑	48	0	72	H	96	`	120	x	144	É	168	¿	192	L	216	±	240	≡
1	⊖	25	↓	49	1	73	I	97	a	121	y	145	æ	169	⌈	193	⊥	217	⌋	241	≡
2	⊗	26	→	50	2	74	J	98	b	122	z	146	⦶	170	⌋	194	⌊	218	⌌	242	≡
3	♥	27	←	51	3	75	K	99	c	123	(147	ô	171	⌌	195	⌊	219	▀	243	≡
4	♣	28	↔	52	4	76	L	100	d	124		148	ö	172	⌌	196	⌊	220	▀	244	≡
5	♠	29	↔	53	5	77	M	101	e	125)	149	ò	173	⌌	197	⌊	221	▀	245	≡
6	♣	30	^	54	6	78	N	102	f	126	~	150	û	174	⌌	198	⌊	222	▀	246	≡
7		31	v	55	7	79	O	103	g	127	Δ	151	ù	175	⌌	199	⌊	223	▀	247	≡
8		32		56	8	80	P	104	h	128	Ç	152	Û	176	⌌	200	⌊	224	▀	248	≡
9		33	!	57	9	81	Q	105	i	129	ü	153	Ö	177	⌌	201	⌊	225	▀	249	≡
10		34	"	58	:	82	R	106	j	130	é	154	Ü	178	⌌	202	⌊	226	▀	250	≡
11	♂	35	#	59	:	83	S	107	k	131	â	155	ç	179	⌌	203	⌊	227	▀	251	≡
12	♀	36	\$	60	<	84	T	108	l	132	ä	156	£	180	⌌	204	⌊	228	▀	252	≡
13		37	%	61	=	85	U	109	m	133	à	157	¥	181	⌌	205	⌊	229	▀	253	≡
14	J	38	&	62	>	86	U	110	n	134	ã	158	₣	182	⌌	206	⌊	230	▀	254	≡
15	∞	39	'	63	?	87	W	111	o	135	ç	159	₣	183	⌌	207	⌊	231	▀	255	≡
16	▶	40	(64	@	88	X	112	p	136	ê	160	á	184	⌌	208	⌊	232	▀		≡
17	◀	41)	65	A	89	Y	113	q	137	ë	161	í	185	⌌	209	⌊	233	▀		≡
18	↓	42	*	66	B	90	Z	114	r	138	è	162	ó	186	⌌	210	⌊	234	▀		≡
19	!!	43	+	67	C	91	[115	s	139	ÿ	163	ú	187	⌌	211	⌊	235	▀		≡
20	¶	44	.	68	D	92	\	116	t	140	î	164	ñ	188	⌌	212	⌊	236	▀		≡
21	§	45	-	69	E	93]	117	u	141	ï	165	Ñ	189	⌌	213	⌊	237	▀		≡
22	≡	46	.	70	F	94	^	118	v	142	ÿ	166	ª	190	⌌	214	⌊	238	▀		≡
23	‡	47	/	71	G	95	_	119	w	143	ÿ	167	ª	191	⌌	215	⌊	239	▀		≡

Figura 1.4 - Código padrão ASCII.³

1.2 - Mercado Computacional

Um ponto que precisa ficar claro é a noção básica da dimensão do mercado computacional em que esta obra está inserida. Por serem os microcomputadores muito populares, a maior parte das pessoas enfatiza apenas esse tipo de computador no aprendizado, esquecendo-se de que o mercado de computação mundial é formado por três segmentos de computadores muito distintos: os computadores de grande porte (*mainframes*), médio porte (*minicomputadores*) e de pequeno porte (*microcomputadores*). Os microcomputadores estão divididos em duas categorias: os computadores de mesa (*desktops*) e os portáteis (*laptops*⁴ ou *notebooks* e os *handhelds*⁵).

O ensino de programação proposto é voltado para a programação de computadores de modo geral, independentemente da categoria de inserção no mercado, pois a lógica de programação utilizada em um computador é a mesma para qualquer tipo. O que de fato muda é o foco da aplicação computacional desenvolvida e as linguagens de programação mais utilizadas em cada um dos segmentos de mercado.

³ Imagem obtida no sítio <http://www.docjojo.com/Archive/homepage/crypto.htm> em janeiro de 2009.

⁴ *Laptop* é a junção das palavras em inglês *lap* (colo) e *top* (em cima). Refere-se a computadores portáteis que podem ser usados em cima das pernas. Os termos *laptop* e *notebook* são considerados sinônimos, duas referências ao mesmo tipo de equipamento.

⁵ *Handhelds* são computadores de tamanho pequeno, de mão, utilizados como assistentes digitais pessoais, conhecidos também como PDAs (*Personal Digital Assistants*). São encontrados em diversos formatos e cada vez mais comuns até em telefones celulares.

Os computadores de grande porte (*mainframes*) são equipamentos com alto poder de processamento por trabalharem e controlarem grandes volumes de dados na casa de milhões de instruções por segundo. Possuem grandes capacidades de memória, principalmente memória secundária, e custo de aquisição e manutenção bastante elevado, tendo seu público-alvo configurado como grandes empresas e órgãos (universidades, centros de pesquisa, bancos, aviação etc.). Esse computador é capaz de fornecer serviços de processamento a milhares de terminais e usuários simultaneamente conectados em rede.

Um *mainframe* ocupa, normalmente, uma sala com muitos metros quadrados. Segundo Norton (1996, p. 30), a origem do termo *mainframe* é desconhecida, mas cita que na empresa IBM existem documentos com referência ao termo *frame* (estrutura) para componentes integrados a um computador e que, a partir disso, pode-se supor que o computador que integrava esses componentes passou a ser chamado *mainframe* (estrutura principal).

Os minicomputadores são equipamentos com médio poder de processamento capazes de suportar serviços de processamento na casa de duas centenas de terminais e usuários simultaneamente conectados em rede. Normalmente são equipamentos usados como estações de trabalho. São máquinas que se inserem num nível intermediário de processamento, podendo estar muito próximas em capacidade de um computador de grande porte menos robusto, como estar muito próximas em capacidade de uma estação de trabalho a partir do mais alto poder de computação conseguido com o uso de microcomputadores.

Por ter um custo de aquisição e manutenção mais baixo que os computadores de grande porte e ocupar basicamente quase que o espaço de um microcomputador de mesa, esse equipamento vem sendo cada vez mais utilizado no lugar de computadores de grande porte menos robustos. São muito usados em empresas de grande e médio portes.

O termo minicomputador surgiu no início da década de 1960, quando a empresa DEC lançou o computador PDP e a imprensa começou a chamá-lo de minicomputador pelo fato de ser menor que um computador de grande porte (NORTON, 1996, p. 30).

Os microcomputadores (micros ou computadores pessoais), sejam de mesa (*desktop*), portáteis (*laptops* ou *notebooks*) ou de mão (*handhelds*), são equipamentos com baixo poder de processamento, quando comparados com os computadores de médio porte e de grande porte. São normalmente utilizados na computação pessoal, em escritórios, consultórios, escolas e também em empresas de modo geral. Atendem às necessidades e aplicações básicas como processamento de texto, planilhas eletrônicas, manipulação de bancos de dados de pequeno a médio porte, apresentações, editorações, além de serem muito utilizados como terminais de acesso à rede mundial Internet.

Por terem um custo de aquisição e manutenção extremamente baixo, são os equipamentos mais facilmente encontrados e estão à disposição do público em geral. As configurações mais robustas podem e são usadas como estações de trabalho. Esse tipo de computador pode ser operado tanto em redes como fora delas, diferentemente dos computadores de grande e médio portes que sempre estão em rede. Quando usado em redes, o conjunto de microcomputadores em operação pode fornecer, a um baixo custo, um poder de processamento próximo ao encontrado nos minicomputadores.

Os microcomputadores de mesa e os portáteis são genericamente chamados de PCs. O termo microcomputador surgiu em 1975 quando começaram a aparecer pequenos computadores que podiam ser utilizados por pessoas em suas casas, pois antes o uso era restrito a órgãos governamentais, empresas e universidades.

Parece não existir um estudo estatístico oficial que apresente o percentual de participação e utilização dos tipos de computadores no mercado mundial, mas fazendo uma pesquisa nos principais serviços de busca da Internet (*Google*, *MSN* e *Yahoo*), é possível especular o grau de interesse do público nesses segmentos. Nos valores

apresentados não se consideram prós ou contras de cada um dos segmentos, apenas o valor médio⁶ de referências apresentadas com a pesquisa das palavras-chave *mainframe*, *minicomputer* e *microcomputer*. Os dados de referência apresentados na tabela seguinte foram obtidos no dia 23/09/2010 por volta das 11h.

Número de referências encontradas					
Palavra-chave	Google	MSN	Yahoo	Média	%
MAINFRAME	11.200.000	10.900.000	32.800.000	18.300.000	57
MINICOMPUTER	521.000	163.000	1.250.000	644.667	2
MICROCOMPUTER	28.700.000	1.910.000	8.490.000	13.033.333	41

Com base nos valores médios encontrados, a Figura 1.17 mostra a divisão de participação de cada segmento computacional segundo as referências encontradas de cada palavra-chave frente ao mercado mundial.

Mesmo não sendo a Figura 1.17 a representação real de participação no mercado de cada um dos segmentos computacionais, ela fornece uma ideia do número de referências encontradas para as palavras-chave *mainframe*, *minicomputer* e *microcomputer*.

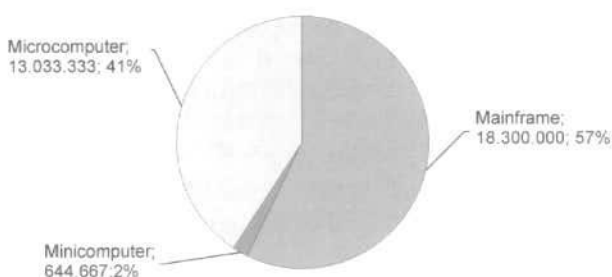


Figura 1.5 - Mercado computacional mundial.

Os computadores de grande porte representam uma grande parcela de mercado no que tange ao volume médio de referências encontradas para a palavra-chave *mainframe* nos serviços de busca (média de 57%). Isso denota alto grau de interesse do público nesse segmento. Apesar de visto por algumas pessoas com certo descrédito, este é um segmento utilizado por grandes empresas e costuma pagar excelentes salários, pois existe escassez de profissionais. Para trabalhar nesse segmento, exige-se alto grau de experiência nas áreas de computação e de negócios.

Os minicomputadores representam uma parcela de mercado pequena (média de 2%) em relação ao número médio de referências encontradas para a palavra-chave *minicomputer* nos serviços de busca. Esse segmento faz frente ao segmento de grande porte e costuma pagar ótimos salários; seu volume de concorrência profissional é de médio a pequeno. Para trabalhar nesse segmento, exige-se de médio a alto grau de experiência nas áreas de computação e de negócios.

Os microcomputadores, que são mais populares, representam uma parcela de mercado bastante elevada (média de 41%) em relação ao número médio de referências encontradas para a palavra-chave *microcomputer*. Esse segmento paga, em média, os menores salários de mercado, com exceção de alguns serviços específicos na área de *hardware* e desenvolvimento de *software* para equipamentos da linha *handhelds*. O volume de concorrência profissional é muito grande, o que diminui o valor dos salários pagos. Para trabalhar nesse segmento, exige-se de pequeno a médio grau de experiência na área de computação. O conhecimento na área de negócios geralmente não é exigido, mas esse quadro vem mudando nos últimos anos.

⁶ Média aritmética.

1.3 - Linguagens de Programação

Para que um computador eletrônico funcione, necessita ser programado. O processo de programação é uma "conversa" controlada entre um ser humano e o computador propriamente dito. O processo de comunicação se faz com o uso de uma linguagem de programação que o computador entenda. É possível comunicar-se com um computador utilizando linguagem de baixo ou alto nível.

As linguagens de baixo nível possibilitam uma comunicação em termos de máquina. É uma forma pouco convencional e de grande dificuldade para a maioria dos seres humanos. Destacam-se nessa categoria as linguagens de *máquina* e *assembly*. A linguagem *assembly* (e não *assembler*) é mais fácil de usar do que uma linguagem de máquina, por ser baseada em comandos de instruções em formato mnemônico (siglas com significado definido de suas ações).

As linguagens de alto nível possibilitam maior facilidade de comunicação com um computador, pelo fato de serem próximas à comunicação humana, pois baseiam-se em palavras do idioma inglês. Destacam-se, nessa categoria, linguagens de programação como FORTRAN, COBOL, BASIC, PASCAL, C, JAVA, Lua, C++, entre outras. Esse tipo de linguagem é mais facilmente assimilado por seres humanos. Assim sendo, o número de pessoas que as conhece é bastante grande.

As linguagens de programação encontram-se divididas em quatro categorias de gerações, sendo: primeira geração representada pelas linguagens de *máquina* e *assembly*; segunda geração representada pelas linguagens FORTRAN (primeira linguagem de alto nível), ALGOL, COBOL e, de certa forma, a linguagem BASIC; terceira geração representada pelas linguagens PL/1, PASCAL, C, MODULA-2, C++, JAVA, Lua e ADA; quarta geração representada basicamente por linguagens de consulta estruturada, além de outras ferramentas, destacando-se a linguagem de consulta estruturada SQL.

O número de linguagens de programação existentes e em operação é bastante extenso, como pode ser observado no sítio "*The Language List*"⁷ em que estão classificadas 2.500 linguagens de programação. Esse catálogo não é completo, visto que existem muitas linguagens que são de uso exclusivo de empresas e órgãos governamentais em todo o mundo.

As linguagens de programação de computadores, de acordo com o catálogo do sítio "*The Language List*", podem ser classificadas segundo suas características funcionais. A seguir apresentam-se, da lista indicada, as mais importantes:

- ▶ *Linguagem procedural* - possibilita o desenvolvimento da programação de forma estruturada, permitindo a construção de rotinas por meio de módulos de procedimentos ou de funções que estejam interligados, sendo por vezes classificada como *linguagem imperativa*.
- ▶ *Linguagem declarativa* - possibilita o desenvolvimento de programação normalmente estática, sendo por vezes classificada como *linguagem de marcação*.
- ▶ *Linguagem orientada a objetos* - permite o desenvolvimento de composições e interações de programas entre várias unidades de programa denominadas objetos, em que objeto é um elemento abstrato que representa uma entidade do mundo real em um computador.
- ▶ *Linguagem concorrente* - possibilita o desenvolvimento de programas com características de execução de rotinas em paralelo e não da forma sequencial normalmente encontrada nas outras categorias.

⁷ <http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm> (sítio acessado em abril de 2010)

- ▶ *Linguagem de consulta* - garante acesso para a extração de informações em bases de dados existentes em programas de gerenciamento de bancos de dados.
- ▶ *Linguagem de especificação* - define a documentação descritiva em alto nível de um sistema, sendo por vezes classificada como *linguagem de projeto de programação*.

No aspecto de operação de uma linguagem de programação de computadores, esta pode ser aplicada para o desenvolvimento de programação científica, comercial ou de uso geral. Não existem motivos partidários para achar que uma linguagem de programação é melhor do que outra, uma vez que cada linguagem é desenvolvida para atender determinados problemas e aplicações. A melhor linguagem de programação para um programador normalmente é aquela em que ele sabe programar, mas nunca deixe de estar pronto para aprender novas linguagens de programação.

1.4 - Paradigmas de Programação

O modelo (o estilo ou o paradigma) da forma de programar computadores eletrônicos também sofreu mudanças desde seu surgimento, saindo de um formato simples para um formato mais complexo. De acordo com Román (2008, p. 437), os paradigmas de programação passaram por cinco fases evolucionárias, sendo programação tradicional, programação estruturada, programação modular, programação com abstração de dados e programação orientada a objetos.

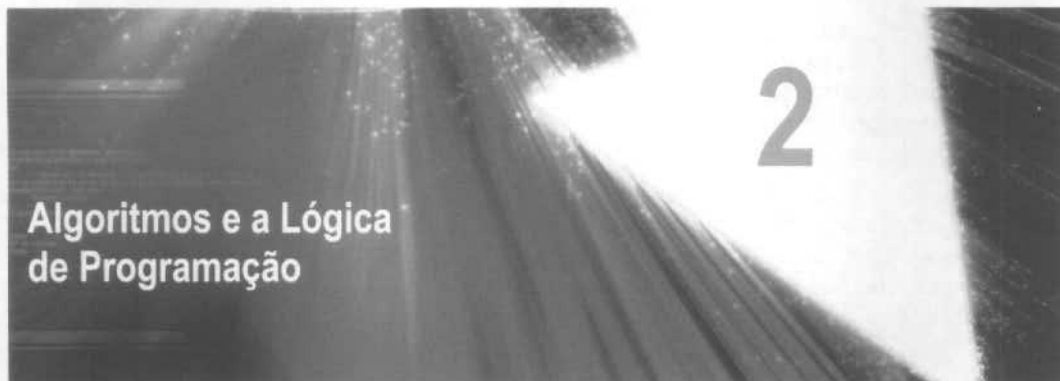
A programação tradicional iniciou-se no final da década de 1950. Teve como maiores representantes as linguagens de programação FORTRAN, COBOL e BASIC, nas quais se encontram as estruturas de programação de sequência, desvio condicional simples, desvio condicional composto e laço de repetição incondicional.

A programação estruturada iniciou-se em meados da década de 1960 com o surgimento da linguagem de programação PASCAL. Posteriormente ocorreram mudanças nas estruturas das linguagens de programação FORTRAN, COBOL e BASIC para que suportassem os paradigmas da programação estruturada. Na década de 1970 surgiu a linguagem de programação C. Nessas linguagens de programação encontram-se, além das estruturas tradicionais, as estruturas de programação de laços de repetição condicionais e seleção, paradigma que é o assunto central desta obra.

A programação modular iniciou-se no final da década de 1970 com o surgimento da linguagem de programação MODULA-2, na qual se encontram, além das estruturas de programação existentes em outros paradigmas, o uso de encapsulamento de módulos com pacotes de dados e funções.

A programação com abstração de dados iniciou-se na década de 1980 com a linguagem de programação ADA, na qual se encontram muitas das estruturas de programação existentes em outras linguagens, além de possuir a estrutura de programação de laço de repetição condicional seletivo (também encontrada na linguagem BASIC estruturada) e ter introduzido o conceito de dados abstratos.

As ideias do paradigma da programação orientada a objetos iniciaram-se na década de 1960 com o surgimento das linguagens SIMULA I, depois com SIMULA 67 (1967), tendo seu auge de percepção pelo mercado a partir da década de 1980 com o surgimento da linguagem SMALLTALK (iniciada por volta da década de 1970) nos laboratórios da empresa XEROX), depois com as linguagens C++ (1980), EIFFEL, Object PASCAL (ambas em 1985), e posteriormente com as linguagens JAVA (1995) e C# (2000), nas quais se encontram a definição e construção de classes, objetos, encapsulamento, herança e polimorfismo, que ao final desta obra são apresentados a título de introdução.



Este capítulo apresenta informações sobre o significado da palavra algoritmo e sua aplicação computacional. Outro ponto de destaque é a orientação em relação aos níveis de conhecimento de um programador de computador. Aborda detalhes sobre o uso de lógica de programação de computadores e da norma ISO 5807:1985 (E) na elaboração de diagramas de blocos. Descreve a técnica de linguagem de projeto de programação e, por fim, orienta quanto ao uso de compiladores, interpretadores e tradutores. A leitura deste capítulo é indicada a todos, tenham ou não experiência no uso de computadores.

2.1 - Algoritmos Computacionais

O termo *algoritmo* normalmente causa certa estranheza a algumas pessoas, pois muitas acreditam que está escrito ou pronunciado de forma incorreta. A palavra *algoritmos* vem do latim, dos termos *algorismos* ou *algorithmos* que estão associados à ideia de algarismos por influência do idioma grego a partir do termo *arithmós* associado à ideia de números.

A palavra *algoritmo* é aplicada e empregada, segundo o dicionário Aurélio, em matemática e computação. Na esfera matemática está associada a um processo de cálculo, ou de resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições, regras formais para a obtenção do resultado, ou da solução do problema. Na ciência da computação (informática) está associada a um conjunto de regras e operações bem definidas e ordenadas, destinadas à solução de um problema, ou de uma classe de problemas, em um número finito de passos.

Há também uma crença na área da computação de que o termo *algoritmos* originou-se no ano de 830 d.C. a partir da publicação de um importante livro de álgebra escrito por um famoso matemático e astrônomo muçulmano que viveu na Pérsia (atual Irã), chamado Abu Abdullah Muhammad ibn Musa al-Khwarizmi, nascido no ano de 780 d.C. e falecido no ano de 850 d.C., Figura 2.1, conhecido também por al-Khwarizmi.

Acredita-se que seu nome sofreu corruptelas ao longo dos anos, levando a mudanças na pronúncia para Al-Karismi, Algarismi, chegando a algarismo, que é a representação numérica do sistema de cálculos atualmente em uso. Supõe-se que é deste mesmo radical que vem o termo *algoritmo*, utilizado na área da computação.

O termo *algoritmo* é também referenciado nas várias obras publicadas sobre os temas relacionados às atividades de programação de computadores com as mais diversas descrições, podendo ser um conjunto de regras formais para a obtenção de um resultado ou da solução de um problema (FORBELLONE & EBERSPACHER, 2000), englobando fórmulas de expressões aritméticas.



Figura 2.1 - Muhammad ibn Musa al-Khwarizmi.⁸

Um *algoritmo* pode ser descrito basicamente de duas formas (BERG & FIGUEIRÓ, 1998): uma forma gráfica a partir da utilização de diagramas de blocos, e outra forma textual a partir de uma linguagem de projeto de programação ou mesmo de uma linguagem de programação de computadores formal. Num sentido mais amplo, *algoritmo* é um processo sistemático para a resolução de um problema (SZWARCFITER & MARKENZON, 1994), ou de uma sequência ordenada de passos a ser observada para a realização de uma tarefa (SALIBA, 1992) e (BERG & FIGUEIRÓ, 1998) Para Berlinski (2002, p. 21), algoritmo é um método finito, escrito em um vocabulário simbólico fixo regido por instruções precisas, que se movem em passos discretos, 1, 2, 3, ..., cuja execução não requer *insight*, esperteza, intuição, inteligência ou clareza e lucidez, e que mais cedo ou mais tarde chega a um fim.

A partir das definições dadas, o termo algoritmo, do ponto de vista computacional, pode ser entendido como *regras formais, sequenciais e bem definidas a partir do entendimento lógico de um problema a ser resolvido por um programador com o objetivo de transformá-lo em um programa que seja possível de ser tratado e executado por um computador.*

Desta forma, é importante entender e compreender a palavra "problema". Pode-se dizer que problema é uma proposta duvidosa, que pode ter numerosas soluções, ou questão não solvida e que é objeto de discussão, segundo definição encontrada no dicionário Aurélio. No entanto, do ponto de vista computacional, problema é uma questão que foge a uma determinada regra, ou melhor, é o desvio de um percurso, o qual impede de atingir um objetivo com eficiência e eficácia.

2.2 - Cozinha x Computador

Do ponto de vista computacional, um algoritmo pode ser didaticamente comparado a uma receita culinária. Toda receita culinária é dividida em dois blocos de ação, sendo o *bloco ingredientes* no qual se definem os dados a serem usados e as quantidades que devem estar preparadas e separadas para a elaboração da receita, e o *bloco modo de preparo* em que estão descritos o programa de ações e a sequência de atividades.

Note que tanto o bloco de ingredientes como o bloco do modo de preparo estabelecem o roteiro de trabalho a ser seguido (o programa de atividades). Se uma das etapas definidas em toda a receita deixar de ser seguida, ter-se-á no final um resultado diferente do preestabelecido na receita.

Qualquer pessoa de posse de uma receita consegue, se seguir os seus passos, preparar a refeição indicada sem grandes dificuldades; essas pessoas são os *cozinheiros*. No entanto, existem outras que criam e inventam receitas, normalmente chamadas de *mestres-cucas*. Há ainda uma terceira pessoa que sim-

⁸ Imagem obtida no site <http://www.orqcie.splinder.com/post/9977179> em outubro de 2010.

plesmente vai provar a refeição; é o usuário da receita preparada pela pessoa que cozinhou, que não é necessariamente um mestre-cuca, mas alguém que, ao longo de um determinado tempo, pode se tornar um.

Na atividade de programação de computadores há etapas de trabalho semelhantes às de uma cozinha. O programador de computador é, de certa forma, um "mestre-cuca" da programação, pois prepara o programa a ser utilizado por uma pessoa denominada *usuário*. O usuário quer consumir, usar um programa como alguém que entra em um restaurante e deseja comer um alimento. Ele não está preocupado com a maneira como o alimento foi preparado (ou como um programa de computador foi escrito), simplesmente quer usá-lo.

Existem pessoas que montam programas a partir de algoritmos escritos por programadores. São, na verdade, simples "cozinheiros", o que não é demérito nenhum, porém não têm experiência para resolver todos os problemas que possam surgir.

O programador de computador "mestre-cuca" é um profissional que possui muita sensibilidade em lógica de programação, normalmente com muita experiência, algo em torno de dez anos de trabalho ou até mais tempo (NORVIG, 2007); é um profissional em nível *sênior*. Consegue elaborar algoritmos computacionais extremamente refinados e avançados, sem que precise de um modelo previamente definido para conseguir resolver um problema.

O programador de computador "cozinheiro" é um profissional que possui pouca ou média sensibilidade em lógica de programação, normalmente com pouca experiência na tarefa de programar computador, algo em torno de quatro a seis anos (no nível *júnior*) ou entre sete e nove anos (no nível *pleno*). O profissional *júnior* consegue desenvolver algoritmos de nível simples a médio e necessita do auxílio e da supervisão contínua de um programador *sênior*. O profissional *pleno* consegue desenvolver algoritmos de nível médio a complexo e necessita de supervisão e auxílio esporádicos de um programador *sênior*. Ao usar algoritmos prontos e escritos por outros programadores, gradativamente o programador *júnior* é preparado até se tornar *pleno*, e o programador *pleno* se prepara para chegar a um nível *sênior* e tornar-se, de certa forma, um "mestre-cuca".

Assim como uma cozinha necessita de ajudantes de cozinheiro, também é necessário haver ajudantes de programador. Esses profissionais normalmente possuem pouca experiência, algo entre um e três anos (*trainee*). Devido ao pouco tempo de experiência não assumem tarefas complexas. Normalmente montam programas a partir de algoritmos já escritos por programadores mais experientes. Quando escrevem algum algoritmo, usam de mais simplicidade que o programador *júnior*. Geralmente precisam de auxílio e supervisão rígida de um programador *sênior*.

Um fato deve ficar claro. Ninguém chega ao auge de uma profissão começando de cima. É necessário dar os primeiros passos devagar, adquirir experiência e vivência e, ao longo de um período, galgar os degraus da profissão escolhida. Assim sendo, não existe muito tempo na vida para cometer erros profissionais. Antes de escolher alguma profissão, pense muito bem, procure conhecê-la, bem como os pontos positivos e negativos, e seja tenaz e disciplinado no aprendizado para conseguir conquistar seu espaço.

2.3 - Lógica de Programação de Computadores

Muitos gostam de afirmar que possuem e sabem usar o raciocínio lógico, porém quando questionados direta ou indiretamente, perdem essa linha de raciocínio, pois inúmeros fatores são necessários para completá-lo, tais como conhecimento, versatilidade, experiência, criatividade, responsabilidade, ponderação, calma, autodisciplina, entre outros.

Para usar o raciocínio lógico, é necessário ter domínio do pensar, bem como saber pensar, ou seja, possuir e usar a "arte de pensar". Alguns definem o raciocínio lógico como um conjunto de estudos que visa determinar os processos intelectuais que são as condições gerais do conhecimento verdadeiro. Isso é válido para a tradição filosófica clássica aristotelista. Outros preferem dizer que é a sequência coerente, regular e

necessária de acontecimentos, de coisas ou fatos, ou até mesmo a maneira do raciocínio particular que cabe a um indivíduo ou a um grupo. Estas são algumas definições encontradas no dicionário Aurélio, mas existem outras que expressam o verdadeiro raciocínio lógico dos profissionais da área de tecnologia da informação (da área da computação), tais como um esquema sistemático que define as interações de sinais no equipamento automático do processamento de dados, ou o computador científico com o critério e princípios formais de raciocínio e pensamento.

A partir do exposto pode-se dizer que lógica é a ciência que estuda as leis e os critérios de validade que regem o pensamento e a demonstração, ou seja, ciência dos princípios formais do raciocínio.

Usar raciocínio lógico é um fator a ser considerado por todos, mas principalmente pelos profissionais da área de tecnologia de informação envolvidos com o desenvolvimento da programação de computadores (destacando-se programadores, analistas de sistemas e analistas de suporte), pois seu dia a dia dentro das organizações é solucionar problemas e atingir os objetivos apresentados por seus usuários com eficiência e eficácia, utilizando recursos computacionais e/ou automatizados mecanicamente. Saber lidar com problemas de ordem administrativa, de controle, de planejamento e de estratégia requer atenção e boa performance de conhecimento do pensar e do realizar. Porém, é necessário considerar que ninguém ensina ninguém a pensar, pois as pessoas nascem com esse "dom".

O objetivo central desta obra é mostrar como desenvolver e aperfeiçoar essa técnica de pensar, considerando-a no contexto da programação de computadores. É fundamental que o estudante de programação aprenda a pensar o "computador", ou seja, é necessário modelar o pensar e o raciocínio ao formato operacional preestabelecido e funcional de um computador eletrônico. Essa tarefa não é trabalho fácil e exige muita determinação, persistência e autodisciplina por parte do estudante de programação, pois para aprender a programar um computador é necessário executar repetidamente diversos exercícios e praticá-los constantemente, o que leva à exaustão física e mental.

As bases que norteiam o processo da programação de computadores vêm das mesmas ideias estudadas e apresentadas por Charles Babbage com a máquina analítica e da programação idealizada por sua assistente, Ada Augusta Byron King, Figura 2.2, condessa de Lovelace, nascida em 10 de dezembro de 1815 e falecida em 27 de novembro de 1852, filha de George Gordon Byron, conhecido como Lord Byron, destacado poeta britânico.

Ada Augusta é considerada o primeiro programador de computadores⁹, pois desenvolveu os algoritmos que permitiriam à máquina analítica computar os valores de funções matemáticas. Publicou também uma série de notas sobre a máquina analítica, que criou muitas bases utilizadas atualmente para programar computadores.



Figura 2.2 - Ada Augusta.¹⁰

⁹ O termo *programador de computador* está associado à pessoa que projeta, escreve e testa programas para computadores. Já o termo *programador* está relacionado à ocupação profissional de uma pessoa e não ao gênero. O substantivo masculino *programador* não é flexionado por ser usado como gênero comum de dois. Cabe ressaltar que o termo *programadora* (substantivo feminino) é associado à pessoa jurídica que produz e/ou fornece programas ou programações para a televisão.

¹⁰ Imagem obtida em http://upload.wikimedia.org/wikipedia/commons/8/87/Ada_Lovelace.jpg em outubro de 2010.

2.3.1 - Uso de Lógica na Programação de Computadores

Muitos profissionais da área de programação de computadores (principalmente os mais experientes, cautelosos e cuidadosos) preferem elaborar seus programas com base em um projeto que aborde todos os aspectos técnicos do desenvolvimento, com atenção especial sempre à parte do projeto lógico.

O projeto de desenvolvimento de sistemas (conjunto de programas de computador com o mesmo propósito e interligados) segue diversas etapas de um processo normalmente conhecido como análise de sistemas. O foco desta obra é o projeto lógico, a parte do desenvolvimento do programa de um sistema em si. Assim sendo, apenas os aspectos relacionados ao desenvolvimento e à escrita de rotinas de programas e seu projeto serão abordados.

Normalmente, o projeto lógico de um programa ou conjunto de programas é idealizado utilizando ferramentas gráficas e textuais:

- ▶ As ferramentas gráficas utilizadas no projeto lógico da programação podem ser os *diagramas de blocos* (e não *fluxogramas* como alguns profissionais referem, baseados na norma internacional ISO 5807:1985 (E)) ou *diagramas de quadros* (modelo conhecido também como *NS - lê-se enés*, *NSD - lê-se enésidi* ou *Chapin - lê-se chapâm*). O uso dessas ferramentas gráficas possibilita demonstrar de forma concreta a linha de raciocínio lógico (que é um elemento abstrato) que o profissional de desenvolvimento usou para escrever um programa de computador. Os *diagramas de blocos* ou *de quadros* são instrumentos que estabelecem visualmente a sequência de operações a ser efetuada por um programa de computador. A técnica de uso e desenvolvimento de diagramas concede ao profissional da área de desenvolvimento facilidade na posterior codificação e também manutenção do programa em qualquer uma das linguagens formais de programação existentes. Na elaboração dos diagramas não se levam em consideração particularidades e detalhamentos sintáticos e estruturais utilizados por uma linguagem de computador, e sim apenas as ações a serem realizadas. Os diagramas são ferramentas que possibilitam definir o detalhamento operacional que um programa deve executar, sendo um instrumento tão valioso quanto é uma planta para um arquiteto ou engenheiro.
- ▶ As ferramentas textuais (*pseudocódigos* ou *metalinguagens*) permitem descrever de forma simples e sem o rigor técnico de uma linguagem de programação formal (uso de parênteses, pontuações e parâmetros) as etapas que o programa de computador deve executar, desde que essas etapas estejam definidas e delineadas com uma das ferramentas gráficas existentes: diagramas de blocos ou diagramas de quadros. O modelo de codificação textual de um programa pode ser usado com base na técnica chamada PDL (*Program Design Language*) que é uma linguagem de projeto de programação e não uma linguagem de programação em si, servindo como propósito inclusive de documentação. No Brasil essa técnica é utilizada normalmente com os nomes *português estruturado* ou *portugol*. Esta obra usa a referência *português estruturado*.

A técnica mais importante no projeto da lógica de programas baseada em algoritmos denomina-se programação estruturada ou programação modular, estando em consonância com o pensamento, que é estruturado e serve como base e fundamentação para o uso e estudo de outras técnicas, como a técnica de programação orientada a objetos.

A técnica de programação estruturada usa uma metodologia de projeto, que objetiva:

- ▶ agilizar a codificação da escrita da programação;
- ▶ facilitar a depuração da leitura;
- ▶ permitir a verificação de possíveis falhas apresentadas pelos programas;
- ▶ permitir a reutilização de código dentro do próprio programa ou em outros programas com a criação de bibliotecas;
- ▶ facilitar as alterações e atualizações dos programas, bem como o processo de manutenção. E deve ser composta de quatro passos fundamentais:

- ▶ Escrever instruções ligadas entre si apenas por estruturas sequenciais, tomadas de decisão, laços de repetição e de selecionamento.
- ▶ Escrever instruções em grupos pequenos e combiná-las na forma de sub-rotinas ou de módulos estruturados ou orientados a objeto.
- ▶ Distribuir módulos do programa entre os diferentes programadores que trabalharão sob a supervisão de um programador *sênior*, chefe de programação ou analista de sistemas de informação.
- ▶ Revisar o trabalho executado em reuniões regulares e previamente programadas, em que compareçam apenas programadores de um mesmo nível.

A programação de computadores é uma prática que necessita de metodologia, disciplina e autoconhecimento do pensar. É necessário direcionar o pensamento, dentro da limitação técnica de um computador, para atender às necessidades dos usuários, que veem neste a solução de todos os problemas que nem sempre podem ser solucionados pelo computador, ou seja, atendidos.

Dentro do contexto e do princípio da lógica de programação de computadores, é fundamental deixar claro o papel de dois dos mais importantes profissionais desse contexto, que são o *analista de sistema de informação* (*analista de sistemas*) e o *programador de computador*.

O papel do analista de um sistema é semelhante ao papel executado por um arquiteto. O analista é responsável por obter a visão macro que o sistema deve executar e passá-la para a equipe de programação que deve então delineá-la. Assim como um arquiteto, o analista de sistemas tem a responsabilidade de desenhar a planta do sistema (planta estrutural), a qual é chamada, de forma genérica, de *fluxograma*. Os fluxogramas são diagramas que fornecem a representação esquemática de um processo computacional e de sua organização funcional. Em hipótese nenhuma um fluxograma, como ferramenta de projeto, é usado para demonstrar a funcionalidade lógica de um programa. Isso é feito apenas com diagramas de blocos ou mesmo com diagramas de quadros.

A função do programador de computador é semelhante à de um construtor (num nível mais avançado, é semelhante à função de um pedreiro altamente especializado, pois o programador é responsável por construir o programa, empilhando as instruções de comando de uma linguagem de programação como se fossem tijolos e, inclusive, fazer o acabamento da construção com a elaboração do projeto visual chamado de interface gráfica).

Além de interpretar o fluxograma desenhado pelo analista, o programador de computador precisa destrinchar os detalhes lógicos que o programa deve executar para atender ao que fora solicitado pelo analista e, conseqüentemente, atender à necessidade de um usuário. É nessa fase do trabalho que o programador deve desenhar num nível micro a planta operacional com as atividades a serem executadas pelo programa. Essa planta operacional é chamada de diagrama de blocos ou diagrama de quadros.

2.3.2 - Norma ISO 5807:1985 (E)

O processo de desenvolvimento de programação de computadores ocorre, normalmente, baseado em duas etapas de trabalho, que são análise de sistemas e programação, quando são feitos os desenhos dos fluxogramas, diagramas de bloco ou de quadros.

Por ser um trabalho voltado ao desenvolvimento da lógica de programação utilizada em computadores eletrônicos, o foco ora apresentado é baseado na construção de diagramas de blocos e pseudocódigos. O estilo em diagrama de quadros não será apresentado, uma vez que esse modelo não é aceito como norma oficial de trabalho, tendo um apelo mais acadêmico do que comercial.

A norma internacional ISO 5807:1985 (E) é a consolidação de duas normas anteriores denominadas ISO 1028¹¹ e ISO 2636¹², ambas publicadas no ano de 1973. Em particular, a extinta norma internacional ISO 1028 foi editada a partir da norma regional norte-americana ANSI X3.5¹³ publicada no ano de 1970.

Segundo Pressman (1995, p. 453), "um quadro¹⁴ vale mil palavras, mas é importante diferenciar qual quadro e quais mil palavras" pretende-se realmente referenciar. A importância da representação gráfica da linha de raciocínio lógico é considerada também por Berg & Figueiró (1998, p. 18), quando afirmam que as representações gráficas implicam ações distintas, deixando claro que "tal propriedade facilita o entendimento das ideias (...) e justifica a sua popularidade". A importância da representação gráfica da lógica de programação de computadores não é uma discussão técnica recente, pois já fora apresentada por Goldstein & Von Neumann no ano de 1947.

Segundo a norma ISO 5807:1985 (E), seu uso não deve restringir aplicações ou soluções particulares, uma vez que podem existir várias soluções para os diversos problemas de processamento de informação. Assim sendo, essa norma sugere o uso de critérios que devem ser adaptados segundo as necessidades existentes.

Os símbolos gráficos da norma ISO 5807:1985 (E) permitem demonstrar de forma clara a linha de raciocínio lógico utilizada por um programador de computadores, de forma que seja fácil a quem não conhece programação entender o que se pretende de um determinado programa. A Figura 2.3 mostra como exemplo a imagem de um gabarito que facilita o desenho de fluxogramas e de diagramas de blocos, baseado num gabarito original desenvolvido pela empresa IBM. Nesse gabarito existem símbolos padrão (incluídos e aceitos na norma internacional) e símbolos particulares de uso da IBM, que foram criados por ela, para atender às próprias necessidades.

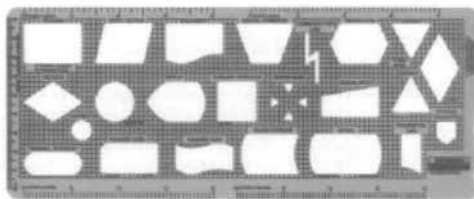


Figura 2.3 - Gabarito para diagramação de programas.
Fonte: Rapidesign.





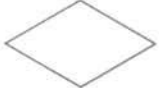




A tabela seguinte apresenta apenas alguns dos símbolos normatizados e de uso geral que serão usados nesta obra.

¹¹ Information processing - Flowchart symbols

¹² Information processing - Conventions for incorporating flowchart symbols in flowcharts

¹³ American National Standards

¹⁴ O termo *quadro* é utilizado no sentido figurado para representar o termo *símbolo*.

Símbolo	Significado	Descrição
	Terminal <i>Terminator</i>	O símbolo representa a definição de início e fim do fluxo lógico de um programa. Também é utilizado na definição de sub-rotinas de procedimento ou de função.
	Entrada manual <i>Manual input</i>	Representa a entrada manual de dados, normalmente efetuada em um teclado conectado diretamente ao console do computador.
	Processamento <i>Process</i>	Representa a execução de uma operação ou grupo de operações que estabelecem o resultado de uma operação lógica ou matemática.
	Exibição <i>Display</i>	Representa a execução da operação de saída visual de dados em um monitor de vídeo conectado ao console do computador.
	Decisão <i>Decision</i>	O símbolo representa o uso de desvios condicionais para outros pontos do programa de acordo com situações variáveis.
	Preparação <i>Preparation</i>	Representa a modificação de instruções ou grupo de instruções existentes em relação à ação de sua atividade subsequencial.
	Processo predefinido <i>Predefined process</i>	Definição de um grupo de operações estabelecidas como uma sub-rotina de processamento anexa ao diagrama de blocos.
	Conector <i>Connector</i>	Representa a entrada ou a saída em outra parte do diagrama de blocos. Pode ser usado na definição de quebras de linha e na continuação da execução de decisões.
	Linha <i>Line</i>	O símbolo representa a ação de vínculo existente entre os vários símbolos de um diagrama de blocos. Possui a ponta de uma seta indicando a direção do fluxo de ação.

A partir de uma rápida definição dos símbolos que podem ser usados e de suas descrições é pertinente estabelecer algumas regras para sua devida utilização, em consonância com a norma apresentada. Desta forma, atente aos pontos seguintes:

- ▶ Os símbolos de identificação gráfica representam sempre uma operação ou conjunto de operações similares, podendo ser identificados por um rótulo relacionado à própria ação do símbolo em uso, somente quando necessário.
- ▶ Os símbolos devem ser conectados uns aos outros por linhas de setas que mostrem explicitamente a direção do fluxo a ser executado pelo programa.
- ▶ A estrutura visual do diagrama deve, a princípio, estar orientada no sentido de cima para baixo, da direita para a esquerda e ser desenhada no centro da folha de papel. No entanto, dependendo da situação, esse critério pode ser alterado, o que leva à necessidade de manter o uso das linhas de seta indicando a direção do fluxo.
- ▶ A definição de inicialização e finalização de um diagrama ocorre com o uso do símbolo "terminal" devidamente identificado com um dos rótulos: início, fim, retorno ou a definição de um nome particular, quando for necessário, desde que seguidas as regras de utilização de sub-rotinas (a serem apresentadas em momento oportuno).

- ▶ As operações de entrada de dados para esta obra serão genericamente representadas com o uso do símbolo de "entrada manual", uma vez que o teclado será utilizado como periférico genérico dessa ação.
- ▶ As operações de saída de dados serão genericamente definidas com o símbolo "exibição", considerando o fato de um monitor de vídeo estar sempre em uso.
- ▶ A definição das variáveis nas operações de entrada e saída será feita nos símbolos apropriados. Quando houver mais de uma variável a ser utilizada, serão separadas por vírgulas.
- ▶ As operações de processamento matemático e lógico estarão definidas com o símbolo "processamento". Quando houver mais de uma operação a ser definida em um mesmo símbolo, devem estar separadas por linhas de ação sem o uso de vírgulas ou serem escritas em símbolos distintos.
- ▶ As operações de tomada de decisão para condições simples, compostas, sequenciais, encadeadas ou de execução de laços interativos (condicionais) serão representadas pelo símbolo de "decisão", que conterá a condição a ser avaliada logicamente. Cada símbolo de decisão pode possuir apenas uma condição lógica. É considerada lógica uma condição isolada ou de um conjunto de condições vinculadas com o uso de um operador lógico de conjunção, disjunção ou disjunção exclusiva.
- ▶ As operações de laços iterativos e não iterativos (incondicionais) serão representadas com o símbolo "preparação" que permite a realização de um grupo de tarefas predefinidas e relacionadas.
- ▶ A definição e o uso de sub-rotinas são representados pelo "processo predefinido". Esse símbolo deve estar identificado com um rótulo associado a outro diagrama de bloco ou blocos interdependente ao programa e indicado como rótulo do símbolo "terminal" da rotina associada.
- ▶ As operações que necessitarem de conexão utilizarão o símbolo "connector" na finalização de operações de decisões ou na identificação de vínculos entre partes de um programa e, neste caso, devem estar identificados com rótulos alfanuméricos.
- ▶ Fica eleito o símbolo "processamento" como curinga, que pode representar qualquer ação definida ou não, desde que a operação seja devidamente identificada por um rótulo descritivo. Exceções aos símbolos "decisão" e "preparação" que representam operações bem definidas e não devem, em hipótese nenhuma, ser substituídos por qualquer outro símbolo.
- ▶ O símbolo *terminal* só pode ser usado duas vezes dentro de cada programa ou programa de sub-rotina. Em nenhuma hipótese pode este símbolo ser usado mais de duas vezes.

Além das bases anteriormente propostas e já discutidas amplamente por vários cientistas da computação, podem surgir pontos a serem definidos ao longo deste estudo ou do trabalho a ser efetivamente realizado na vida profissional. Neste caso, demais critérios que se fizerem necessários e que não estão previstos na norma ISO 5807:1985 (E) devem ser analisados, discutidos, mediados e definidos pela gerência de cada equipe de desenvolvimento.

2.3.3 - Diagrama de Bloco e de Quadro

A representação gráfica baseada nas formas geométricas apresentadas no tópico anterior implica no uso e implementação de ações distintas. O uso de diagramas facilita o entendimento das ideias de uma pessoa ou equipe e justifica sua popularidade.

A representação gráfica baseada em diagrama de bloco é também referenciada erroneamente no Brasil como fluxograma. O termo fluxograma deve ser utilizado, como já comentado, apenas na esfera da análise de sistemas, e não em programação. Possivelmente o erro de definição do termo ocorre devido à estrutura da palavra original no idioma inglês: *flowchart* (*flow* = fluxo, *chart* = diagrama), portanto diagrama de fluxo. No entanto, diagrama de fluxo não é o mesmo que fluxograma. Vale lembrar que fluxograma é um conceito macro e diagrama é micro do processo de documentação gráfica da linha de raciocínio a ser usada na programação de um computador eletrônico.

Além da representação tradicional, há a representação alternativa denominada diagrama de quadro (ou diagrama de *NS* ou diagrama de *Chapin*). Apesar de utilizada por alguns profissionais, essa forma não é aceita por normas internacionais e, por esta razão, não será abordada em detalhes, mas isso não impede de ser apresentada.

O diagrama *NS* ou *NSD* foi desenvolvido por Isaac Nassi e Ben Shneiderman nos anos de 1972/73 e ampliado por Ned Chapin no ano de 1974. Esse modelo de diagramação substitui o formato tradicional por uma forma estrutural diferente baseada no uso de quadros.

As Figuras 2.4, 2.5, 2.6, 2.7, 2.8 e 2.9 mostram, respectivamente, as estruturas lógicas de operação computacional mais triviais e comuns e que serão apresentadas ao longo desta obra: sequência, decisão simples, decisão composta, laço de repetição condicional pré-teste, laço de repetição condicional pós-teste e laço de repetição incondicional, desenhadas em diagrama de bloco (lado esquerdo das imagens) e diagrama de quadro (lado direito das imagens). As situações apresentadas se referem à demonstração de alguns simples programas de computador.

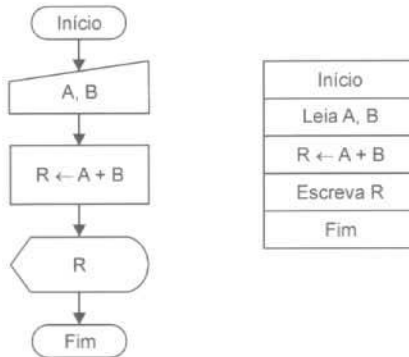


Figura 2.4 - Estrutura de operação computacional de sequência.

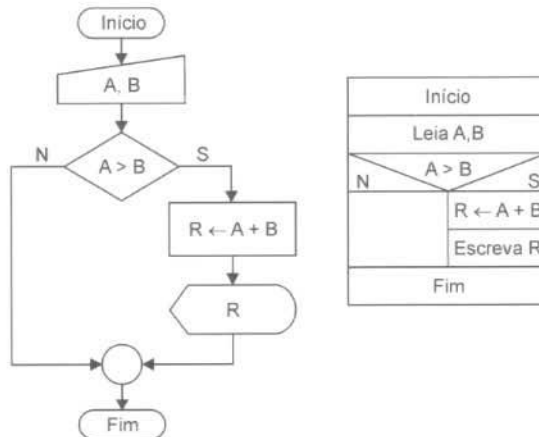


Figura 2.5 - Estrutura de operação computacional de decisão simples.

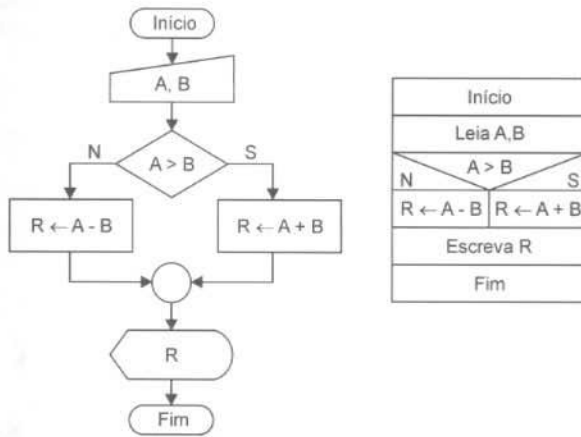


Figura 2.6 - Estrutura de operação computacional de decisão composta.

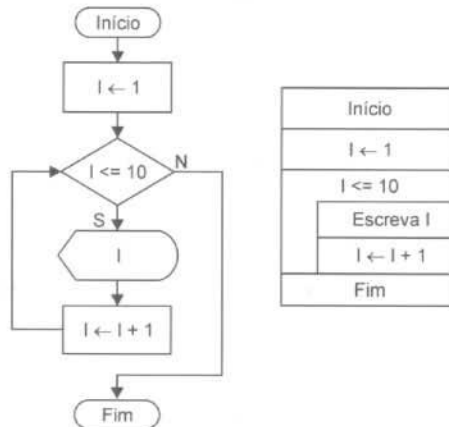


Figura 2.7 - Estrutura de operação computacional de laço de repetição condicional pré-teste.

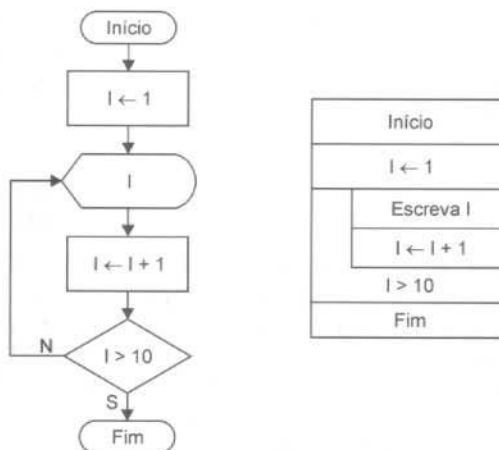


Figura 2.8 - Estrutura de operação computacional de laço de repetição condicional pós-teste.

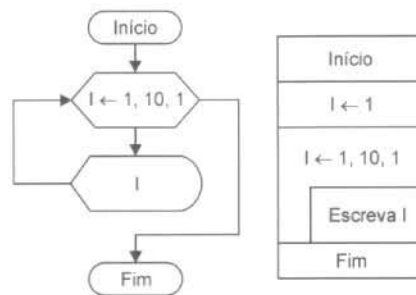


Figura 2.9 - Estrutura de operação computacional de laço de repetição incondicional.

Ao usar ferramentas gráficas para representar a linha lógica de raciocínio a ser implementada em um computador eletrônico, como é o caso dos diagramas de blocos ou dos diagramas de quadros, torna-se necessário diferenciar quatro subcategorias entre esses diagramas. Assim sendo, tanto o diagrama de bloco quanto o diagrama de quadro podem ser mencionados como:

- ▶ *diagrama de bloco ou diagrama de quadro;*
- ▶ *diagrama de blocos ou diagrama de quadros;*
- ▶ *diagramas de bloco ou diagramas de quadro;*
- ▶ *diagramas de blocos ou diagramas de quadros.*

Ao longo do tema desenvolvido no livro, esses detalhes serão apresentados.

2.3.4 - Linguagem de Projeto de Programação

No processo de desenvolvimento de programas de computadores, pode-se utilizar a forma de representação textual denominada pseudocódigo (metalguagem ou linguagem de especificação). O estilo de pseudocódigo é uma *linguagem de projeto de programação* e não uma linguagem de programação real. Assim sendo, uma *linguagem de projeto de programação* não pode e não deve ter o mesmo rigor sintático que possui uma linguagem de programação formal (linguagem de programação real). Não há motivos para que o código escrito em LPP (Linguagem de Projeto de Programação) tenha, além de comandos escritos no idioma local, outras formas de apresentação de caracteres auxiliares. No livro são usados caracteres alfanuméricos e, em algumas circunstâncias, parênteses na representação de expressões lógicas.

As linguagens de projeto de programação existentes possuem como característica principal o regionalismo, pois normalmente são expressas no idioma oficial do país em que são utilizadas. Há exemplos dessa forma de descrição em diversos idiomas. No Brasil essa forma de trabalho foi denominada *portugol*, segundo os autores Guimarães & Lages (1994), ou *português estruturado*, forma referenciada neste livro.

A ideia de usar uma LPP é herança de sua técnica irmã denominada PDL (*Program Design Language*), proposta por Caine & Gordon em 1975 a partir da publicação do artigo PDL - *A Tool for Software Design*.

Essa técnica de trabalho visa a ser uma forma preliminar de escrever um programa de computador sem preocupar-se inicialmente com o rigor da linguagem de programação real a ser utilizada. Ela possibilita, num primeiro momento, rapidez na escrita do projeto de código do programa sem preocupar-se com os rigores técnicos e sintáticos particulares de cada uma das linguagens de programação existentes, além de servir como instrumento de documentação de código de programa, facilitando os trabalhos de manutenção e de auditoria. Num segundo momento possibilita facilidade na tradução desse código preliminar em uma linguagem de programação que, de preferência, tenha características estruturadas.

Um ponto que pode parecer negativo no uso dessa técnica é a falta de conformidade no formato entre os vários profissionais do mercado, principalmente no código a ser escrito em idioma português (MANZANO, 2008). Neste sentido, torna-se necessário definir algumas regras de trabalho. Assim sendo, apresentam-se a seguir a relação de comandos da Linguagem de Projeto de Programação (LPP) de computadores utilizada nos exemplos e exercícios deste trabalho e a classificação sintática da língua portuguesa, de acordo com a representação da ação a ser executada por cada um dos comandos.

LPP	Classificação sintática
ATÉ	Preposição
ATÉ_QUE	Conjunção (de acordo com seu equivalente em inglês - <i>until</i>)
ATÉ_SEJA	Preposição com interjeição
CADEIA	Substantivo feminino
CARACTERE	Substantivo masculino
CASO	Substantivo masculino
CLASSE	Substantivo feminino
CONJUNTO	Adjetivo
CONST (constante)	Adjetivo
CONTINUA	Verbo (imperativo afirmativo)
DE	Preposição
EFETUE	Verbo (imperativo afirmativo)
ENQUANTO	Conjunção
ENQUANTO_SEJA	Conjunção com verbo
ENTÃO	Advérbio
ESCREVA	Verbo (imperativo afirmativo)
FAÇA	Verbo (imperativo afirmativo)
FIM	Substantivo masculino
FIM_ATÉ_SEJA	Substantivo masculino com preposição e com interjeição
FIM_CASO	Substantivo masculino com substantivo masculino
FIM_CLASSE	Substantivo masculino com substantivo feminino
FIM_ENQUANTO	Substantivo masculino com conjunção
FIM_FAÇA	Substantivo masculino com verbo
FIM_LAÇO	Substantivo masculino com substantivo masculino
FIM_PARA	Substantivo masculino com preposição
FIM_REGISTRO	Substantivo masculino com substantivo masculino
FIM_SE	Substantivo masculino com conjunção
FUNÇÃO	Substantivo feminino
HERANÇA	Substantivo feminino
INÍCIO	Substantivo masculino

LPP	Classificação sintática
INTEIRO	Adjetivo
LAÇO	Substantivo masculino
LEIA	Verbo (imperativo afirmativo)
LÓGICO	Adjetivo
OBJETO	Substantivo masculino
PARA	Preposição
PASSO	Substantivo masculino
PRIVADA	Substantivo feminino
PROCEDIMENTO	Substantivo masculino
PROGRAMA	Substantivo masculino
PROTEGIDA	Adjetivo
PÚBLICA	Adjetivo
REAL	Substantivo masculino
REGISTRO	Substantivo masculino
REPITA	Verbo (imperativo afirmativo)
SAIA_CASO	Verbo imperativo afirmativo com substantivo masculino
SE	Conjunção
SEÇÃO_PRIVADA	Substantivo feminino com substantivo feminino
SEÇÃO_PROTEGIDA	Substantivo feminino com adjetivo
SEÇÃO_PÚBLICA	Substantivo feminino com adjetivo
SEJA	Interjeição
SENÃO	Conjunção
TIPO	Substantivo masculino
VAR (variável)	Substantivo feminino
VIRTUAL	Adjetivo

Observe que a LPP, ou português estruturado, por ser uma linguagem hipotética de comunicação humano-máquina, está formada com estruturas de verbos, substantivos, conjunções, interjeições, preposições e adjetivos. O uso de uma linguagem de programação formal, ou seja, uma linguagem de programação de alto nível (Lua, PASCAL, BCPL, C, C++, JAVA, C#, COBOL, BASIC, FORTRAN, JavaScript, entre outras), é também baseado nessas mesmas estruturas, com a diferença de serem sempre escritas no idioma inglês, não por ser os Estados Unidos da América o maior exportador de tecnologia computacional, mas pelo fato de a ideia de constituição de *hardware* e de *software* ter sido proposta por cientistas ingleses (Charles Babbage e Ada Augusta).

Nessa regra não se aceita exceção. Toda linguagem de programação formal, para ser aceita mundialmente como ferramenta de trabalho na área da computação, é sempre definida em inglês. Veja, por exemplo, o caso da linguagem brasileira Lua desenvolvida na PUC do Rio de Janeiro, no ano de 1993, por Roberto Ierusalimschy (<http://www.lua.org>).

Cabe ressaltar que em relação à codificação manual de um programa de computador, seja ele escrito na linguagem de programação formal ou informal, ele deve ser efetuado sempre com letras maiúsculas e de fôrma (letra bastão). Nenhum código escrito manualmente deve ter letras minúsculas, tanto de fôrma como de mão (cursiva). Essa regra se aplica de forma geral no sentido de evitar erros de interpretação e de leitura de códigos escritos por outro programador. Leve em consideração que os programadores podem ter letra ilegível. Desta forma, ao escrever o código manual em letras de fôrma e maiúsculas, evitam-se muitos aborrecimentos.

Outro ponto a ser considerado em relação a escrever com letras maiúsculas de fôrma é a necessidade de diferenciar o número 0 (zero) da letra O, por isso o hábito de cortar o número zero com um traço perpendicular da direita superior para a esquerda inferior, assemelhando-o ao símbolo de conjunto vazio.

2.4 - Compiladores, Interpretadores e Tradutores

A secretária, o estudante, o escritor, assim como muitos profissionais que usam computador no trabalho, normalmente utilizam para escrever textos uma ferramenta denominada *processador de textos* (e não *editores de texto*, a menos que estejam envolvidos com a escrita de programas de computadores). Contadores, engenheiros, economistas, físicos, químicos, matemáticos e outros profissionais das áreas de cálculo usam *planilhas eletrônicas*. Publicitários, desenhistas, entre outros profissionais da área de comunicação, utilizam ferramentas gráficas como *programas de apresentação* ou *editores gráficos de desenho e fotografia*. Programadores e desenvolvedores de *software* possuem também ferramentas de trabalho, como *editores de texto*, *compiladores*, *interpretadores* e *tradutores*.

Assim que o projeto de um programa de computador está concluído, é necessário transformá-lo em um *software*. Para tanto, faz-se a tradução do projeto definido para uma linguagem de programação formal, aquela que é executada em um computador. Para a efetivação desse trabalho, é necessário fazer a escrita do código de programa em uma ferramenta de edição de textos para depois passar o programa por ferramentas de tradução, interpretação e compilação, conforme a necessidade.

A ferramenta de edição do texto de um programa, denominada *editor de texto*, consiste em um programa simples que permite ao programador escrever o texto do código do programa e, basicamente, gravá-lo. Um editor de texto possui como característica básica a capacidade de copiar e colar blocos de texto, gravar e recuperar arquivos e imprimi-los. Como exemplos de editores de texto têm-se o *EDIT* encontrado no MS-DOS, *Bloco de Notas* do Windows e o *VI* (vi-ai) encontrado no Linux e no UNIX. Não é necessário a um editor de texto possuir mais do que esses recursos, pois quando isso ocorre, o programa deixa de ser um editor e passa a ser um processador de texto, como é o caso do programa *wordPad* do Windows. Os programas *Writer* (dos pacotes BrOffice.org, OpenOffice.org ou StarOffice) e *Word* (do pacote Microsoft Office) são, na verdade, processadores de palavras e não mais processadores de texto. A diferença é que essas ferramentas possuem dicionários e recursos de correção ortográfica em mais de um idioma.

As ferramentas de tradução são programas que permitem fazer a tradução de um programa escrito em uma linguagem formal para outra. Por exemplo, imagine um programador que, apesar de saber programar e possuir excelente lógica, só sabe escrever programas-fonte na linguagem de programação PASCAL, e ele precisa entregar uma rotina de programa-fonte escrita na linguagem de programação C. O fato é que ele não sabe nada sobre a linguagem C.

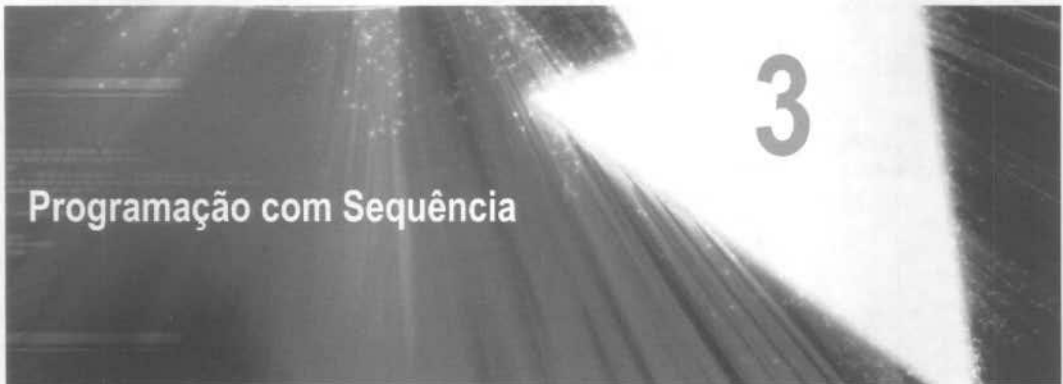
Neste caso, esse programador pode usar uma ferramenta de tradução. O programador escreve o programa-fonte na linguagem PASCAL e a ferramenta de tradução reescreve-o na linguagem C, gerando um código-fonte na linguagem C como se o próprio programador tivesse escrito. Uma ferramenta conhecida no mercado para essa finalidade é o programa **p2c**. Existem vários tipos de tradutor para diversas linguagens

de programação. São encontradas desde ferramentas de livre distribuição, como é o caso do programa *p2c*, até ferramentas comerciais.

As ferramentas de interpretação são programas que executam um programa-fonte escrito em uma linguagem na própria memória principal do computador, sem que ele seja executado diretamente no processador central da máquina. Normalmente esse tipo de ferramenta permite rápida execução dos programas, no entanto não assegura impossibilidade de acesso ao programa-fonte. Existem diversas linguagens de programação no mercado que são interpretadas, tais como BASIC, PERL, PYTHON, FORTH, JavaScript, LOGO, entre outras.

As ferramentas de compilação são programas que traduzem para uma linguagem de baixo nível (linguagem de máquina) um programa-fonte escrito em uma linguagem de alto nível. Ao fazer essa tradução, o programa-fonte se torna um programa-objeto (escrito em linguagem de alto nível compatível com o processador em uso) e depois faz a ligação do programa-objeto com as rotinas de execução de programas do sistema operacional, tornando o programa um código executável. Apesar de os programas compilados serem um pouco mais lentos quando comparados com programas interpretados, eles garantem a dificuldade de acesso ao código-fonte.

O tipo de ferramenta de programação ou de linguagem de programação usado depende de diversos fatores a serem analisados e escolhidos pela equipe de desenvolvimento.



Anteriormente, foram apresentados detalhes sobre a organização básica do computador, as unidades de medidas computacionais, as linguagens de programação, os diagramas de blocos, de quadros, linguagem de projeto de programação, entre outros. Este capítulo destaca a parte prática, enfatizando a primeira das três técnicas básicas de programação de computadores (entrada, processamento e saída de dados) denominada programação com sequência, que visa o desenvolvimento de programas simples, mas operacionais, com o uso dos tipos de dados, de variáveis e constantes, da aplicação de operadores aritméticos e das expressões aritméticas, das instruções e dos comandos de operação de um programa.

3.1 - Etapas de Ação de um Computador

Um computador eletrônico (computador digital), independentemente de ser de grande, médio ou de pequeno porte, executa basicamente três ações de trabalho, sendo:

- ▶ a entrada de dados;
- ▶ o processamento de dados;
- ▶ a saída de dados.

A etapa de *entrada de dados* é a parte em que o computador recebe os dados do mundo externo, podendo armazená-los na memória principal para realizar algum tipo de processamento, ou armazenar na memória secundária para usar futuramente. Essa etapa é realizada de forma bastante variada nas linguagens de programação, pois são muitas as formas de realização de entrada de dados.

A etapa de *processamento de dados* é quando o computador, por meio de um programa (*software*) executado em sua memória primária, faz a transformação dos dados entrados ou previamente armazenados em sua memória secundária, tornando-os elementos que possam ser usados como fontes de informação para o mundo externo. Essa etapa é realizada de forma muito comum nas linguagens de programação, pois independentemente do tipo de linguagem em uso, sofre muito pouca alteração.

Na etapa de *saída de dados* o computador envia os dados processados na memória principal ou armazenados na memória secundária para o mundo externo. Os dados processados podem ser usados como fontes de informação, e assim facilitar a vida das pessoas que necessitam tomar decisões e precisam dos computadores como ferramentas desse processo. Essa etapa, assim como a entrada de dados, é realizada de forma bastante variada nas linguagens de programação.

Todo programa de computador deve, de alguma forma, possibilitar a entrada dos dados do mundo exterior, produzir a ação de processamento, tanto lógico quanto matemático, sempre que essas ações forem necessárias, e acima de tudo possibilitar a saída de dados que tenham sido processados ou estejam apenas armazenados.

Um programa de computador é um conjunto de instruções dispostas numa ordem lógica e de forma sequencial que executam uma ação, solucionam um problema de um conjunto de dados. Assim sendo, um computador e seu programa ou programas de controle usam dois tipos de informação, sendo *dados* e *instruções*.

3.2 - Tipos de Dados Primitivos ou Dados Básicos

Os **dados** são elementos do mundo exterior, que representam dentro de um computador digital as informações manipuladas pelos seres humanos. Os dados a serem utilizados devem primeiramente ser abstraídos para serem então processados. Eles podem ser classificados em três *tipos primitivos* ou *tipos básicos*: numéricos (representados por valores numéricos inteiros ou reais), caracteres (representados por valores alfabéticos ou alfanuméricos) e lógicos (valores dos tipos falso e verdadeiro).

3.2.1 - Inteiro

Os dados numéricos positivos e negativos pertencem ao conjunto de números inteiros, excluindo qualquer valor numérico fracionário (que pertence ao conjunto de números reais), por exemplo, os valores 35, 0, 234, -56, -9, entre outros. Nesta obra a representação do dado inteiro é feita em português estruturado com o comando **inteiro**. O tipo de dado *inteiro* é utilizado em operações de processamento matemático.

3.2.2 - Real

São reais os dados numéricos positivos e negativos que pertencem ao conjunto de números reais, incluindo todos os valores fracionários e inteiros, por exemplo, os valores 35, 0, -56, -9, -45.999, 4.5, entre outros. Nesta obra o tipo de dado real será representado em português estruturado pelo comando **real**. O tipo de dado *real* é utilizado em operações de processamento matemático.

3.2.3 - Caractere/Cadeia

São caracteres delimitados pelos símbolos aspas (" "). Eles são representados por letras (de A até Z), números (de 0 até 9), símbolos (por exemplo, todos os símbolos imprimíveis existentes num teclado) ou palavras contendo esses símbolos. O tipo de dado caractere é conhecido também como alfanumérico, *string* (em inglês, cordão, colar), literal ou cadeia, por exemplo, os valores "PROGRAMAÇÃO", "Rua Alfa, 52 - Apto. 1", "Fone: (0xx99) 5544-3322", "CEP: 11222-333", " " (espaço em branco), "7", "-90", "45.989", entre outros. Nesta obra o tipo de dado caractere será representado em português estruturado pelos comandos **caractere** ou **cadeia**.

Os tipos de dados *caractere* e *cadeia* são normalmente utilizados em operações de entrada e saída de dados no sentido de apresentar mensagens de orientação para o usuário do programa em execução. O tipo de dado *caractere* é utilizado quando se faz referência a um único caractere delimitado por aspas, já o tipo de dados *cadeia* é utilizado quando se faz referência a um conjunto de caracteres delimitados por aspas.

3.2.4 - Lógico

São lógicos os dados com valores binários do tipo *sim* e *não*, *verdadeiro* e *falso*, 1 e 0, entre outros, em que apenas um dos valores pode ser escolhido. Neste livro o tipo de dado lógico será representado em português estruturado pelo comando **lógico**, utilizado em operações de processamento lógico. Para que um dado do tipo lógico seja devidamente usado, é necessário estabelecer a forma de sua representação, que neste caso será feita com os valores **.F.** (para representar falso, pode-se também fazer referência como **.FALSO.**) e **.V.** (para representar verdadeiro, pode-se também fazer referência como **.VERDADEIRO.**). Podem ainda ser utilizados os valores **.S.** ou **.SIM.** (para representar sim) e **.N.** ou **.NÃO.** (para representar não). O tipo de dado *lógico* é também conhecido como booleano, devido à contribuição do filósofo e matemático inglês George Boole à área de lógica matemática e à eletrônica digital.

3.3 - O Uso de Variáveis

Variável é tudo que está sujeito a variações, que é incerto, instável ou inconstante. E quando se fala de computadores, é preciso ter em mente que o volume de dados a serem tratados é grande e diversificado. Desta forma, os dados a serem processados são bastante variáveis.

Todo dado a ser armazenado na memória de um computador deve ser previamente identificado segundo seu tipo, ou seja, primeiramente é necessário saber o tipo do dado para depois fazer seu armazenamento adequado. Armazenado o dado desejado, ele pode ser utilizado e processado a qualquer momento.

Para entender o conceito de variável, imagine que a memória principal de um computador é um arquivo com muitas gavetas, e cada uma delas pode armazenar apenas um valor por vez, seja um dado inteiro, real, lógico ou caractere. Por ser um arquivo com várias gavetas, Figura 3.1, é necessário que cada uma das gavetas seja identificada com um nome. Desta forma, o valor armazenado pode ser utilizado a qualquer momento.

Imagine a memória de um computador como um grande arquivo com várias gavetas nas quais é possível guardar apenas um valor por vez, e como em um arquivo, essas gavetas devem estar identificadas por uma "etiqueta" com um nome.

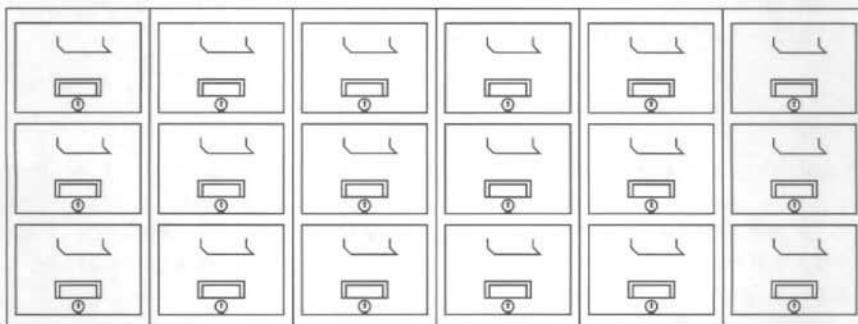


Figura 3.1 - Representação gráfica da memória de um computador com variáveis.

O nome de uma variável é utilizado para sua identificação e representação em um programa de computador. É necessário estabelecer e seguir algumas regras para uso de variáveis:

- ▶ Os nomes de identificação de uma variável podem utilizar um ou mais caracteres, limitando-se a restrições da própria linguagem formal de programação em uso. No caso do português estruturado essa restrição não existe.
- ▶ O primeiro caractere de identificação do nome de uma variável não pode ser, em hipótese nenhuma, numérico ou de símbolo gráfico (cifrão, tralha, cachimbo, vírgula, ponto e vírgula, traço, parênteses, chaves, colchetes, entre outros). O primeiro caractere de identificação do nome de uma variável deve ser sempre alfabético. Os demais caracteres do nome de uma variável podem ser alfanuméricos (números ou letras).
- ▶ Na definição de um nome composto de variável não pode haver espaços em branco. Caso deseje separar nomes compostos, deve-se utilizar o caractere de separação "_".
- ▶ Jamais uma variável pode ser definida com o mesmo nome de uma palavra que represente um dos comandos ou instruções de uma linguagem de programação de computadores.
- ▶ Não pode ser utilizado como nome de variável algum rótulo que já tenha sido usado para identificar o nome de um programa ou mesmo de outra variável. Um nome torna-se exclusivo no programa em que foi definido.

São nomes válidos de variáveis: NOMEUSUÁRIO, NOME_USUÁRIO, NUSUÁRIO, N_USUÁRIO, FONE1, FONE_1, F1, F_1, X, DELTA25, entre outras possibilidades. No entanto, definições como NOME USUÁRIO, 1X, FONE#, INTEIRO, REAL, CARACTERE, LÓGICO, entre outras, são consideradas inválidas. As palavras INTEIRO, REAL, CARACTERE, CADEIA, LÓGICO são inválidas por já estarem definidas como comandos de instrução de código da linguagem de projeto de programação "português estruturado". Além destas, as palavras apresentadas na tabela do tópico 2.3.4 também não podem ser usadas como nomes de variáveis.

O conceito de variável, do ponto de vista computacional, não estaria completo se não fosse levado em consideração o fato de que uma variável pode assumir dois papéis em um programa. Um papel de *ação*, quando seu valor inicial é modificado ao longo da execução de um programa, ou o papel de *controle*, quando seu valor é "vigado" e utilizado principalmente em operações lógicas de decisão e laços de repetição ao longo de um programa. Este capítulo mostra variáveis no papel de ação de um programa. Nos capítulos 4 e 5 as variáveis executam o papel de controle nos programas. Nesta obra uma variável tanto de ação como de controle será representada, em português estruturado, pelo comando **var**.

Do ponto de vista computacional pode-se definir, de forma simplista, que uma variável é a representação de uma região de memória utilizada para armazenar, acessar e modificar certo valor por um determinado espaço de tempo. O tempo de armazenamento de um valor em uma variável está relacionado ao tempo de duração da execução de um programa.

3.4 - O Uso de Constantes

Constante é tudo que é fixo, estável, inalterável, imutável, contínuo, incessante, invariável, de valor fixo e que é aplicado em diversos pontos de vista. Assim sendo, do ponto de vista computacional, que é semelhante ao matemático ou científico, *constante* é uma grandeza numérica fixa utilizada normalmente numa expressão aritmética ou matemática, a qual define um valor que será inalterado na expressão, independentemente das variáveis envolvidas na operação a ser realizada.

Como exemplo prático tem-se a constante matemática *pi* que equivale ao valor aproximado **3.14159265**, ou então a expressão matemática **SAÍDA = ENTRADA + 1.23**, em que o valor **1.23** é a constante da expressão e **SAÍDA** e **ENTRADA** são, respectivamente, as variáveis da referida expressão.

Do ponto de vista computacional, além de uma constante ser um valor fixo usado em uma expressão aritmética ou matemática, pode ser usada como rótulo de identificação, um nome a partir do comando

const. Por exemplo, a expressão matemática **SAÍDA = ENTRADA + PI** é, do ponto de vista computacional, formada pelas variáveis **SAÍDA** e **ENTRADA** e pela constante **PI**, desde que a constante esteja previamente definida com a instrução **const PI = 3.14159265**.

Para determinar os nomes de constantes com o comando **const**, é preciso obedecer às mesmas regras de nomes para variáveis.

A definição de constantes em um programa de computador tem por objetivo fazer uso de algum valor que, apesar de ter sido definido inicialmente ao programa, não será alterado em momento algum até o término de execução desse programa.

3.5 - Os Operadores Aritméticos

São responsáveis pelas operações matemáticas a serem realizadas em um computador. O termo *operador* é utilizado na área de programação para estabelecer as ferramentas responsáveis por executar algum tipo de ação computacional. Os *operadores aritméticos* são responsáveis pela execução do processamento matemático, exceto o operador de atribuição que pode ser usado também em ações de processamento lógico.

Os operadores aritméticos são classificados em duas categorias, sendo **binários** ou **unários**. São binários quando utilizados em operações matemáticas de radiciação, exponenciação, divisão, multiplicação, adição e subtração; são unários quando atuam na inversão do estado de um valor numérico, que pode ser passado de positivo para negativo ou vice-versa.

A tabela seguinte apresenta os operadores aritméticos a serem usados e considerados na elaboração de expressões aritméticas.

Tabela de operadores aritméticos					
Operador	Operação	Descrição	Tipo	Prioridade	Resultado
+	"n" ou "n"	Manutenção de sinal	Unário	-	Positivo
-	-n	Inversão de sinal	Unário	-	Negativo
←	x ← n	Atribuição do valor "n" a "x"	Binário	-	Positivo ou Negativo
↑	x ↑ n	Exponenciação de x^n	Unário	1	Inteiro ou Real ¹⁵
↑ (1/n)	x ↑ (1/n)	Radiciação de $\sqrt[n]{x}$	Unário	1	Real
/	x / n	Divisão de "x" por "n"	Binário	2	Real
*	x * n	Multiplicação de "x" por "n"	Binário	2	Inteiro ou Real
+	x + n	Adição de "x" com "n"	Binário	3	Inteiro ou Real
-	x - n	Subtração de "n" de "x"	Binário	3	Inteiro ou Real
div	x div n	Divisão de "x" por "n"	Binário	4	Inteiro

A tabela apresenta os operadores segundo a ordem de prioridade matemática em que as operações são realizadas, exceção à definição do operador **div** que é aritmético e de uso exclusivo para obtenção de quocientes inteiros, por esta razão possui nível de prioridade inferior a qualquer outro operador aritmético.

¹⁵ Desde que consideradas as regras matemáticas. Nesta obra os exemplos de exponenciação estarão grafados com tipo de dado inteiro.

Caso necessite alterar o nível de prioridade de um referido cálculo, ele deve ser colocado entre parênteses. Na abordagem computacional não se utilizam chaves ou colchetes, normalmente usados na área matemática.

3.6 - Expressões Aritméticas

Uma operação muito comum em programação de computadores é usar *expressões aritméticas* para o estabelecimento de processamentos matemáticos. As expressões aritméticas são realizadas a partir do relacionamento existente entre variáveis e constantes numéricas com a utilização dos operadores aritméticos.

Como exemplo de uma expressão aritmética considere a fórmula de cálculo de área de circunferência: $\text{ÁREA} = \pi \cdot \text{RAIO}^2$, em que estão presentes as variáveis **ÁREA**, **RAIO**, a constante π ($\pi = 3.14159265$), os operadores aritméticos de multiplicação e exponenciação quando se eleva o valor da variável **RAIO** ao quadrado.

As expressões aritméticas usadas na programação de computadores são escritas de uma forma um pouco diferente da conhecida na matemática. Por exemplo, a expressão $X = \{ 43 \cdot [55 : (30 + 2)] \}$ será escrita na forma computacional como $X \leftarrow (43 * (55 / (30 + 2)))$.

Perceba que as chaves e colchetes são abolidos, utilizando em seu lugar apenas os parênteses. É substituído o sinal de atribuição matemática, identificado pelo símbolo (=) "igual a", pelo símbolo (\leftarrow) "seta para a esquerda", que indica a operação de atribuição.

O símbolo de atribuição (\leftarrow) é usado para indicar que o valor de uma expressão aritmética, que é a forma computacional de descrever uma fórmula matemática, está sendo passado, transferido para a variável posicionada à esquerda do símbolo de atribuição. No caso da fórmula para o cálculo da área de uma circunferência, ela pode ser escrita como $\text{ÁREA} \leftarrow 3.14159265 * \text{RAIO} \uparrow 2$.

Se a fórmula a ser utilizada fosse calcular a área de um triângulo, em que é necessário multiplicar a base pela altura e em seguida dividir pela constante 2, como ficaria? Observe a fórmula padrão:

$$A = \frac{b \cdot h}{2}$$

Neste caso, a variável **A** representa a **área**, a variável **b** representa a **base**, a variável **h** a **altura** e o valor 2 a constante matemática da fórmula. A fórmula para o cálculo da área de um triângulo é definida como a expressão $A \leftarrow (B * H) / 2$.

Considere ainda a necessidade de converter em expressão aritmética a fórmula de Bhaskara (ou também Báskara) usada para obter as raízes reais de uma equação de segundo grau, a seguir:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

$$\Delta = b^2 - 4ac$$

A fórmula de Bhaskara deve ser convertida em sua forma equivalente em expressão aritmética. Assim sendo, ficam definidas as expressões:

$$\text{DELTA} \leftarrow B \uparrow 2 - 4 * A * C$$

$$X1 \leftarrow (-B + \text{DELTA} \uparrow (1 / 2)) / (2 * A)$$

$$X2 \leftarrow (-B - \text{DELTA} \uparrow (1 / 2)) / (2 * A)$$

As variáveis **X1** e **X2** são utilizadas para representar, respectivamente, suas equivalentes matemáticas x' e x'' . A variável **DELTA** é utilizada para representar sua equivalente matemática Δ .

Nas expressões de cálculo das variáveis **X1** e **X2** a definição de **DELTA** $\uparrow (1/2)$ representa a extração da raiz quadrada de **DELTA**. Uma raiz, seja ela de qual índice for, pode ser extraída a partir do cálculo da exponenciação de sua base em relação ao valor inverso de seu índice.

Uma das funções de um programador de computador é saber converter em expressões aritméticas computáveis as fórmulas matemáticas que são utilizadas pelos usuários dos programas.

3.7 - Instruções e Comandos

O controle operacional de um computador eletrônico é realizado por um conjunto de programas. Um programa, além da lógica de programação necessária e dos dados a serem manipulados, necessita de um conjunto de instruções para comandar as ações programadas de um computador.

Uma instrução a ser executada em um computador pode ter um ou mais comandos. Comando (palavra-chave) é um componente que pertence ao dicionário de uma linguagem de programação. Assim sendo, uma linguagem de programação formal (FORTRAN, COBOL, BASIC, PASCAL, Lua, C, C++, ASSEMBLY, entre outras) ou informal (LPP) é um conjunto de comandos que, utilizados isoladamente ou em conjunto, determinam as instruções a serem dadas por um programa ao computador.

O número de linguagens de programação existentes é bastante grande, o que pode ocasionar ao estudante certa dificuldade para adaptar-se aos diversos "idiomas" computacionais. O objetivo desta obra é ajudar a desenvolver a lógica de programação independentemente da linguagem formal de computação a ser utilizada. Existem várias formas de escrever programas para computadores, mas só existe um modo de programá-los, e é utilizando boa lógica.

Considere a seguinte descrição: *via pública para circulação urbana, total ou parcialmente ladeada de casas*. Em português chama-se *rua*, em inglês *street*, em castelhano *calle*. Observe que não importa a forma usada para descrever uma via pública para circulação urbana, total ou parcialmente ladeada de casas. A forma escrita em cada idioma não mudou em nenhum momento o significado das palavras *rua*, *street* e *calle*. Algo semelhante ocorre com as linguagens de programação de computadores. O que importa é saber o quê e como fazer num programa, saber usar a lógica de programação, ou seja, o conceito de programação, não importa em qual linguagem o programa será escrito.

Este livro adota o uso de códigos escritos em *português estruturado*, uma linguagem de programação informal de documentação, como já descrito no tópico 2.3.4.

Deste ponto em diante o leitor terá contato com os comandos da linguagem de projeto de programação em português estruturado: ATÉ, ATÉ_QUE, ATÉ_SEJA, CADEIA, CARACTERE, CASO, CLASSE, CONJUNTO, CONST, CONTINUA, DE, EFETUE, ENQUANTO, ENQUANTO_SEJA, ENTÃO, ESCREVA, FAÇA, FIM, FIM_ATÉ_SEJA, FIM_CASO, FIM_CLASSE, FIM_ENQUANTO, FIM_FAÇA, FIM_LAÇO, FIM_PARA, FIM_REGISTRO, FIM_SE, FUNÇÃO, HERANÇA, INÍCIO, INTEIRO, LAÇO, LEIA, LÓGICO, PARA, PASSO, PRIVADA, PROCEDIMENTO, PROGRAMA, PROTEGIDA, PÚBLICA, REAL, REGISTRO, REPITA, SAIA_CASO, SE, SEÇÃO_PRIVADA, SEÇÃO_PROTEGIDA, SEÇÃO_PÚBLICA, SEJA, SENÃO, TIPO, VAR e VIRTUAL.

Lembre-se de que os nomes de variáveis e de constantes, quando definidos em programas codificados (em qualquer linguagem de programação formal ou informal), não podem, em hipótese nenhuma, ser iguais às palavras-chave citadas no parágrafo anterior.

Anteriormente foram estabelecidos o significado de variável e de constante e algumas regras para seu uso. Agora é necessário conhecer mais algumas regras que serão usadas daqui em diante para representar a descrição de comandos e instruções, a saber:

- ▶ Toda referência feita a um comando em português estruturado nesta obra será grafada com caracteres minúsculos e em negrito.
- ▶ Em hipótese nenhuma os comandos do código português estruturado estarão escritos dentro dos símbolos dos diagramas de blocos.
- ▶ A codificação de um programa escrita à mão é sempre realizada com caracteres maiúsculos e em letra de forma, o que obriga sempre a cortar o valor zero com uma barra para diferenciá-lo da letra "O".
- ▶ Não se escrevem programas de computador à mão com letra cursiva.
- ▶ Toda referência feita a uma variável ou constante, quando definida nesta obra, será grafada com caracteres maiúsculos em itálico.
- ▶ Os nomes das variáveis serão sempre indicados e utilizados dentro dos símbolos dos diagramas de blocos.
- ▶ Todo valor atribuído diretamente no código do programa a uma variável será realizado por meio do símbolo de atribuição (\leftarrow), seta para a esquerda, tanto nos diagramas de blocos como nos códigos escritos em português estruturado.
- ▶ Todo valor atribuído diretamente no código do programa a uma constante será realizado por meio do símbolo de atribuição ($=$), igual a, tanto nos diagramas de blocos como nos códigos escritos em português estruturado.

Para operações de multiplicação, é preciso ter o cuidado de usar o símbolo asterisco (*) no código escrito em português estruturado. Quando se tratar de representar essa operação em um diagrama de bloco, deve-se usar o símbolo (\otimes).

3.8 - Exercício de Aprendizagem

Como descrito no tópico 3.1, um computador eletrônico executa as ações de entrada de dados, processamento de dados e a saída de dados. Assim sendo, qualquer programa deve, de alguma forma, executar essas ações e a partir delas o programador planejar seu trabalho. O processo de entrada e de saída de dados é representado em código português estruturado, respectivamente, com os comandos **leia** (entrada) e **escreva** (saída). As operações de processamento matemático ou lógico, nesta etapa, serão representadas com o uso do operador de atribuição (\leftarrow).

Para representar as operações de entrada (**leia**), processamento (\leftarrow) e saída (**escreva**) serão utilizados, respectivamente, os símbolos *manual input*, *process* e *display* indicados na tabela do tópico 2.3.2. Além desses símbolos será utilizado o símbolo *terminal* para indicar o início e o fim do programa representado no diagrama de bloco.

1º Exemplo

Para aprender os detalhes apresentados até o momento, considere o seguinte exemplo de um problema a ser transformado em programa de computador:

Desenvolver um programa de computador que efetue a leitura de dois valores numéricos inteiros. Processe a operação de adição dos dois valores e apresente na sequência a soma obtida com a operação.

Note que o programador sempre estará diante de um problema, o qual deve ser resolvido primeiramente por ele para depois ser implementado em um computador e então utilizado por um usuário.

Para cumprir estas etapas, é necessário e fundamental que o programador primeiramente entenda o problema, para depois buscar e implementar a sua solução em um computador, ou seja, o programador é o

profissional que "ensina" o computador a realizar uma determinada tarefa controlada por meio de um programa. Desta forma, o segredo de aplicação de uma boa lógica de programação está na devida compreensão do problema a ser solucionado.

Em relação ao problema proposto, é necessário interpretá-lo adequadamente (entendê-lo) com o auxílio de uma ferramenta lógica denominada algoritmo, que deve estabelecer os passos necessários a serem cumpridos na busca da solução do problema. Lembre-se de que um algoritmo é semelhante a uma "receita culinária".

A solução de um algoritmo deve ocorrer com base nas etapas de ação de um computador (entrada, processamento e saída). Neste sentido, o programador deve executar três atividades na busca da solução de um problema computacional: primeiramente faz o *entendimento* (descrição dos passos), depois faz a *diagramação* (diagrama de bloco) e a *codificação* (português estruturado), como mostra a Figura 3.2.

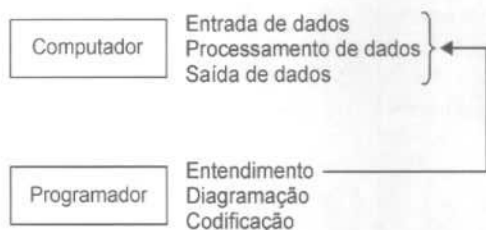


Figura 3.2 - Forma esquemática de comportamento do programador diante de um problema.

Das atividades a serem cumpridas para a solução do problema, a atividade de entendimento tem característica mental, ou seja, não é necessário colocá-la no papel, a menos que se queira. No entanto, a diagramação e a codificação devem ser consideradas obrigatórias. A diagramação por ser a contextualização gráfica da linha de entendimento de quem escreveu o programa de computador e a codificação em uma linguagem de programação (não necessariamente o código em português estruturado) por ser o produto final.

Entendimento (análise passo a passo do que deve ser feito pelo programa)

Observe a seguir a descrição dos passos para que ocorra a entrada dos dois valores inteiros (que são desconhecidos pelo programador e serão representados pelas variáveis A e B) e sua respectiva operação de adição (consequência da soma dos valores informados com as variáveis A e B) que será atribuída à variável X.

1. Ler dois valores desconhecidos, representados pelas variáveis A e B.
2. Efetuar a adição das variáveis A e B, cujo resultado será atribuído à variável X.
3. Apresentar o valor da variável X, que é o resultado da soma realizada.

Os três passos anteriores estabelecem a sequência a ser executada por um algoritmo (gráfico e textual). Descrevem de forma sucinta a ação passo a passo para a solução do problema indicado, baseando-se nas etapas de trabalho de um computador: entrada, processamento e saída.

A seguir, o leitor terá contato com o algoritmo na forma gráfica (*diagrama de bloco*) e na forma textual (*português estruturado*) para representar a solução do problema proposto.

Diagramação (confeção do diagrama de bloco de acordo com o entendimento)

Completada a fase de interpretação do problema, em que se toma consciência do que é necessário fazer, passa-se para a fase de diagramação do algoritmo, que será feita de acordo com os detalhes definidos na norma ISO 5807:1985(E), como mostra a Figura 3.3.

Observe a indicação dos rótulos **início** e **fim** no diagrama de bloco com o uso do símbolo *terminal*, que deve estar sempre presente, indicando o ponto de início e fim do diagrama. Note também a existência das linhas com setas ligando os símbolos entre si. Isso é necessário, pois desta forma sabe-se a direção que o fluxo de ações de um programa deve seguir. Veja também o uso dos símbolos *manual input* (equivalente ao comando **leia**) com a definição das variáveis de entrada **A** e **B**, *process* (equivalente à ação de processamento) com a definição do processamento matemático de adição $X \leftarrow A + B$ e *display* (equivalente ao comando **escreva**) com a definição da variável de saída **X**.

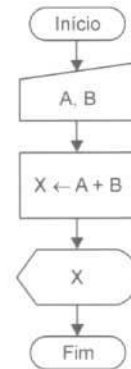


Figura 3.3 - Diagrama de bloco do programa de adição de dois valores inteiros.

Codificação (escrita do programa de forma textual)

Após estabelecer os passos anteriores (entendimento e diagramação), passa-se à fase de codificação do programa, que obedece ao que está definido no diagrama de bloco, pois ele é a representação gráfica (concreta) da linha de raciocínio lógico do programador a ser colocado em um programa que controla as ações de um computador. Nessa etapa do trabalho deve haver a preocupação do programador em definir o tipo de dado de cada variável. Essa atitude é normalmente característica das linguagens formais estruturadas de programação para reservar na memória o espaço adequado à manipulação de cada tipo de dado, segundo o valor a ser utilizado. Serão usadas três variáveis: A, B e X, as quais devem ser relacionadas e definidas antes do seu uso. Observe a seguir o preâmbulo de um programa típico escrito em português estruturado.

```

programa SOMA_NÚMEROS
var
  X : inteiro
  A : inteiro
  B : inteiro
  
```

Um programa codificado em português estruturado é iniciado com uma instrução de identificação codificada a partir do comando **programa** seguida de um nome de identificação. O nome do programa **SOMA_NÚMEROS** é formado por duas palavras separadas com o símbolo *underline*. As regras para definição de nome de programa são as mesmas usadas para dar nomes às variáveis e constantes. Lembre-se de que não podem existir nomes de programa, variável ou constante iguais.

Após o nome de identificação do programa é estabelecida a instrução que fará a definição em memória primária dos espaços para a manipulação das variáveis, segundo seus respectivos tipos de dados. Essa instrução é conseguida com o comando **var** seguido dos nomes de identificação das variáveis a serem usadas e seus respectivos tipos de dados. A definição, ou seja, a instância das variáveis, é feita abaixo do comando **var**, em média com deslocamento de duas posições à frente.

Esse deslocamento de espaços nos nomes das variáveis em relação ao comando **var** é chamado de *indentação*¹⁶.

¹⁶ O termo "indentação" (forma aportuguesada) origina-se do termo em inglês "indent ou indented" que possui como significado os termos entalhe, denteação, parágrafo, dentear, cortar, recortar, recuar ou contratar. Quando aplicado à programação de computadores, o termo significa recuo de parágrafo à direita na medida mínima de dois caracteres, considerando o uso de caracteres da categoria *monoespaçado*, como as fontes *Courier* ou *Lucida*, entre outras em que todos os caracteres possuem a mesma medida de largura.

Após relacionar as variáveis a serem usadas no programa com o comando **var** e estabelecer a instrução de criação das variáveis na memória principal, passa-se para a fase de montagem do que está estabelecido no diagrama de bloco entre os símbolos *terminal* (indicações de início e fim do programa).

A parte indicada entre o trecho sinalizado com o símbolo *terminal* como início e fim chama-se *bloco*, daí o nome diagrama de bloco. As ações do bloco devem ser traduzidas para a linguagem de codificação, neste caso português estruturado.

```
início
  leia A
  leia B
  X ← A + B
  escreva X
fim
```

O bloco de instruções do programa (sinalizado entre os comandos **início** e **fim**) indica as ações de entrada (instrução **leia A**), de processamento (instrução **X ← A + B**) e de saída (instrução **escreva X**) a ser realizada pelo programa. O código escrito entre os comandos **início** e **fim** é apresentado com um deslocamento (*indentação*) de duas posições à direita.

O uso de *indentação* é um estilo de escrita que deve sempre ser respeitado, pois visa facilitar a leitura do código dos vários blocos de ação que um programa de computador possa ter. É muito comum um programa possuir milhares de linhas e nestes casos o uso da *indentação* é um instrumento de grande valia e organização.

Após a leitura (entrada) dos valores para as variáveis A e B por meio do comando **leia**, o programa faz o processamento da adição, atribuindo à variável X o resultado da soma obtida, e com o comando **escreva** (saída) faz a apresentação do resultado armazenado na variável X. A seguir é exibido o programa completo.

```
programa SOMA_NÚMEROS
var
  X : inteiro
  A : inteiro
  B : inteiro
início
  leia A
  leia B
  X ← A + B
  escreva X
fim
```

A título de fixação de aprendizagem, são apresentados mais três exemplos de programas que aplicam os conceitos de entrada, processamento e saída estudados anteriormente. Observe atentamente cada exemplo de aprendizagem e procure perceber os detalhes existentes, pois são importantes para resolver os exercícios de fixação do final deste capítulo.

2º Exemplo

Elaborar um programa de computador que calcule a área de uma circunferência e apresente a medida da área calculada.

Entendimento

Para fazer o cálculo da área de uma circunferência, é necessário conhecer primeiramente a fórmula que executa o cálculo, sendo $A = \pi R^2$, em que **A** é a variável que conterà o resultado do cálculo da área, π é o valor da constante pi (3.14159265) e **R** o valor da variável que representa o raio. Basta estabelecer o seguinte:

1. Ler um valor para o raio, no caso variável R.
2. Estabelecer que pi venha a possuir o valor 3.14159265.
3. Efetuar o cálculo da área, elevando ao quadrado o valor de R e multiplicando esse valor por pi.
4. Apresentar o valor da variável A.

A fórmula para o cálculo da área pode ser escrita na forma de expressão aritmética como $A \leftarrow 3.14159265 * R \uparrow 2$. No entanto, o exemplo faz menção à definição da constante pi associada ao seu valor.

Quando for mencionada uma operação de multiplicação num diagrama de bloco, deve ser utilizado o símbolo \otimes , como na Figura 3.4.

Observe o uso do primeiro símbolo *process* em que é definido o valor da constante pi. Note o uso do símbolo = (igual a) para estabelecer o valor 3.14159265 para a constante pi a ser utilizada. Perceba a diferença no uso de uma constante em relação a um processamento. O processamento usa o símbolo de atribuição (\leftarrow), enquanto uma constante o símbolo "igual a".

Diagramação

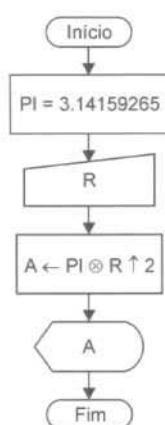


Figura 3.4 - Diagrama de bloco do programa de cálculo de área de circunferência.

Codificação

```

programa AREA_CIRCULO
const
  PI = 3.14159265
var
  A : real
  R : real
inicio
  leia R
  A ← PI * R ↑ 2
  escreva A
fim
  
```

Como regra geral de trabalho e de organização, constantes com o comando **const** ficam sempre à frente de variáveis com o comando **var**.

3º Exemplo

Desenvolver um programa que calcule o salário líquido de um professor. Para elaborar o programa, é necessário possuir alguns dados, tais como valor da hora aula, número de horas trabalhadas no mês e percentual de desconto do INSS. Em primeiro lugar, deve-se estabelecer o seu salário bruto para fazer o desconto e ter o valor do salário líquido.

Observe a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 3.5.

Entendimento

1. Estabelecer a leitura da variável HT (horas trabalhadas no mês).
2. Estabelecer a leitura da variável VH (valor hora aula).
3. Estabelecer a leitura da variável PD (percentual de desconto).
4. Calcular o salário bruto (SB), sendo a multiplicação das variáveis HT e VH.
5. Calcular o total de desconto (TD) com base no valor de PD dividido por 100.

6. Calcular o salário líquido (SL), deduzindo o desconto do salário bruto.
7. Apresentar os valores dos salários bruto e líquido: SB e SL.

Diagramação

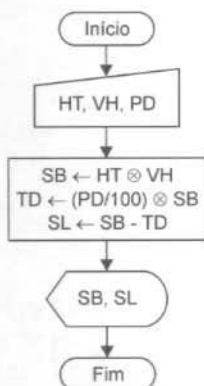


Figura 3.5 - Diagrama de bloco do programa de cálculo de salário.

Codificação

```

programa SALARIO_PROFESSOR
var
  HT : inteiro
  VH, PD, TD, SB, SL : real
inicio
  leia HT, VH, PD
  SB ← HT * VH
  TD ← (PD/100) * SB
  SL ← SB - TD
  escreva SB, SL
fim
  
```

4º Exemplo

Desenvolver um programa que faça a entrada do nome de uma pessoa e de seu sexo, em seguida deve apresentar os dados anteriormente informados.

Considere para a solução deste problema que a entrada do nome será realizada na variável **NOME** e a entrada do sexo na variável **SEXO**. Considere ainda que a variável **NOME** terá seu tipo de dado definido com o comando **cadeia** (por ser um conjunto com mais de um caractere) e que a variável **SEXO** terá seu tipo de dado definido com o comando **caractere**, uma vez que a entrada do sexo será indicada apenas por uma letra: *M* para masculino ou *F* para feminino.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 3.6.

Entendimento

1. Efetuar a entrada do nome na variável NOME.
2. Efetuar a entrada do sexo na variável SEXO.
3. Apresentar o nome e o sexo informados.

Observe na Figura 3.6 o fato de o programa não ter um processamento matemático. Após a entrada do nome e do sexo, realiza-se a saída desses dados. Apesar de não existir explicitamente um processamento, não significa que o computador não fará um para conseguir atender o que está sendo pedido.

Diagramação

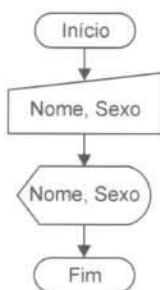


Figura 3.6 - Diagrama de bloco do programa de entrada e saída do nome e do sexo.

Codificação

```

programa NOME_SEXO
var
  NOME : cadeia
  SEXO : caractere
inicio
  leia NOME, SEXO
  escreva NOME, SEXO
fim
  
```

Faz-se necessário esclarecer uma prática computacional muito usada ao longo dos anos e pouco conhecida de fato. O código de um programa escrito à mão deve sempre ser feito em caracteres de fôrma (letra bastão) grafados em maiúsculo. Por causa desse hábito, há a necessidade de sempre cortar o valor zero com uma barra no sentido de diferenciá-lo da letra "O" para não confundir-nos nem gerar erros na escrita da sintaxe de uma linguagem computacional, pois tanto o número zero como a letra "O" de fôrma maiúscula escritos por uma pessoa têm o mesmo desenho. Assim sendo, o número zero será cortado somente nessa circunstância e em nenhum outro momento, pois fora da área de programação o zero cortado significa conjunto vazio.

3.9 - Exercícios de Fixação

1. Escreva ao lado de cada valor o tipo em que se enquadra (inteiro, real, caractere, cadeia ou lógico), levando em consideração que um valor numérico pertencente ao conjunto de valores numéricos inteiros está contido também no conjunto de valores numéricos reais.

-456	_____	0	_____
.F.	_____	1.56	_____
.Falso.	_____	-1.56	_____
.V.	_____	34	_____
"0.87"	_____	45.8976	_____
"0"	_____	-465	_____
"-9.12"	_____	678	_____
"-900"	_____	-678	_____
"Casa 8"	_____	-99.8	_____
"Cinco"	_____	.V.	_____
"V"	_____	1000	_____

2. Assinale com um X os nomes válidos para uma variável.

() ENDEREÇO	() END*A-6
() 21BRASIL	() CIDADE3
() FONE\$COM	() #CABEC
() NAMEUSER	() REAL

- l) Elaborar um programa que apresente o valor da conversão em dólar (US\$) de um valor lido em real (R\$). O programa deve solicitar o valor da cotação do dólar e também a quantidade de reais disponível com o usuário.
- m) Construir um programa que leia três valores numéricos inteiros (representados pelas variáveis A, B e C) e apresente como resultado final o valor da soma dos quadrados dos três valores lidos.
- n) Construir um programa que leia três valores numéricos inteiros (representados pelas variáveis A, B e C) e apresente como resultado final o valor do quadrado da soma dos três valores lidos.
- o) Elaborar um programa que leia quatro valores numéricos inteiros (variáveis A, B, C e D). Ao final o programa deve apresentar o resultado do produto (variável P) do primeiro com o terceiro valor, e o resultado da soma (variável S) do segundo com o quarto valor.
- p) Elaborar um programa que leia o valor numérico correspondente ao salário mensal (variável SM) de um trabalhador e também faça a leitura do valor do percentual de reajuste (variável PR) a ser atribuído. Apresentar o valor do novo salário (variável NS).
- q) Elaborar um programa que calcule e apresente o valor do resultado da área de uma circunferência (variável A). O programa deve solicitar a entrada do valor do raio da circunferência (variável R). Para a execução deste problema utilize a fórmula $A \leftarrow 3.14159265 * R \uparrow 2$.
- r) Em uma eleição sindical concorreram ao cargo de presidente três candidatos (representados pelas variáveis A, B e C). Durante a apuração dos votos foram computados votos nulos e em branco, além dos votos válidos para cada candidato. Deve ser criado um programa de computador que faça a leitura da quantidade de votos válidos para cada candidato, além ler também a quantidade de votos nulos e em branco. Ao final o programa deve apresentar o número total de eleitores, considerando votos válidos, nulos e em branco; o percentual correspondente de votos válidos em relação à quantidade de eleitores; o percentual correspondente de votos válidos do candidato A em relação à quantidade de eleitores; o percentual correspondente de votos válidos do candidato B em relação à quantidade de eleitores; o percentual correspondente de votos válidos do candidato C em relação à quantidade de eleitores; o percentual correspondente de votos nulos em relação à quantidade de eleitores; e por último o percentual correspondente de votos em branco em relação à quantidade de eleitores.
- s) Elaborar um programa que leia dois valores numéricos reais desconhecidos representados pelas variáveis A e B. Calcular e apresentar os resultados das quatro operações aritméticas básicas.
- t) Construir um programa que calcule e apresente em metros por segundo o valor da velocidade de um projétil que percorre uma distância em quilômetros a um espaço de tempo em minutos. Utilize a fórmula $VELOCIDADE \leftarrow (DIST\grave{A}NCIA * 1000) / (TEMPO * 60)$.
- u) Elaborar um programa de computador que calcule e apresente o valor do volume de uma esfera. Utilize a fórmula $VOLUME \leftarrow (4 / 3) * 3.14159 * (RAIO \uparrow 3)$.
- v) Elaborar um programa que leia dois valores numéricos inteiros, os quais devem representar a base e o expoente de uma potência, calcule a potência e apresente o resultado obtido.
- w) Elaborar um programa que leia uma medida em pés e apresente o seu valor convertido em metros, lembrando que um pé mede 0,3048 centímetros.
- x) Elaborar um programa que calcule uma raiz de base qualquer com índice qualquer.
- y) Construir um programa que leia um valor numérico inteiro e apresente como resultado os seus valores sucessor e antecessor.



Programação com Decisão

Anteriormente foram estudadas as etapas de entradas, processamentos e saídas de dados com o uso de algumas ferramentas básicas, tais como variáveis, constantes, operadores aritméticos e, principalmente, expressões aritméticas. Apesar de já ser possível, com essas ferramentas, solucionar alguns problemas simples e assim transformá-los em programas, os recursos até então estudados são muito limitados por permitirem apenas soluções sequenciais simples. Em certas ocasiões um determinado valor deve ser tratado de forma a realizar um desvio no processamento executado no computador, usando o princípio de tomada de decisões.

A fim de orientar o estudo do processo de tomada de decisão, este capítulo apresenta operadores relacionais e lógicos, além do uso de decisões simples, compostas, sequenciais, encadeadas e de seleção. O fato de um computador tomar decisões não o torna uma máquina "inteligente", apenas garante o controle, de forma lógica, da máquina em si. Descreve também noções básicas e simples de divisibilidade entre números.

4.1 - Ser Programador

Antes de prosseguir o estudo, cabe apresentar as três virtudes de um programador de computador, sabiamente indicadas pelo professor Guerreiro da Universidade Nova de Lisboa. Segundo Guerreiro, o programador de computador deve possuir as virtudes: disciplina, humildade e perseverança (GUERREIRO, 2000, p. 2-3), descritas resumidamente em seguida.

- ▶ Disciplina, pois ao programar sem nenhuma metodologia, corre-se o risco de ficar soterrado numa avalanche de conceitos conflitantes. Disciplina é qualidade essencial de um programador, seja em que linguagem de programação for. Um programador indisciplinado, mesmo genial, é de pouca utilidade para a equipe de desenvolvimento de *software* da qual faz parte.
- ▶ Humildade, porque o programador é frequentemente confrontado com suas próprias limitações. Mesmo usando adequadamente algum método (disciplina), muitas vezes se cometem erros, não por ter entendido mal o problema (sem dúvida este é o erro mais grave), mas por pensar que já se conhecem a linguagem de programação em uso e seu método de aplicação. O excesso de confiança é mau conselheiro na tarefa de programação. Por outro lado, errar, reconhecer o erro, corrigi-lo faz bem à alma. É fundamental aceitar as próprias limitações e nunca partir do pressuposto de que é fácil programar.

- ▶ Perseverança, pois é necessário deixar o trabalho bem feito. É preciso ultrapassar muitos erros, rever e tomar decisões que se pensava estarem resolvidas. É preciso atender pormenores que inicialmente passaram despercebidos e afinar pontos não resolvidos. É necessário deixar o programa em condições de outro concluir. Frequentemente isso tudo tem de ser conseguido com prazos estipulados. É normal que ocorram momentos de desânimo e se queira deixar as coisas como estão.

Dentro do exposto, cabe deixar claro que se no estudo desta obra não forem assumidas essas três virtudes, é melhor não dar continuidade. É mais saudável parar neste ponto e repensar seus reais interesses, pois sem dúvida será um verdadeiro martírio. Mas se houver concordância com essas questões e seguir as recomendações indicadas até este ponto, tenha certeza de que se sentirá muito bem ao final do estudo, pois estará no time dos que são programadores de computador, pelo menos num nível de conhecimento que permite assumir responsabilidades como *trainee* ou *júnior*.

4.2 - Decisões, Condições e Operadores Relacionais

O foco de estudo deste capítulo é entender a capacidade de computadores realizarem tomada de decisões por meio de processamento lógico. A tomada de decisão realizada pelo computador estabelece uma ação de desvio na operação do fluxo do programa. Desta forma, um determinado trecho do programa pode realizar uma ou outra tarefa de processamento.

Para entender o tema deste capítulo, é importante entender separadamente *condição* e *decisão*. Assim sendo, *condição* pode ser entendida como uma obrigação que se impõe e se aceita, enquanto *decisão* pode ser o ato ou efeito de decidir, ou seja, de optar, de tomar uma decisão. Nota-se que o ato de tomar uma decisão está calcado no fato de haver uma condição. A condição codificada em português estruturado deve estar entre parênteses.

A representação gráfica da ideia de tomada de decisão é feita com os símbolos *decision* e *connector* indicados na tabela do tópico 2.3.2. Esses símbolos são utilizados para representar blocos adjacentes de instruções subordinadas à condição definida. De forma geral, usa-se um bloco adjacente quando se trabalha com a tomada de decisão simples ou dois blocos adjacentes com a tomada de decisão composta. Com uso dos símbolos *decision* e *connector* tem-se um diagrama de blocos e não mais um diagrama de bloco.

Do ponto de vista computacional uma condição é uma expressão booleana cujo resultado é um valor lógico *falso* ou *verdadeiro*. Desta forma, uma expressão booleana como condição é conseguida com uma relação lógica entre dois elementos e um operador relacional.

Os elementos relacionados em uma expressão lógica (condição) são representados por relações binárias entre variáveis e constantes. São possíveis as relações de *variáveis versus variáveis* e de *variáveis versus constantes*.

O estabelecimento de uma condição, ou seja, de uma relação lógica entre dois elementos é feito a partir de operadores relacionais, que se encontram definidos na tabela seguinte.

Tabela de operadores relacionais	
Operador	Descrição
=	Igual a
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
<>	Diferente de

Com o uso de operadores relacionais têm-se como condições válidas numa lógica de tomada de decisão do tipo *variável versus variável* as relações, por exemplo, entre as variáveis **A** e **B**: $A = B$, $A > B$, $A < B$, $A >= B$, $A <= B$ e $A <> B$.

Usando operadores relacionais têm-se como condições válidas numa lógica de tomada de decisão do tipo *variável versus constante* (ou *constante versus variável*) as relações, por exemplo, entre a variável **A** e a constante de valor numérico inteiro **5**: $A = 5$, $A > 5$, $A < 5$, $A >= 5$, $A <= 5$ e $A <> 5$.

É pertinente considerar uma observação importante em relação à utilização do operador "igual a" identificado pelo símbolo $=$. Em algumas linguagens de programação de computadores (como ocorre, por exemplo, na linguagem BASIC) esse símbolo é utilizado para representar duas ações: atribuição quando usado em operações matemáticas e operação relacional quando usado em operações lógicas, o que gera, em alguns casos, certa confusão. Por esta razão, nesta obra o símbolo $=$ é exclusivo para representar operações lógicas de igualdade, enquanto as ações de atribuição para operações matemáticas são representadas pelo símbolo \leftarrow . Esta diferenciação entre as simbologias é encontrada nas linguagens PASCAL, C, C++, Lua, entre outras.

Os operadores relacionais possuem o mesmo nível de precedência entre si. Assim não há necessidade de preocupar-se em alterar o nível de prioridade entre eles.

4.3 - Desvio Condicional Simples

A tomada de decisão simples (desvio condicional simples) do ponto de vista do diagrama de blocos é representada pelos símbolos *decision* e *connector*. A partir do símbolo *decision* é estabelecido o foco do desvio do fluxo de um programa. Esse desvio é processado apenas para o lado que indicar o resultado da condição como verdadeira, não importando se essa ação estará sinalizada do lado esquerdo ou direito do símbolo *decision*. Por esta razão é importante sinalizar as duas linhas de fluxo que saem do símbolo *decision* com os rótulos **S** e **N** indicando, respectivamente, os lados **sim** e **não** da condição estabelecida, deixando bem claro o lado da ação considerada para a condição verdadeira.

Observe na Figura 4.1 a imagem básica de um diagrama de blocos com os rótulos **S** e **N** para a execução de uma tomada de decisão simples com base na **CONDIÇÃO** definida no símbolo *decision*. Note o uso das linhas de fluxo com as setas indicando a direção do fluxo de processamento do programa.

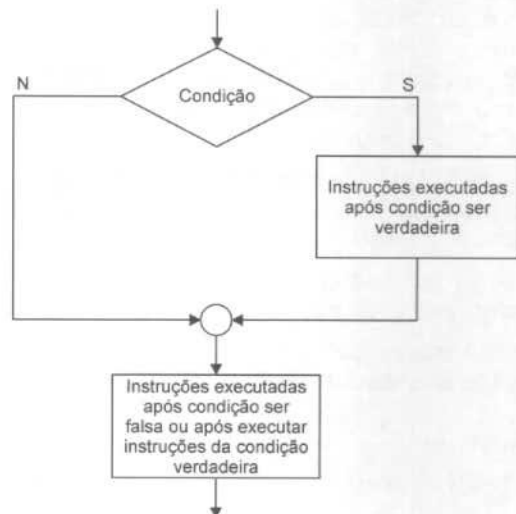


Figura 4.1 - Estrutura de tomada de decisão simples.

No lado sinalizado com o rótulo **S** está a execução das instruções subordinadas desse bloco caso a condição estabelecida seja verdadeira, para depois direcionar o fluxo do programa para o símbolo *connector*. O lado sinalizado com o rótulo **N** direciona o fluxo do programa diretamente para o símbolo *connector*, indicando que nenhuma ação será efetuada caso a condição não seja verdadeira. A principal característica de uma tomada de decisão simples é o fato de existir um bloco de operações somente se a condição for verdadeira.

Após a tomada de decisão simples sobre uma determinada condição, independentemente de o resultado ser falso ou verdadeiro, executam-se as eventuais instruções estabelecidas após o símbolo *connector*.

A tomada de decisão simples do ponto de vista da codificação em português estruturado utiliza os comandos **se**, **então** e **fim_se** na construção da instrução **se...então/fim_se**. Nessa instrução, se a condição (definida entre os comandos **se** e **então**) for verdadeira, serão executadas todas as instruções subordinadas e definidas dentro do bloco adjacente entre os comandos **se...então** e **fim_se**. Após a execução ocorre automaticamente a execução das eventuais instruções existentes após o comando **fim_se**. Se a condição for falsa, serão executadas apenas as eventuais instruções que estiverem após o comando **fim_se**. Observe a estrutura sintática seguinte:

```

se (<condição>) então
    [instruções executadas após condição ser verdadeira]
fim_se
    [instruções executadas após condição ser falsa ou após executar instruções da]
    [condição verdadeira]
  
```

Atente para um detalhe muito importante no tocante à codificação do trecho de programa em português estruturado. A definição de um bloco adjacente de instruções subordinadas é evidente em um diagrama de blocos, mas o mesmo não ocorre no código em português estruturado. Isso obriga o programador, mais disciplinado e elegante, a usar o processo de *indentação*.

Entre os comandos **se...então** e **fim_se** ocorre a indicação das instruções subordinadas ao bloco adjacente com deslocamento (mínimo) de duas posições para a direita. Essa atitude na escrita do código deixa claro para o próprio programador ou para a pessoa que for dar continuidade ao programa (quando o trabalho é efetuado em equipes de desenvolvimento) qual de fato é o bloco subordinado a uma determinada condição e quais são as instruções subordinadas. A indicação **<condição>** entre parênteses deve ser substituída pela expressão lógica da condição a ser utilizada. Os trechos sinalizados entre colchetes são instruções a serem executadas pelo programa.

A título de ilustração da tomada de decisão simples em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Elaborar um programa de computador que leia dois valores numéricos reais desconhecidos. Em seguida o programa deve efetuar a adição dos dois valores lidos e apresentar o resultado caso seja maior que 10.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.2.

Entendimento

1. Definir a entrada de dois valores incógnitos (variáveis A e B).
2. Efetuar a adição dos valores A e B e atribuir o resultado da adição à variável X.
3. Apresentar o resultado da soma armazenada na variável X, caso a variável X tenha seu valor maior que 10.

Diagramação

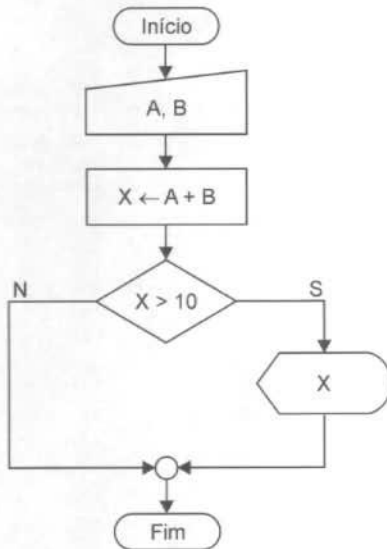


Figura 4.2 - Exemplo da utilização da instrução `se...então/fim_se`.

Codificação

```

programa ADIÇÃO_DE_NÚMEROS_1
var
  A, B, X : real
início
  leia A, B
  X ← A + B
  se (X > 10) então
    escreva X
  fim_se
fim
  
```

Após os tipos de variáveis, é solicitada a leitura dos valores para as variáveis **A** e **B**, depois esses valores são somados e o resultado é atribuído à variável **X**, a qual possui o resultado da adição dos dois valores lidos. Neste ponto, a condição **X > 10** é avaliada e, sendo verdadeira, apresenta-se o resultado da soma. Caso o resultado da variável **X** não seja maior que 10, o programa é encerrado sem apresentar o resultado.

4.4 - Desvio Condicional Composto

A tomada de decisão composta (desvio condicional composto) do ponto de vista do diagrama de blocos é representada também com os símbolos *decision* e *connector*, como ocorreu com a representação da tomada de decisão simples. A tomada de decisão composta desvia o fluxo de programa tanto para o lado indicado verdadeiro como para o lado indicado falso, não importando se está em uso o lado esquerdo ou direito do diagrama, obrigando a manter a sinalização dos lados com os rótulos **S** e **N** para indicar os lados **sim** e **não** da condição estabelecida.

A Figura 4.3 apresenta um diagrama de blocos com os rótulos **S** e **N** para uma tomada de decisão composta. Note o uso das linhas de fluxo com as setas indicando a direção do fluxo de processamento do programa a partir da **CONDIÇÃO**, a qual pode ter seu resultado lógico falso ou verdadeiro. Se o resultado lógico da condição for verdadeiro, executa-se o grupo de instruções subordinadas à linha de fluxo sinalizada pelo rótulo **S**. Após as instruções subordinadas, o fluxo de programa é direcionado para o símbolo *connector*. Se o resultado lógico da condição for falso, ocorre a execução do grupo de instruções subordinadas à linha de fluxo sinalizada pelo rótulo **N**. A principal característica de uma tomada de decisão composta é o fato de existir um bloco de operações para cada um dos lados da condição.

Após a tomada de decisão composta sobre a condição, independentemente de o resultado ser falso ou verdadeiro, executam-se as eventuais instruções estabelecidas após o símbolo *connector*.

A tomada de decisão composta do ponto de vista da codificação em português estruturado utiliza os comandos **se**, **então**, **senão** e **fim_se** na construção da instrução **se...então/senão/ fim_se**.

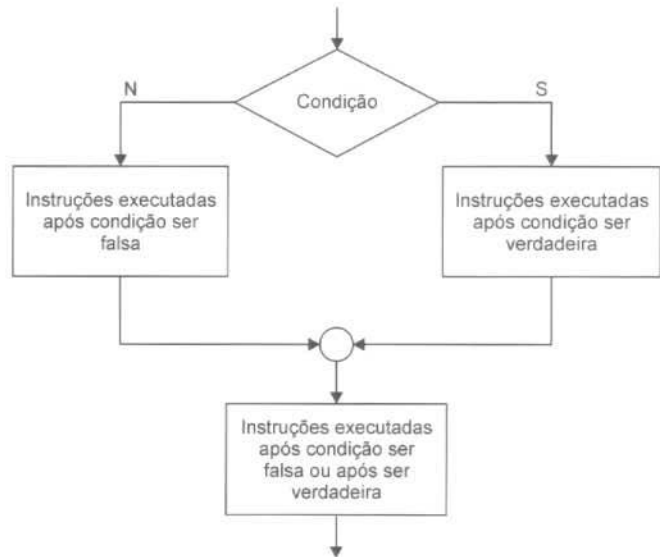


Figura 4.3 - Estrutura de tomada de decisão composta.

Nessa instrução, se a condição (definida entre os comandos **se** e **então**) for verdadeira, são executadas todas as instruções subordinadas do bloco adjacente entre os comandos **se...então** e **senão**. Caso seja a condição falsa, são executadas todas as instruções subordinadas do bloco adjacente entre os comandos **senão** e **fim_se**. Após as instruções de um dos blocos adjacentes são executadas as eventuais instruções que existem após o comando **fim_se**. Observe a estrutura sintática seguinte:

```

se (<condição>) então
    [instruções executadas após condição ser verdadeira]
senão
    [instruções executadas após condição ser falsa]
fim_se
    [instruções executadas após condição ser falsa ou após ser verdadeira]
  
```

Como exemplo de tomada de decisão composta em um contexto operacional considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Elaborar um programa de computador que leia dois valores numéricos reais desconhecidos. Em seguida o programa deve efetuar a adição dos dois valores lidos e caso seja o resultado maior ou igual a 10, deve ser somado a 5. Caso contrário, o valor do resultado deve ser subtraído de 7. Após a obtenção de um dos novos resultados o novo resultado deve ser apresentado.

Veja a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.4.

Entendimento

1. Definir a entrada de dois valores incógnitos (variáveis A e B).
2. Efetuar a adição dos valores A e B e atribuir o resultado da adição à variável X.
3. Verificar se o valor da variável X é maior ou igual a 10; caso seja maior ou igual a 10, proceder ao cálculo de $X + 5$, atribuindo seu resultado à variável R. Se o valor da variável X não for maior ou igual a 10, proceder ao cálculo de $X - 7$, atribuindo seu resultado à variável R.
4. Apresentar o resultado da variável R.

Diagramação

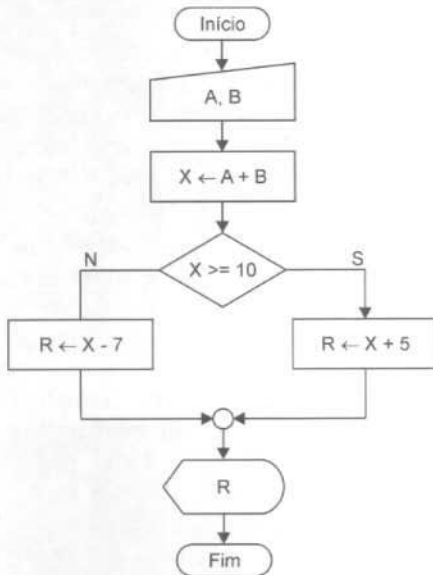


Figura 4.4 - Exemplo da utilização da instrução se...então/senão/fim_se.

Codificação

```

programa ADIÇÃO_DE_NÚMEROS_2
var
  A, B, X, R : real
início
  leia A, B
  X ← A + B
  se (X >= 10) então
    R ← X + 5
  senão
    R ← X - 7
  fim_se
  escreva R
fim
  
```

Após os tipos de variáveis, é solicitada a leitura dos valores para as variáveis **A** e **B**, depois esses valores são somados e o resultado é atribuído à variável **X**, a qual possui o resultado da adição dos dois valores lidos. Neste ponto, a condição $X \geq 10$ é avaliada e, sendo verdadeira, efetua-se uma nova soma do valor da variável **X** com o valor constante **5**, sendo o resultado atribuído à variável **R**. Caso o valor da variável **X** não seja maior ou igual a 10, o valor da variável **X** será subtraído da constante **7**, sendo seu resultado atribuído à variável **R**. Independentemente do resultado da variável **R**, ela será apresentada com seu resultado.

4.5 - Outras Formas de Desvios Condicionais

Existem ocasiões em que é necessário usar sucessivas verificações lógicas para a tomada de decisões baseadas em diversas condições. Neste sentido, são possíveis três tomadas de decisão: sequenciais, encadeadas e por seleção.

4.5.1 - Tomada de Decisão Sequencial

A tomada de decisão sequencial ocorre quando se utilizam tomadas de decisão simples ou compostas sucessivamente. A Figura 4.5 mostra de forma simplificada as duas possibilidades básicas desse tipo de estrutura de decisão.

A Figura 4.5 (a) exibe o diagrama de blocos da estrutura de tomada de decisão sequencial com base na tomada de decisão simples. No primeiro símbolo *decision* está a primeira condição (**CONDIÇÃO 1**) que, se tiver resultado lógico verdadeiro, desvia o fluxo do programa para a linha de fluxo sinalizada com o rótulo **S** e executa o primeiro bloco adjacente da primeira condição com as instruções subordinadas a essa condição, levando o fluxo do programa até o símbolo *connector* da primeira condição. Se o resultado lógico dessa primeira condição for falso, o fluxo do programa é desviado para a linha sinalizada com o rótulo **N** que leva o fluxo do programa diretamente ao símbolo *connector* da primeira condição.

Em seguida o fluxo do programa é direcionado para o segundo símbolo *decision* em que se encontra a segunda condição (**CONDIÇÃO 2**), que se tiver resultado lógico verdadeiro, direciona o fluxo do programa para a linha sinalizada com o rótulo **S** do segundo bloco adjacente da segunda condição com as instruções subordinadas a essa condição, levando o fluxo do programa até o símbolo *connector* da segunda condição. Se o resultado lógico da segunda condição for falso, o fluxo do programa é desviado para a linha sinalizada com o rótulo **N**, o que leva o fluxo do programa diretamente ao símbolo *connector* da segunda condição.

A Figura 4.5 (b) mostra o diagrama de blocos da estrutura de tomada de decisão sequencial com base no uso exclusivo de tomada de decisão composta. No primeiro símbolo *decision* está a primeira condição (**CONDIÇÃO 1**) que, se tiver resultado lógico verdadeiro, desvia o fluxo do programa para a linha sinalizada com o rótulo **S** e executa o primeiro bloco adjacente da primeira condição com as instruções subordinadas a essa condição, levando o fluxo do programa até o símbolo *connector* da primeira condição. Se o resultado lógico da primeira condição for falso, o fluxo do programa é desviado para a linha sinalizada com o rótulo **N** e executa o segundo bloco adjacente da primeira condição com as instruções subordinadas a essa condição, levando o fluxo do programa até o símbolo *connector* da primeira condição.

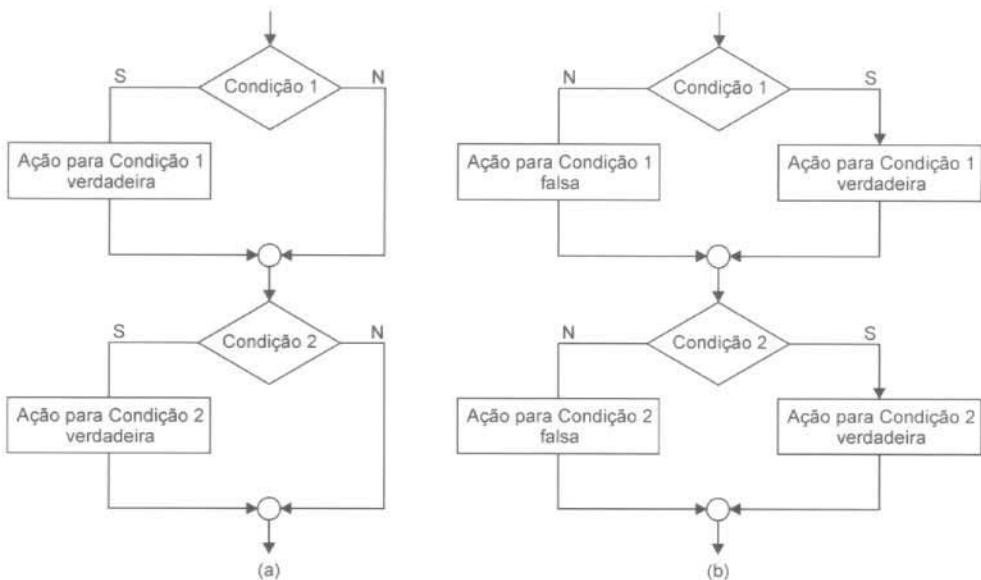


Figura 4.5 - Estrutura de tomada de decisão sequencial.

Em seguida o fluxo do programa é direcionado para o segundo símbolo *decision* no qual se encontra a segunda condição (**CONDIÇÃO 2**), que se tiver resultado lógico verdadeiro, desvia o fluxo do programa para a linha sinalizada com o rótulo **S** e executa o primeiro bloco adjacente da segunda condição com as instruções subordinadas a essa condição, levando o fluxo do programa até o símbolo *connector* da segunda condição. Se o resultado lógico da segunda condição for falso, o fluxo do programa é desviado para a linha sinalizada com o rótulo **N** e executa o segundo bloco adjacente da segunda condição com as instruções subordinadas a essa condição, levando em seguida o fluxo do programa até o símbolo *connector* da segunda condição.

A estrutura de tomada de decisão sequencial baseada em tomada de decisão simples pode ser codificada em português estruturado de acordo com a Figura 4.5 (a) da seguinte forma:

```
se (<condição 1>) então
    [ação para condição 1 verdadeira]
fim_se
se (<condição 2>) então
    [ação para condição 2 verdadeira]
fim_se
```

A estrutura de tomada de decisão sequencial baseada em tomada de decisão composta pode ser codificada em português estruturado de acordo com a Figura 4.5 (b) da seguinte forma:

```
se (<condição 1>) então
    [ação para condição 1 verdadeira]
senão
    [ação para condição 1 falsa]
fim_se
se (<condição 2>) então
    [ação para condição 2 verdadeira]
senão
    [ação para condição 2 falsa]
fim_se
```

As formas (a) e (b) apresentadas na Figura 4.5 podem ser combinadas entre si, gerando outras possibilidades. Assim sendo, podem existir tomadas de decisão sequenciais com tomadas de decisão simples em conjunto com tomadas de decisão compostas. A codificação dessas estruturas segue as formas dos respectivos diagramas de blocos.

A título de ilustração da tomada de decisão sequencial em um contexto operacional considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Desenvolver um programa que solicite a entrada de um valor numérico inteiro e apresente uma das seguintes mensagens: "você entrou o valor 1" se for dada a entrada do valor numérico 1; "você entrou o valor 2" se for dada a entrada do valor numérico 2; "você entrou valor muito baixo" se for dada a entrada de um valor numérico menor que 1 ou "você entrou valor muito alto" se for dada a entrada de um valor numérico maior que 2.

Veja a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.6.

Entendimento

1. Definir a entrada de um valor numérico inteiro (variável N).
2. Verificar se $N = 1$ e se for, apresentar a mensagem "você entrou o valor 1".

3. Verificar se $N = 2$ e se for, apresentar a mensagem "você entrou o valor 2".
4. Verificar se $N < 1$ e se for, apresentar a mensagem "você entrou valor muito baixo".
5. Verificar se $N > 2$ e se for, apresentar a mensagem "você entrou valor muito alto".

Diagramação

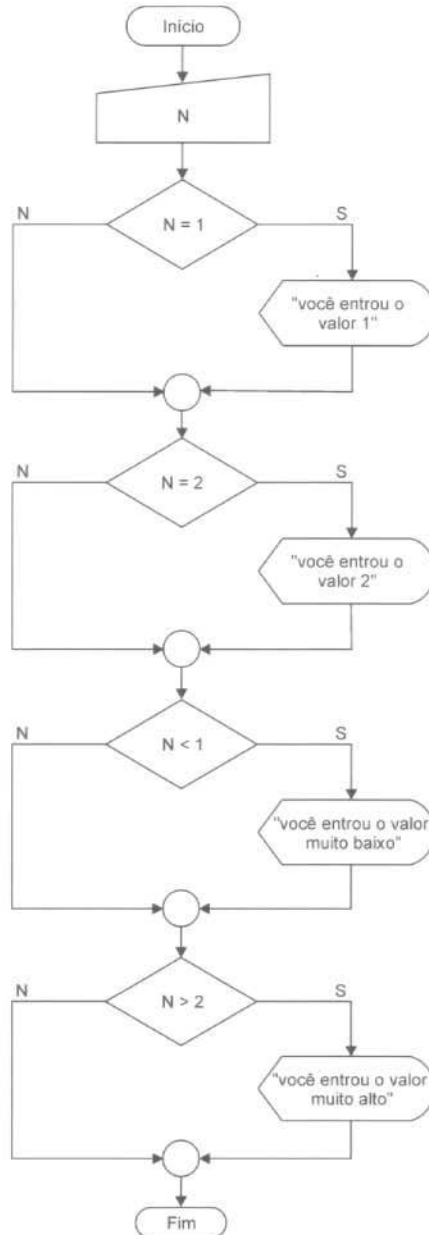


Figura 4.6 - Exemplo de estrutura de tomada de decisão sequencial.

Codificação

```
programa DECISÃO_SEQUENCIAL
var
  N : inteiro
início
  leia N
  se (N = 1) então
    escreva "você entrou o valor 1"
  fim_se
  se (N = 2) então
    escreva "você entrou o valor 2"
  fim_se
  se (N < 1) então
    escreva "você entrou valor muito baixo"
  fim_se
  se (N > 2) então
    escreva "você entrou valor muito alto"
  fim_se
fim
```

Após a indicação da variável de tipo inteiro **N**, é solicitada a leitura de um valor para ela. Assim que a leitura é realizada e o valor é fornecido para a variável **N**, ocorre uma de quatro possibilidades. Se for dada a entrada do valor **1**, é apresentada a mensagem "**você entrou o valor 1**". Se for dada a entrada do valor **2**, é apresentada a mensagem "**você entrou o valor 2**". Se for dada a entrada de um valor maior que **2**, é apresentada a mensagem "**você entrou valor muito alto**". Se for dada a entrada de um valor menor que **1**, é apresentada a mensagem "**você entrou valor muito baixo**". O programa apresenta para o usuário uma mensagem informando a ocorrência, não importa o valor fornecido.

4.5.2 - Tomada de Decisão Encadeada

A tomada de decisão encadeada ocorre quando se utilizam tomadas de decisão simples ou compostas uma dentro de outra. Uma tomada de decisão depende da outra para ser executada. As Figuras 4.7 (a) e 4.7 (b) mostram de forma simplificada as duas possibilidades básicas de estruturas de tomada de decisão encadeada representadas em seus respectivos diagramas de blocos.

A Figura 4.7 (a) apresenta o diagrama de blocos da estrutura de tomada de decisão encadeada com base no uso exclusivo de tomada de decisão simples. No primeiro símbolo *decision* está a primeira condição (**CONDIÇÃO 1**) que, se tiver resultado lógico verdadeiro, desvia o fluxo do programa para a linha sinalizada com o rótulo **S** e executa o primeiro bloco adjacente da primeira condição, que neste caso é a segunda condição (**CONDIÇÃO 2**). Sendo o resultado lógico da segunda condição verdadeiro, serão executadas as instruções subordinadas à linha de fluxo sinalizada com o rótulo **S** de seu bloco adjacente.

Em seguida o fluxo do programa é direcionado para o símbolo *connector* da segunda condição para então ser direcionado para o símbolo *connector* da primeira condição e assim continuar o programa. Se o resultado lógico da primeira condição for verdadeiro, mas falso para a segunda condição, nada acontece a não ser desviar o fluxo do programa para os símbolos *connector* da segunda e primeira condições. Se o resultado lógico da primeira condição for falso, o fluxo do programa é desviado para a linha sinalizada com o rótulo **N** que leva o fluxo para o símbolo *connector* da primeira condição que dará continuidade ao programa.

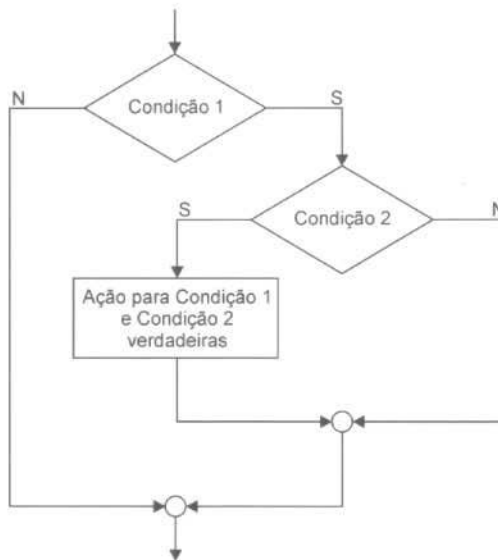


Figura 4.7 (a) - Estrutura de tomada de decisão encadeada com decisão simples.

A estrutura de tomada de decisão encadeada baseada em tomada de decisão simples pode ser codificada em português estruturado de acordo com a Figura 4.7 (a) da seguinte forma:

```

se (<condição 1>) então
  se (<condição 2>) então
    [ação para condição 1 e condição 2 verdadeiras]
  fim_se
fim_se
  
```

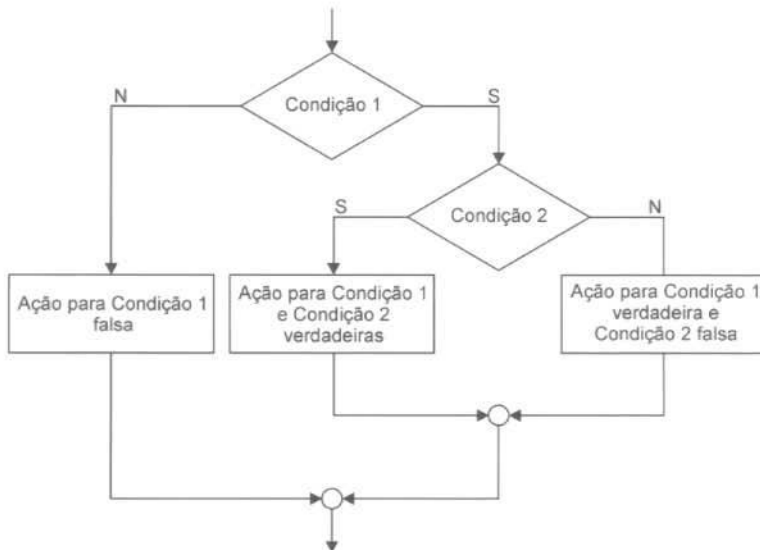


Figura 4.7 (b) - Estrutura de tomada de decisão encadeada com decisão composta.

A Figura 4.7 (b) mostra o diagrama de blocos da estrutura de tomada de decisão encadeada com base na tomada de decisão composta. No primeiro símbolo *decision* está a primeira condição (**CONDIÇÃO 1**) que, se tiver resultado lógico verdadeiro, desvia o fluxo do programa para a linha sinalizada com o rótulo **S** e executa o primeiro bloco adjacente da primeira condição, que neste caso é a segunda condição (**CONDIÇÃO 2**). Sendo o resultado lógico da segunda condição verdadeiro, são executadas as instruções subordinadas à linha sinalizada com o rótulo **S** de seu bloco adjacente.

Em seguida o fluxo do programa é direcionado para os símbolos *connector* da segunda e primeira condições e assim continua o programa. No entanto, se a primeira condição obtiver resultado lógico verdadeiro e a segunda condição obtiver resultado lógico falso, são executadas as instruções subordinadas ao bloco adjacente sinalizado na linha com o rótulo **N** da segunda condição. Em seguida o fluxo do programa é direcionado para os símbolos *connector* da segunda e primeira condições e assim continua o fluxo do programa.

Se o resultado lógico da primeira condição for falso, o fluxo é desviado para a linha sinalizada com o rótulo **N** e serão executadas as instruções subordinadas ao segundo bloco da primeira condição. Em seguida o fluxo é direcionado para o símbolo *connector* da primeira condição e assim continua a execução do programa.

A estrutura de tomada de decisão encadeada baseada em tomada de decisão composta pode ser codificada em português estruturado de acordo com a Figura 4.7 (b) da seguinte forma:

```
se (<condição 1>) então
  se (<condição 2>) então
    [ação para condição 1 e condição 2 verdadeiras]
  senão
    [ação para condição 1 verdadeira e condição 2 falsa]
  fim_se
senão
  [ação para condição 1 falsa]
fim_se
```

As formas das Figuras 4.7 (a) e 4.7 (b) podem ser combinadas, gerando uma gama maior de possibilidades. Assim, podem existir tomadas de decisão encadeadas utilizando tomada de decisão simples em conjunto com tomadas de decisão compostas. A codificação dessas estruturas segue o formato dos respectivos diagramas de blocos.

A título de ilustração de tomada de decisão encadeada em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador: entendimento, diagramação e codificação.

Desenvolver um programa de computador que calcule o reajuste de salário de um colaborador de uma empresa. Considere que o colaborador deve receber um reajuste de 15% caso seu salário seja menor que 500. Se o salário for maior ou igual a 500, mas menor ou igual a 1000, seu reajuste será de 10%; caso seja ainda maior que 1000, o reajuste deve ser de 5%.

Veja a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.8.

Entendimento

1. Ler o valor de salário atual (variável SA).
2. Verificar se o valor da variável SA é menor que 500. Se sim, reajustar o valor com mais 15%, atribuindo o novo valor à variável NS. Se não, verificar a próxima condição. Note que essa condição

estabelece o reajuste de 15% aos salários entre os valores de 0 (zero) até 499 (quatrocentos e noventa e nove). O que estiver acima dessa faixa é verificado posteriormente.

3. Verificar se o valor da variável SA é menor ou igual a 1000. Se sim, reajustar o valor com mais 10%, atribuindo o novo valor à variável NS. Essa condição estabelece o reajuste de 10% aos salários entre 500 (quinhentos - após a condição falsa do passo 3) até 1000 (mil). O que estiver fora dessa faixa é automaticamente reajustado com mais 5% atribuído à variável NS, pois trata-se de valores acima de 1000 (mil).
4. Apresentar o valor do novo salário, implicado na variável NS.

A referência feita no passo 4 já deixa determinado o reajuste de 5% para os salários maiores que 1000, não sendo necessário explicitar esta condição, pois as condições do problema já foram definidas no passo 3 e no próprio passo 4.

Diagramação

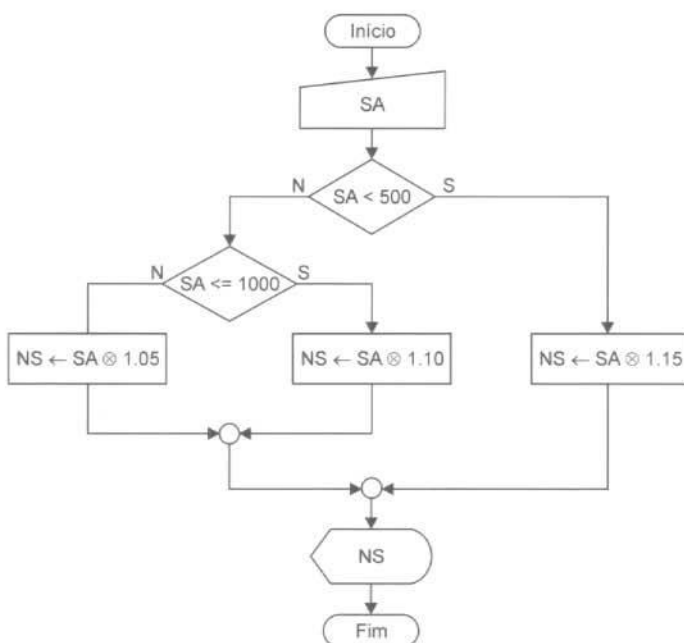


Figura 4.8 - Exemplo da utilização de tomada de decisão encadeada.

Codificação

```

programa REAJUSTA_SALÁRIO
var
  SA, NS : real
início
  leia SA
  se (SA < 500) então
    NS ← SA * 1.15
  senão
    se (SA <= 1000) então

```

```

    NS ← SA * 1.10
senão
    NS ← SA * 1.05
fim_se
fim_se
escreva NS
fim

```

O problema de reajuste de salário estabelece o uso de três condições para calcular o novo salário, sendo:

- ▶ Salário < 500, reajuste será de 15% (multiplicar salário por 1,15);
- ▶ Salário >= 500, mas <= 1000, reajuste será de 10% (multiplicar salário por 1,10);
- ▶ Salário > 1000, reajuste será de 5% (multiplicar salário por 1,05).

Na montagem do diagrama de blocos e na codificação do programa não é necessário usar explicitamente as três condições. Basta usar duas, uma vez que são utilizadas tomadas de decisão compostas e uma das condições pode ser automaticamente descartada.

4.5.3 - Tomada de Decisão por Seleção

A tomada de decisão por seleção é uma alternativa mais rápida ao uso de tomadas de decisão sequenciais ou encadeadas. Essa estrutura lógica de condição é útil e pode ser usada em situações em que se possui um grande número de verificações lógicas a serem realizadas, tanto de forma sequencial quanto de forma encadeada. Essa estrutura é um tanto limitada, como pode ser percebido mais adiante. A Figura 4.9 mostra o diagrama de blocos da estrutura de tomada de decisão por seleção.

Na Figura 4.9, após a verificação de cada condição da estrutura de decisão por seleção ocorre o desvio do fluxo para a ação prevista. Após a ação prevista o fluxo do programa é desviado para o único símbolo *connector* existente em toda a estrutura. Assim sendo, se a primeira condição (**CONDIÇÃO 1**) do primeiro símbolo *decision* possuir resultado lógico verdadeiro, são executadas as instruções subordinadas ao bloco adjacente da primeira condição indicada pela linha sinalizada com o rótulo **S**. Após a efetivação das instruções o fluxo do programa é desviado para o símbolo *connector* que dá continuidade à execução.

O mesmo raciocínio aplicado à primeira condição se aplica às demais condições previstas na Figura 4.5. Se uma das condições gerar resultado lógico falso, o fluxo do programa é desviado pela linha sinalizada pelo rótulo **N** para a próxima condição de avaliação. Se nenhuma das condições for satisfeita, executa-se a última ação antes e em cima do símbolo *connector*.

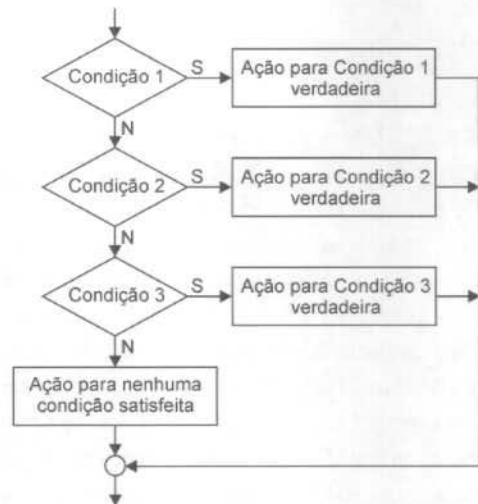


Figura 4.9 - Estrutura de tomada de decisão por seleção.

A tomada de decisão por seleção do ponto de vista da codificação em português estruturado utiliza os comandos denominados **caso**, **seja**, **faça**, **senão** e **fim_caso** na construção da instrução

caso/seja...faça/senão/fim_caso. Essa instrução utiliza uma variável após o comando **caso** que estabelece de forma indireta sua relação lógica. Após o comando **seja** definem-se os valores a serem avaliados e que darão a orientação para executar ações após o comando **faça**. Caso a variável do comando **caso** possua um valor igual a um dos valores das constantes do comando **seja**, são executadas as ações previstas em cada comando **faça**. Se o valor da variável do comando **caso** for diferente dos valores das constantes do comando **seja**, são executadas as eventuais instruções entre os comandos **senão** e **fim_caso**. Após a instrução **caso/seja...faça/senão/fim_caso**, são executadas as eventuais instruções existentes após o comando **fim_caso**. Observe a estrutura sintática seguinte:

```

caso <variável>
  seja <opção 1> faça
    [ação para condição 1 verdadeira]
  seja <opção 2> faça
    [ação para condição 2 verdadeira]
  seja <opção 3> faça
    [ação para condição 3 verdadeira]
senão
  [ação para nenhuma condição satisfeita]
fim_caso

```

Para apresentar a tomada de decisão por seleção em um contexto operacional, considere o problema a seguir observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Desenvolver um programa de computador que leia um valor numérico inteiro entre os valores 1 e 12 e apresente por extenso o nome do mês correspondente ao valor entrado. Caso sejam fornecidos valores menores que 1 e maiores que 12, o programa deve apresentar a mensagem "Valor inválido".

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.10.

Entendimento

1. Efetuar a leitura de um valor numérico inteiro (variável MÊS).
2. Se a variável MÊS for igual a 1, apresentar a mensagem "Janeiro".
3. Se a variável MÊS for igual a 2, apresentar a mensagem "Fevereiro".
4. Se a variável MÊS for igual a 3, apresentar a mensagem "Março".
5. Se a variável MÊS for igual a 4, apresentar a mensagem "Abril".
6. Se a variável MÊS for igual a 5, apresentar a mensagem "Maio".
7. Se a variável MÊS for igual a 6, apresentar a mensagem "Junho".
8. Se a variável MÊS for igual a 7, apresentar a mensagem "Julho".
9. Se a variável MÊS for igual a 8, apresentar a mensagem "Agosto".
10. Se a variável MÊS for igual a 9, apresentar a mensagem "Setembro".
11. Se a variável MÊS for igual a 10, apresentar a mensagem "Outubro".
12. Se a variável MÊS for igual a 11, apresentar a mensagem "Novembro".
13. Se a variável MÊS for igual a 12, apresentar a mensagem "Dezembro".
14. Se a variável MÊS for menor que 1 ou maior que 12, apresentar a mensagem "Valor inválido".

Diagramação

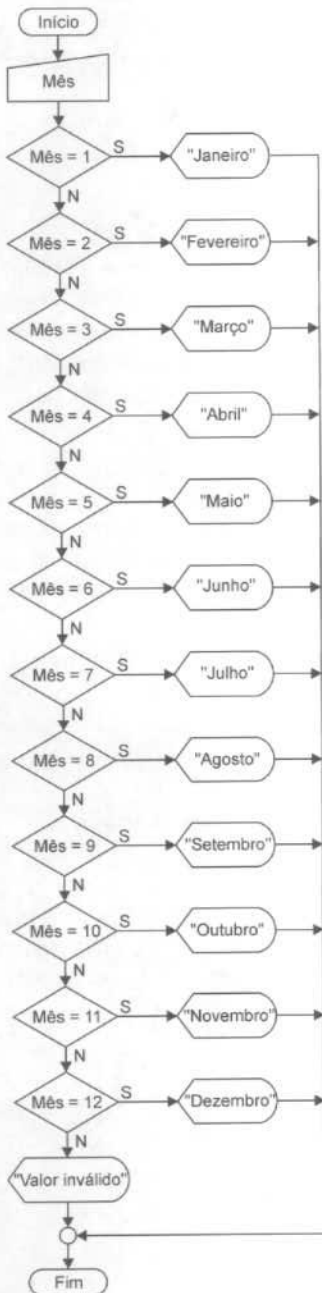


Figura 4.10 - Exemplo da tomada de decisão por seleção.

Codificação

```

programa MÊS_POR_EXTENSO
var
    MÊS : inteiro
início
    leia MÊS
    caso MÊS
        seja 1 faça
            escreva "Janeiro"
        seja 2 faça
            escreva "Fevereiro"
        seja 3 faça
            escreva "Março"
        seja 4 faça
            escreva "Abril"
        seja 5 faça
            escreva "Maio"
        seja 6 faça
            escreva "Junho"
        seja 7 faça
            escreva "Julho"
        seja 8 faça
            escreva "Agosto"
        seja 9 faça
            escreva "Setembro"
        seja 10 faça
            escreva "Outubro"
        seja 11 faça
            escreva "Novembro"
        seja 12 faça
            escreva "Dezembro"
        senão
            escreva "Valor inválido"
    fim_caso
fim
    
```

A instrução **caso/seja...faça/senão/fim_caso** é útil apenas nos casos em que há tomadas de decisão sequenciais ou encadeadas utilizando as ações previstas com o operador relacional "igual a".

Outro detalhe importante é que o comando **senão** na estrutura de tomada de decisão por seleção é opcional. Ele pode ser omitido quando não se desejar deixar definida uma ação quando as condições gerais não forem satisfeitas.

4.6 - Operadores Lógicos

Este capítulo apresentou situações que demonstram a tomada de decisão com apenas uma condição. No entanto, o que fazer quando houver necessidade de tomar uma única decisão com várias condições?

É nesse momento que se utilizam ferramentas auxiliares chamadas *operadores lógicos* (não confundir com operadores relacionais). Os operadores lógicos são também referenciados como *operadores booleanos* que recebem este nome devido à contribuição do matemático inglês George Boole que, em 1854, criou um sistema de relação lógica, base para o modelo computacional digital utilizado até hoje (MACHADO & MAIA, 2002, p. 5).

Os operadores lógicos (ou booleanos) mais comuns, do ponto de vista da programação, são quatro: operador lógico **.e.** (operador lógico de conjunção), operador lógico **.ou.** (de disjunção inclusiva), operador lógico **.xou.** (de disjunção exclusiva) e operador lógico **.não.** (de negação). Dos quatro operadores três trabalham diretamente com mais de uma condição, vinculando-as entre si para que seja tomada uma única decisão, sendo os operadores lógicos **.e.**, **.ou.** e **.xou.**. O operador lógico **.não.** pode ser usado sempre à frente de uma condição no sentido de inverter seu resultado lógico.

É pertinente salientar que qualquer operador lógico em uso possibilita obter uma de duas respostas lógicas possíveis. A condição avaliada pode ter resultado lógico falso ou verdadeiro.

Por questões de conveniência, os exemplos deste tópico usam tomadas de decisão simples, no entanto os operadores lógicos podem ser utilizados com tomadas de decisão compostas, sequenciais e encadeadas sem nenhum problema.

4.6.1 - Operador Lógico de Conjunção

Do ponto de vista filosófico, a lógica de conjunção é a relação lógica entre duas ou mais proposições (entende-se, na esfera da programação, o termo proposição como sendo condição) que geram um resultado lógico verdadeiro quando todas as proposições forem verdadeiras. Em seguida é apresentada a tabela verdade para o operador lógico **.e.**:

Tabela verdade do operador lógico de conjunção		
Condição 1	Condição 2	Resultado lógico
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

O raciocínio apresentado na tabela verdade para o operador lógico de conjunção pode ser exemplificado de acordo com o diagrama de Venn¹⁷, conforme a Figura 4.11, que mostra que apenas parte da interseção entre a totalidade dos círculos está preenchida. Isso indica que algo é verdadeiro para parte do todo quando é verdadeiro para o todo.

O uso de um operador lógico de conjunção em um diagrama de blocos é demonstrado na Figura 4.12 com um exemplo de tomada de decisão simples.



Figura 4.11 - Diagrama de Venn para o operador de conjunção.

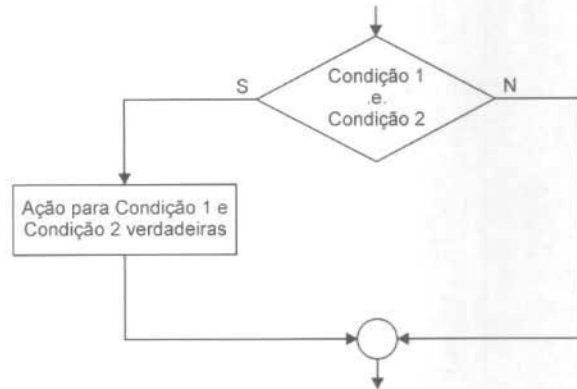


Figura 4.12 - Estrutura de tomada de decisão com operador de conjunção.

A codificação em português estruturado do operador de conjunção utilizado na Figura 4.12 é realizada de acordo com o modelo do seguinte trecho de programa:

```
se (<condição 1> .e. <condição 2>) então
    [ação para condição 1 e condição 2 verdadeiras]
fim_se
```

O código em português estruturado mostra o uso dos sinais de parênteses para indicar cada condição em separado. Note o uso do operador lógico `.e.` entre as condições envolvidas na relação lógica de conjunção.

Como exemplo da tomada de decisão com operador lógico de conjunção em um contexto operacional considere o problema a seguir, observando detalhadamente as tarefas de um programador de computador: entendimento, diagramação e codificação.

Desenvolver um programa de computador que leia um valor numérico inteiro que esteja na faixa de valores entre 20 e 90. O programa deve apresentar a mensagem "O valor está na faixa permitida", caso o valor informado esteja entre 20 e 90. Se o valor estiver fora da faixa permitida, o programa deve apresentar a mensagem "O valor está fora da faixa permitida".

Veja a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.13.

¹⁷ Forma de representação da álgebra booleana em consonância com a teoria de conjuntos apresentada em 1881 pelo Padre John Venn, matemático, filósofo e professor inglês, nascido em 4 de agosto de 1834 e falecido em 4 de abril de 1923.

Entendimento

1. Efetuar a leitura de um valor numérico inteiro (variável NÚMERO).
2. Verificar se o valor fornecido é maior ou igual a 20 e se o mesmo valor é menor ou igual a 90. Se esta condição for verdadeira, apresentar a mensagem "O número está na faixa de 20 a 90"; caso contrário, apresentar a mensagem "O número está fora da faixa de 20 a 90".

Diagramação

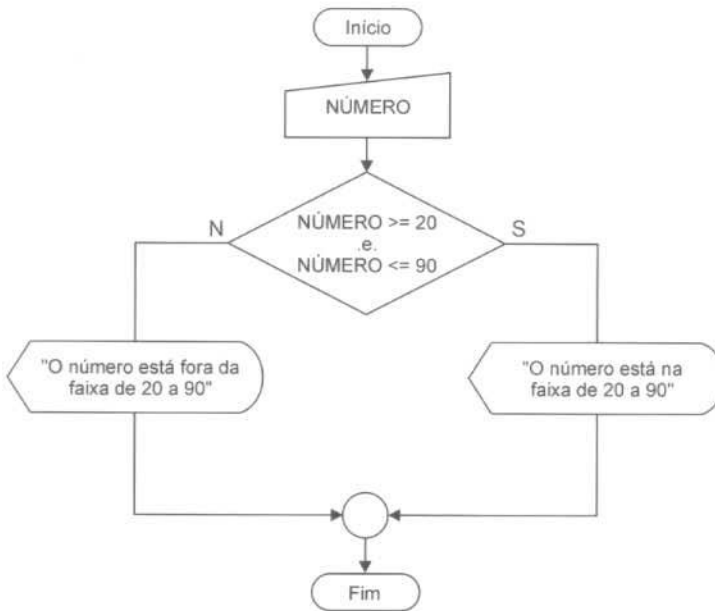


Figura 4.13 - Utilização de tomada de decisão com operador lógico de conjunção.

Codificação

```

programa TESTA_LÓGICA_E
var
  NÚMERO : inteiro
início
  leia NÚMERO
  se (NÚMERO >= 20) .e. (NÚMERO <= 90) então
    escreva "O número está na faixa de 20 a 90"
  senão
    escreva "O número está fora da faixa de 20 a 90"
  fim_se
fim
  
```

Se no programa anterior for dada a entrada de um valor menor que 20 ou maior que 90, será apresentada a mensagem "O número está fora da faixa de 20 a 90". No entanto, qualquer valor maior ou igual a 20 e menor ou igual a 90 fará com que a mensagem "O número está na faixa de 20 a 90" seja apresentada.

Tome por base o valor 50 e observe que é maior ou igual a 20 e menor ou igual a 90. Sendo a condição $(NÚMERO \geq 20)$.e. $(NÚMERO \leq 90)$ verdadeira, será apresentada a mensagem "O número está na faixa

de 20 a 90". No entanto, se for dada a entrada do valor 10, a condição **(NÚMERO >= 20)** e **(NÚMERO <= 90)** será falsa, uma vez que 10 não é maior ou igual a 20, apesar de ser menor ou igual a 90. O operador lógico de conjunção exige que todas as condições da expressão lógica sejam verdadeiras para que seu resultado lógico seja verdadeiro.

4.6.2 - Operador Lógico de Disjunção Inclusiva

Do ponto de vista filosófico, lógica de disjunção inclusiva é a relação lógica entre duas ou mais proposições de tal modo que seu resultado lógico será verdadeiro quando pelo menos uma das proposições for verdadeira. Em seguida, é apresentada a tabela verdade para o operador lógico **.ou.**:

Tabela verdade do operador lógico de disjunção inclusiva		
Condição 1	Condição 2	Resultado lógico
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

O raciocínio apresentado na tabela verdade para o operador lógico de disjunção inclusiva está exemplificado de acordo com o diagrama de Venn na Figura 4.14.

A Figura 4.14 mostra que não só a parte de interseção dos círculos está preenchida, como também a totalidade dos círculos. Isso indica que será algo verdadeiro para a parte do todo quando for verdadeiro para qualquer parte desse todo.

A representação de um operador lógico de disjunção inclusiva em um diagrama de blocos está na Figura 4.15 que mostra um exemplo de tomada de decisão simples.



Figura 4.14 - Diagrama de Venn para o operador de disjunção inclusiva.

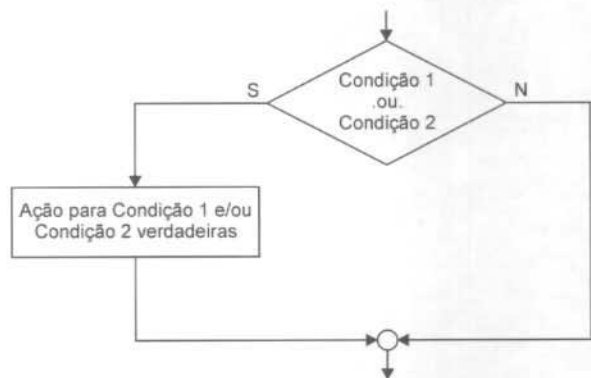


Figura 4.15 - Estrutura de tomada de decisão com operador de disjunção inclusiva.

A codificação em português estruturado do operador de disjunção inclusiva da Figura 4.15 é realizada de acordo com o modelo do seguinte trecho de programa:

```

se (<condição 1> .ou. <condição 2>) então
  [ação para condição 1 e/ou condição 2 verdadeiras]
fim_se

```

O código em português estruturado usa parênteses para indicar cada condição da decisão em separado. Note o uso do operador lógico **.ou.** entre as condições envolvidas na relação lógica de disjunção inclusiva.

A título de ilustração da tomada de decisão com operador lógico de disjunção inclusiva em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Desenvolver um programa que solicite a entrada do sexo de uma pessoa e indique se a informação fornecida é ou não válida. Para o sexo MASCULINO informe a entrada da letra M e para o sexo FEMININO da letra F. Se forem fornecidos os valores M e F, o programa deve apresentar uma mensagem avisando que o sexo informado é válido. No entanto, se for fornecido qualquer outro valor, o programa deve informar que o sexo fornecido é inválido.

Veja a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.16.

Entendimento

1. Efetuar a entrada do sexo de uma pessoa (variável SEXO).
2. Verificar se o valor fornecido para a variável SEXO é válido, ou seja, se o valor da variável SEXO é igual a F ou M. Se o valor for válido, apresentar a mensagem "Sexo válido"; caso contrário, apresentar a mensagem "Sexo inválido".

Diagramação

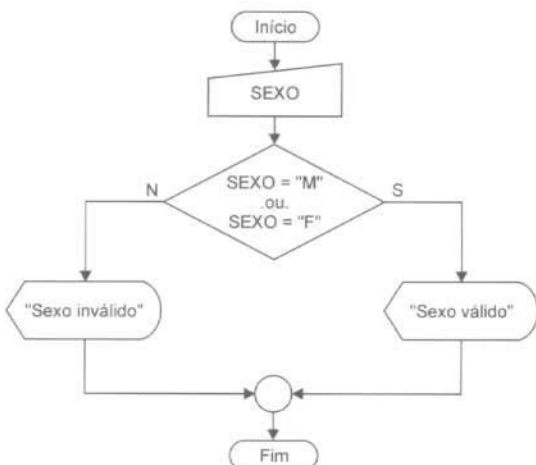


Figura 4.16 - Exemplo de tomada de decisão com operador lógico de disjunção inclusiva.

Codificação

```

programa TESTA_LÓGICA_OU
var
  SEXO : caractere
início
  leia SEXO
  se (SEXO = "M") .ou. (SEXO = "F")
  então
    escreva "Sexo válido"
  senão
    escreva "Sexo inválido"
  fim_se
fim

```

Se no programa anterior for entrado um valor de sexo como M ou F, aparece a mensagem "Sexo válido". Se for fornecido qualquer outro valor diferente de F ou M, será a mensagem "Sexo inválido".

Tome por base a entrada do valor F para o sexo FEMININO. Observe que o valor F não é igual ao valor "M" da primeira condição, mas é igual ao valor "F" da segunda condição. Sendo a condição **(SEXO = "M")** .ou. **(SEXO = "F")** verdadeira, será apresentada a mensagem "Sexo válido". No entanto, se for dada a entrada de algum outro valor para um dos sexos, a condição **(SEXO = "M")** .ou. **(SEXO = "F")** será falsa. O operador lógico de disjunção inclusiva exige que pelo menos uma das condições da expressão lógica seja verdadeira para que seu resultado lógico seja verdadeiro.

4.6.3 - Operador Lógico de Negação

Do ponto de vista filosófico negação é a rejeição ou a contradição do todo ou de parte desse todo. Pode ser a relação entre uma proposição p e sua negação $\text{não-}p$. Se p for verdadeira, $\text{não-}p$ é falsa e se p for falsa, $\text{não-}p$ é verdadeira. Em seguida é apresentada a tabela verdade para o operador lógico .não.:

Tabela verdade do operador lógico de negação	
Condição	Resultado lógico
Verdadeiro	Falso
Falso	Verdadeiro

A representação de um operador lógico de negação em um diagrama de blocos está na Figura 4.17 com um exemplo de tomada de decisão simples.

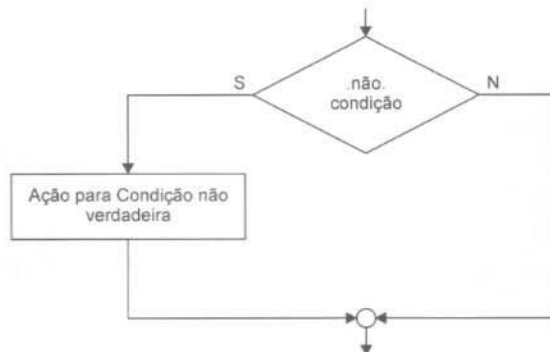


Figura 4.17 - Estrutura de tomada de decisão com operador de negação.

A codificação em português estruturado do operador de negação da Figura 4.17 é realizada de acordo com o modelo do seguinte trecho de programa:

```
se .não. (<condição>) então
    [ação para condição não verdadeira]
fim_se
```

O código em português estruturado mostra o operador lógico **.não.** à frente da condição a ser avaliada. Se a condição for verdadeira, o operador **.não.** fará com que a condição seja considerada falsa, e neste caso nenhuma das instruções subordinadas do bloco adjacente entre os comandos **se...então** e **fim_se** será executada. No entanto, se a condição for falsa, o operador **.não.** fará com que a condição seja considerada verdadeira, e neste caso serão executadas as instruções subordinadas do bloco adjacente dos comandos **se...então** e **fim_se**.

A tomada de decisão com o operador lógico de negação em um contexto operacional é demonstrada no problema a seguir. Observe detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Elaborar um programa de computador que leia três valores numéricos inteiros, sendo dois representados pelas variáveis A e B e que serão utilizados para a elaboração de um de dois cálculos programados: $A + B$ e $A - B$. O terceiro representado pela variável X será um valor chave de seleção da operação a ser efetuada. Se o valor da variável X não for maior que 5, será realizada a operação $C \leftarrow A + B$; caso contrário, deve ser realizada a operação $C \leftarrow A - B$. Ao final o programa deve apresentar o resultado armazenado na variável C.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.18.

Entendimento

1. Efetuar a entrada, respectivamente, dos valores das variáveis A, B e X.
2. Verificar se o valor fornecido para a variável X realmente não é maior que 5. Sendo esta condição verdadeira, processar a operação $C \leftarrow A + B$; caso contrário, deve ser realizada a operação $C \leftarrow A - B$.
3. Apresentar o resultado obtido na variável C.

Diagramação

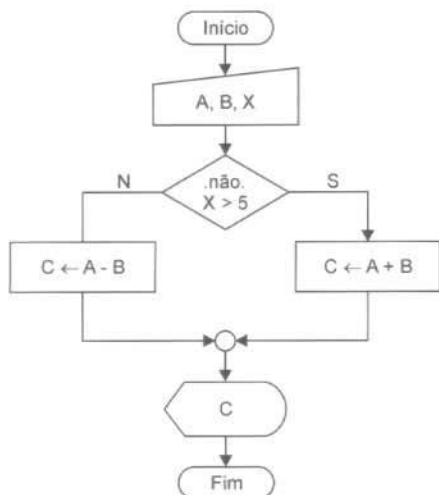


Figura 4.18 - Exemplo de tomada de decisão com operador lógico de negação.

Codificação

```

programa TESTA_LÓGICA_NÃO
var
  A, B, C, X : inteiro
início
  leia A, B, X
  se .não. (X > 5) então
    C ← A + B
  senão
    C ← A - B
  fim_se
  escreva C
fim
  
```


Se forem informados os valores 5, 1 e 2, respectivamente, para as variáveis A , B e X , resulta para a variável C o valor 6, pois o valor 2 da variável X é controlado pela instrução **se .não. ($X > 5$) então**, como sendo verdadeiro, uma vez que **não é maior que** 5. Os valores 5 e 1 são somados resultando 6. Mas se forem informados os valores 5, 1, e 6, respectivamente, para as variáveis A , B e X , resulta para a variável C o valor 4, pois o valor 6 da variável X é controlado pela instrução **se .não. ($X > 5$) então** como sendo falso.

Para os olhos de um novato em programação pode parecer o operador lógico **.não.** uma ferramenta sem sentido, mas esse operador é de grande valia em várias situações que envolvem principalmente tomadas de decisão do tipo simples. No entanto, existe uma situação em particular que sem dúvida justifica sua existência.

Imagine que você deve tomar uma decisão apenas se a condição for falsa, ou seja, se a condição for verdadeira, nada deve ser feito. Esse tipo de ocorrência é muito comum no contexto humano e por esta razão é de fácil solução, mas no contexto computacional, apesar de ser aceita, essa ocorrência não pode ser programada. Ela se torna um grave problema operacional.

A fim de ilustrar esta ocorrência, de certa forma bizarra, imagine uma condição baseada na questão "você está saudável?". Se estiver saudável, não precisa fazer nada, mas caso esteja doente, é necessário procurar acompanhamento médico.

Dentro do exposto é possível que o leitor esteja pensando em solucionar o problema com base na estrutura gráfica do diagrama de blocos da Figura 4.19, o que seria uma ótima ideia do ponto de vista humano, mas do ponto de vista computacional nada feito, uma vez que não é possível estruturar essa construção lógica em um computador.

O problema computacional em relação à estrutura lógica da Figura 4.19 está no fato de não poder ser codificada em uma linguagem de programação, não sendo exceção o português estruturado.

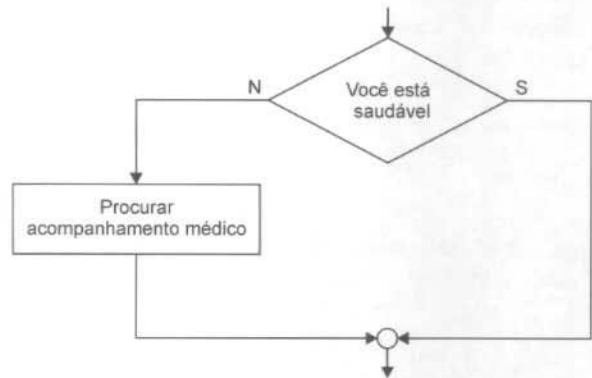


Figura 4.19 - Estrutura de tomada de decisão sem o operador de negação.

Em uma tomada de decisão simples o bloco adjacente de instruções subordinadas só é executado quando a condição é considerada verdadeira. Assim sendo, não é possível codificar a estrutura lógica da Figura 4.19. Pode ser que venha à mente a possibilidade de manter o desenho do diagrama de blocos como está (no estilo tomada de decisão simples) e então usar o código em português estruturado como se fosse uma tomada de decisão composta, deixando o trecho de código entre os comandos **se...então** e **senão** em branco e colocando a ação desejada no trecho de comandos **senão** e **fim_se**. Isso seria errado, uma vez que não pode haver, em hipótese nenhuma, em uma estrutura de tomada de decisão trechos em branco entre os comandos **se...então** e **senão**. De fato o programador está acuado e a solução plausível é o uso do operador lógico de negação para validar a ação lógica desejada na esfera computacional. Observe a solução apresentada na Figura 4.20.

Com o operador lógico de negação fica fácil resolver o problema apresentado, de forma que um computador consiga processar a ação, simulando um contexto humano, que não pode ser executado de forma direta por um computador.

Pode até parecer que, inicialmente, o operador lógico de negação seja um pouco confuso, mas tudo é uma questão de costume. À medida que os problemas vão surgindo, o programador novato vai se adaptando e passa a usar os recursos que aprendeu de uma forma mais natural.

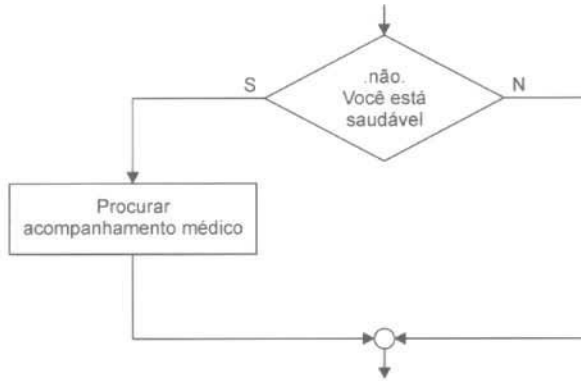


Figura 4.20 - Estrutura de tomada de decisão com o operador de negação.

Na Figura 4.18 observe o código válido em português estruturado para o problema exposto e solucionado na Figura 4.19.

```

se .não. (você está saudável) então
  procurar acompanhamento médico
fim_se
  
```

Para a avaliação da questão "não você está saudável?" considere a ação "procurar acompanhamento médico". A questão "não você está saudável?" pode ser entendida como "você não está saudável?".

4.6.4 - Operador Lógico de Disjunção Exclusiva

Do ponto de vista filosófico a lógica de disjunção exclusiva é a relação que avalia duas ou mais proposições, de tal modo que o resultado lógico será verdadeiro quando uma e apenas uma das proposições for verdadeira. Em seguida é apresentada a tabela verdade para o operador lógico **.xou.** (podendo ser o operador lógico de disjunção exclusiva referenciado em português estruturado também como **.oue.**):

Tabela verdade do operador lógico de disjunção exclusiva		
Condição 1	Condição 2	Resultado lógico
Verdadeiro	Verdadeiro	Falso
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

O raciocínio apresentado na tabela verdade para o operador lógico de disjunção exclusiva pode ser simplificado de acordo com o diagrama de Venn na Figura 4.21.

Na Figura 4.21 estão preenchidas apenas as partes externas ao ponto de interseção dos círculos. Isso indica que algo é verdadeiro para o todo quando for verdadeiro para uma das partes.

A representação de um operador lógico de disjunção exclusiva em um diagrama de blocos está na Figura 4.22 com um exemplo de tomada de decisão simples.



Figura 4.21 - Diagrama de Venn para o operador de disjunção exclusiva.

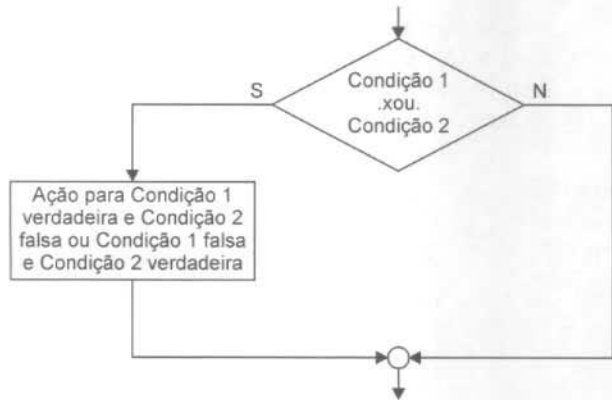


Figura 4.22 - Estrutura de tomada de decisão com operador de disjunção exclusiva.

A codificação em português estruturado do operador de disjunção exclusiva utilizado na Figura 4.22 é realizada de acordo com o modelo do seguinte trecho de programa:

```
se (<condição 1>) .xou. (<condição 2>) então
  [ação para condição 1 verdadeira e condição 2 falsa ou condição 1 falsa e condição 2
  verdadeira]
fim_se
```

No código em português estruturado observe os parênteses para cada condição em separado. Note o uso do operador lógico **.xou.** entre as condições envolvidas na relação lógica de disjunção exclusiva.

A título de ilustração da tomada de decisão com operador lógico de disjunção exclusiva em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Desenvolver um programa de computador que efetue a entrada do nome e respectivo sexo de duas pessoas que pretendem formar um par para participar de uma quadrilha em uma festa junina. Os administradores da festa determinaram que somente serão aceitos pares heterogêneos (formados por pessoas de sexos diferentes). Não serão aceitos casais formados por pessoas do mesmo sexo. Para atender este requisito o programa deve, após a entrada do sexo das duas pessoas, verificar se elas formam par, e no caso deve apresentar uma mensagem informando esta possibilidade. Caso contrário, o programa deve indicar a impossibilidade de formação de par.

Veja a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.23.

Entendimento

1. Efetuar a entrada do nome (variável N1) e do sexo (variável S1) da primeira pessoa.
2. Efetuar a entrada do nome (variável N2) e do sexo (variável S2) da segunda pessoa.
3. Verificar se o sexo da primeira pessoa é exclusivamente igual a um determinado sexo e se o sexo da segunda pessoa é exclusivamente igual ao sexo informado exclusivamente para a primeira pessoa. Se um dos sexos for exclusivamente igual a um dos sexos informados e o outro for diferente, o programa apresenta uma mensagem informando que podem ser formados pares. Caso contrário, o programa apresenta uma mensagem informando que o par não pode ser formado.

Diagramação

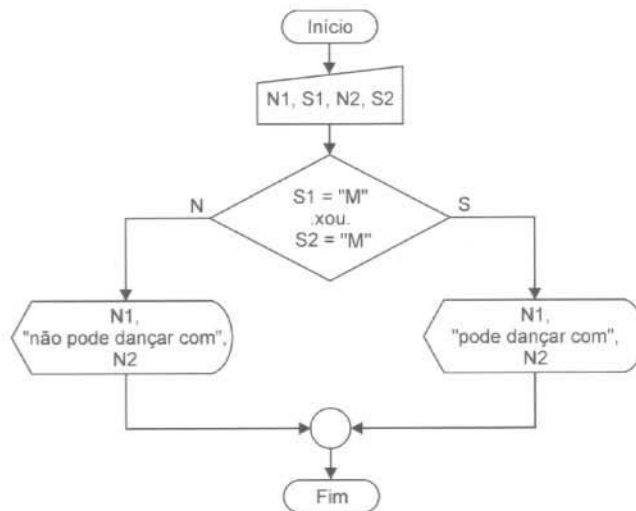


Figura 4.23 - Exemplo de tomada de decisão com operador lógico de disjunção exclusiva.

Codificação

```

programa TESTA_LOGICA_XOU
var
  N1, N2 : cadeia
  S1, S2 : caractere
inicio
  leia N1, S1
  leia N2, S2
  se (S1 = "M") xou (S2 = "M") então
    escreva N1, " pode dançar com ", N2
  senão
    escreva N1, " não pode dançar com ", N2
  fim_se
fim
  
```

No exemplo apresentado, se forem fornecidos sexos diferentes na entrada para os nomes das duas pessoas (masculino e feminino ou feminino e masculino), o programa confirma a possibilidade de formação de par. Se forem fornecidos sexos iguais (masculino e masculino ou feminino e feminino), o programa indica a não possibilidade de formação de par.

Tome por base a entrada do valor F para o sexo FEMININO da primeira pessoa e do valor M para o sexo MASCULINO da segunda pessoa. Observe que o valor F do sexo da primeira pessoa não é igual ao valor "M" da primeira condição, mas o valor M do sexo da segunda pessoa é igual ao valor "M" da segunda condição. Sendo a condição **(SEXO = "M") .ou. (SEXO = "M")** verdadeira, será apresentada a mensagem informando que essas pessoas podem formar o par de dança. No entanto, se for dada a entrada dos valores M ou F, respectivamente, para as duas pessoas, a condição **(SEXO = "M") .ou. (SEXO = "M")** será falsa. O operador lógico de disjunção exclusiva exige que uma das condições da expressão lógica seja verdadeira para que seu resultado lógico seja verdadeiro.

A expressão lógica **(C1) .ou. (C2)**, considerando C1 a condição <condição 1> e C2 a condição <condição 2>, pode ser matematicamente representada pela expressão lógica **(C1 .e. (.não. C2)) .ou. ((.não. C1) .e. C2)**. É importante ao futuro programador saber este detalhe, pois há linguagens de programação que não possuem o operador lógico de conjunção exclusiva. Neste caso, é necessário conhecer uma forma alternativa para a solução do problema, usando algoritmos.

4.6.5 - Precedência de Uso dos Operadores Lógicos

Os operadores lógicos (**.e.**, **.ou.** e **.xou.**) possibilitam o uso de mais de uma condição para a tomada de uma única decisão. Já o operador lógico **.não.** tem por finalidade a negação do estado lógico de uma determinada condição.

Para usar adequadamente os operadores lógicos em expressões lógicas, é necessário levar em consideração a ordem de precedência desses operadores. A tabela de precedência de operadores lógicos apresenta a ordem hierárquica de execução dos operadores lógicos que podem ser utilizados em uma expressão lógica.

Tabela de precedência de operadores lógicos		
Operador	Operação	Precedência
.não.	Negação	1
.e.	Conjunção	2
.ou.	Disjunção inclusiva	3
.xou.	Disjunção exclusiva	3

A tabela de precedência dos operadores lógicos indica a prioridade da ordem de execução em uma expressão lógica. O operador **.não.** é o que possui maior nível de prioridade, portanto deve ser operacionalizado em primeiro lugar. Num segundo momento tem-se o operador lógico **.e.** que possui médio nível de prioridade e por último os operadores **.ou.** e **.xou.** que possuem baixo nível de prioridade.

Por exemplo, a expressão lógica **(A = B) .e. .não. (A > 5)** deve ser avaliada a partir de **.não. (A > 5)** e somente depois de saber seu resultado é que pode ser realizada avaliação do restante da expressão lógica.

Já na expressão lógica **(A = 1) .xou. (A >= 4) .e. (A <= 9)** será resolvida primeiramente a parte da expressão submetida ao operador lógico **.e.**, depois a parte da expressão submetida ao operador **.xou.**, segundo a prioridade padrão. Imagine que haja necessidade de executar primeiramente a avaliação lógica da

expressão, levando em consideração que o operador **.xou.** seja avaliado em primeiro lugar. Neste caso, a avaliação lógica deve ficar entre parênteses como sendo $((A = 1) .xou. (A >= 4)) .e. (A <= 9)$. Lembre-se de que o uso de parênteses possibilita a mudança de prioridade tanto de expressões aritméticas (processamento matemático) como de expressões lógicas (processamento lógico).

4.7 - Divisibilidade: Múltiplos e Divisores

Divisibilidade é a qualidade do que é divisível. Neste contexto duas questões devem ser conhecidas e entendidas, pelo menos em um nível considerado básico por qualquer programador, sendo os múltiplos e os divisores dos números naturais. Entende-se por número natural um valor numérico que seja inteiro e positivo.

Múltiplos são os resultados obtidos a partir da multiplicação de dois números naturais, enquanto divisores são os números que dividem outros números com o objetivo de gerar um resultado de divisão exato, ou seja, obter resto de divisão sempre zero. Quando o resto de uma divisão de números naturais é igual a zero, ocorre uma divisibilidade. Perceba que valores do tipo real não entram nesta abordagem.

Pelo fato de todo número natural ser múltiplo de si mesmo, uma forma de trabalhar com esses valores é fazer as operações de cálculo com divisores. Em uma divisão existem alguns termos que precisam ser conhecidos: dividendo (valor que será dividido), divisor (valor que divide o dividendo), quociente (resultado da divisão do dividendo pelo divisor) e resto (valor que sobra da operação de divisão). A Figura 4.24 mostra o processo de divisão de dois números naturais.

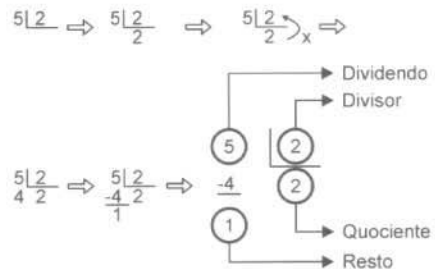


Figura 4.24 - Exemplo de divisão de dois números naturais.

De acordo com a Figura 4.24, fica fácil deduzir como obter a divisibilidade de um dividendo (valor 5) por um divisor (valor 2). Considere a expressão matemática **Resto = Dividendo - Divisor . Quociente** para obter a divisibilidade, na qual as variáveis **Dividendo**, **Divisor** e **Quociente** serão substituídas por valores inteiros positivos. Assim sendo, considere os valores de dividendo 5 e divisor 2 na expressão matemática **Resto = 5 - 2 . 2**. Basta realizar primeiramente o cálculo da multiplicação **2 . 2**, para obter o valor 4 que será subtraído do valor 5 e resultará o valor 1 para a variável **Resto**, ou seja, **Resto** é diferente de zero. Neste caso não ocorreu a divisibilidade do valor 5 pelo valor 2 porque o valor 2 não é múltiplo do valor 5.

A expressão matemática **Resto = Dividendo - Divisor . Quociente**, do ponto de vista matemático, é obtida da seguinte equação matemática:

$$r = a - n \left\lfloor \frac{a}{n} \right\rfloor$$

A variável **r** representa o resto da divisão, a variável **a** representa o dividendo e a variável **n** o divisor (BOUTE, 1992 & LEIJEN, 2001). Observe a indicação do cálculo do quociente com a função *parte inteira inferior* que obtém o resultado inteiro da divisão do dividendo **a** pelo divisor **n**. A função *parte inteira* tem por objetivo retornar a parte inteira (o expoente) de um número real, desconsiderando as casas decimais (a mantissa) desse número.

A função *parte inteira* pode ser representada de duas formas: *parte inteira inferior*, também chamada *chão*, e *parte inteira superior*, também chamada *teto* (IVERSON, 1962). Na forma *inferior* retorna o expoente do valor, desconsiderando toda a mantissa desse valor. Já na forma *superior* retorna o arredondamento do valor para o valor inteiro mais próximo. As funções *parte inteira inferior* e *parte inteira superior* são graficamente representadas como:

$$\lceil x \rceil \text{ Parte inteira superior}$$

$$\lfloor x \rfloor \text{ Parte inteira inferior}$$

Do ponto de vista computacional, a expressão matemática para obter a divisibilidade dos valores deve ser convertida em uma expressão aritmética. Assim sendo, a expressão matemática **Resto = Dividendo – Divisor X Quociente** deve ser escrita no formato **Resto ← Dividendo – Divisor * (Dividendo div Divisor)**. O operador aritmético **div** apresentado no tópico 3.2 faz o cálculo da divisão para obter o quociente inteiro.

Considerando um programa que verifique se o valor **5** é divisível pelo valor **2**, ele seria escrito na forma de expressão aritmética como **Resto ← 5 – 2 * (5 div 2)**. Para obter o valor do quociente inteiro utiliza-se a operação **(5 div 2)** que neste caso retorna o quociente **2** e não **2,5**. O uso de parênteses na expressão aritmética é normalmente obrigatório, pois o operador **div** possui a mais baixa prioridade de cálculo. No entanto, pode ocorrer de alguma linguagem de programação usar o operador aritmético **div** (obtenção de quociente inteiro) com a mesma prioridade do operador aritmético **/** (obtenção de quociente real). Assim sendo, a expressão aritmética **Resto ← Dividendo – Divisor * (Dividendo div Divisor)** é considerada genérica e de fácil implementação em qualquer linguagem de programação.

O uso de múltiplos e divisores na área de desenvolvimento de programação é comum. São várias as ocasiões em que é preciso fazer operações com a divisibilidade de valores numéricos. A forma mais utilizada de operações com divisibilidade de valores é o cálculo de dígitos verificadores encontrados em números de matrícula escolar, de conta-corrente, de cartões de crédito, entre outros.

O dígito verificador é um mecanismo que visa garantir a validade e a autenticidade de um número de registro, sendo um ou mais caracteres acrescidos ao número de registro original, normalmente caracteres numéricos, calculados a partir do próprio número de registro por algoritmos específicos. Existem vários algoritmos para efetuar o cálculo de dígitos verificadores. Alguns são clássicos e outros podem ser desenvolvidos para uso particular.

A expressão aritmética **Resto ← Dividendo – Divisor * (Dividendo div Divisor)** é uma operação algorítmica genérica, que pode ser facilmente adaptada para qualquer linguagem de programação. No entanto, há linguagens de programação que possuem um operador aritmético exclusivo para esse tipo de operação chamado *modulo* (sem acento por estar grafado no idioma inglês, o qual se pronuncia *módulo*). O operador aritmético de *modulo* é responsável por gerar o resto da divisão de valores inteiros positivos (divisibilidade). Existem linguagens de programação que possuem o operador aritmético de *modulo* e, neste caso, a expressão aritmética apresentada será muito útil.

Apesar de existir em algumas linguagens um operador aritmético exclusivo para o cálculo de divisibilidade (*modulo*), há um problema de sintaxe em seu uso. O operador aritmético em questão é expresso na forma escrita de várias maneiras, o que dificulta sua representação em um diagrama de blocos. Assim sendo, o diagrama de blocos não pode conter comandos exclusivos de uma determinada linguagem de programação, por ser uma ferramenta genérica de representação da lógica e do raciocínio do programador e não do programa. Neste caso, no diagrama de blocos usa-se a expressão aritmética indicada e na codificação do programa usa-se o operador existente na linguagem de programação.

Como exemplo, a tabela de operadores de divisibilidade mostra algumas formas escritas da operação para obter o resto de divisão em algumas linguagens de programação. Lembre-se de não usar nenhum desses operadores aritméticos de divisibilidade ou outros em um diagrama de blocos.

Tabela de operadores de divisibilidade	
Operador	Linguagens
mod	PASCAL, BASIC ¹⁸ , ASP, FORTRAN
modulo	SCHEMA
%	C, C++, C#, Lua, JAVA, PYTHON
\	IEFFEL
\	OCCAN
%%	R
~	J

Normalmente o operador aritmético de divisibilidade é usado de acordo com o formato <resultado> <operador de atribuição> <valor1> <operador de divisibilidade> <valor2>. Por exemplo, a operação $R \leftarrow 5 - 2 * (5 \text{ div } 2)$ em PASCAL será escrita como $R := 5 \text{ mod } 2$; em C será escrita como $R = 5 \% 2$; Cada linguagem de programação tem uma forma particular de expressar a operação.

Mas nem tudo é um mar de flores. Existe também um pequeno problema com o operador de divisão com quociente inteiro **div**. Também é representado de diversas formas, nas mais variadas linguagens de programação, considerando-se também a sua não existência em algumas delas. A tabela de operação de divisão com quociente inteiro mostra alguns exemplos de representação em algumas linguagens da expressão aritmética $R \leftarrow 5 - 2 * (5 \text{ div } 2)$.

Tabela de operação com quociente inteiro	
$R \leftarrow 5 - 2 * (5 \text{ div } 2)$	Linguagem
$R := 5 - 2 * (5 \text{ div } 2)$	PASCAL
$R = 5 - 2 * (5 / 2)$	C, C++
$R = 5 - 2 * \text{INT}(5 / 2)$	BASIC, FORTRAN
$R = 5 - 2 * (5 \setminus 2)$	BASIC (Visual Basic .Net)
$R := 5 - 2 * (5 \text{ DIV } 2)$	MODULA-2
$R = 5 - 2 * \text{math.floor}(5 / 2)$	Lua

A título de demonstração e uso do processo de divisibilidade em um programa de computador considere como exemplo o problema seguinte:

Desenvolver um programa de computador que leia um valor numérico inteiro e faça a apresentação desse valor caso seja divisível por 4 e 5. Não sendo divisível por 4 e 5, o programa deve apresentar a mensagem "Valor não é divisível por 4 e 5".

Para resolver o problema proposto, é necessário, além de usar o operador lógico **.e.**, verificar se a condição do valor lido é ou não divisível por 4 e 5.

Veja a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.25.

¹⁸ Entende-se como linguagem BASIC as versões estruturadas dessa linguagem que foram utilizadas pelas empresas Borland no ambiente de programação *Turbo BASIC* e Microsoft no ambiente *Quick BASIC* e, posteriormente, no ambiente *Visual Basic*.

Entendimento

1. Ler um valor numérico inteiro qualquer (variável N).
2. Calcular o resto da divisão de N por 4, usar a variável R4.
3. Calcular o resto da divisão de N por 5, usar a variável R5.
4. Verificar se as variáveis R4 e R5 possuem o valor de resto igual a zero; se sim, apresentar o valor da variável N; se não, apresentar a mensagem "Valor não é divisível por 4 e 5".

Diagramação

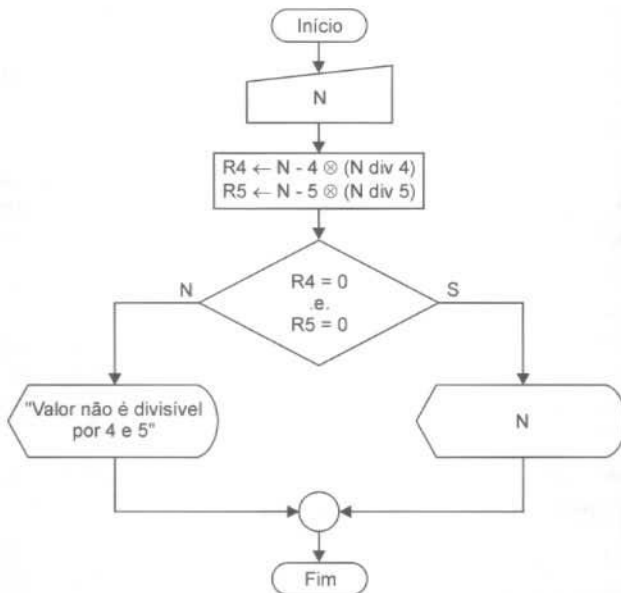


Figura 4.25 - Diagrama de blocos para verificar se N é divisível por 4 e 5.

Codificação

```

programa DIVISIBILIDADE
var
  N, R4, R5 : inteiro
início
  leia N
  R4 ← N - 4 * (N div 4)
  R5 ← N - 5 * (N div 5)
  se (R4 = 0) .e. (R5 = 0) então
    escreva N
  senão
    escreva "Valor não é divisível por 4 e 5"
  fim_se
fim
  
```

Observe o uso das expressões aritméticas $R4 \leftarrow N - 4 * (N \text{ div } 4)$ e $R5 \leftarrow N - 5 * (N \text{ div } 5)$ para a obtenção do valor da divisibilidade, do resto da divisão do valor da variável N pelos divisores 4 e 5 respectivamente. A

instrução de tomada de decisão faz uso da condição $(R4 = 0)$.e. $(R5 = 0)$ que será verdadeira se ambas as condições forem verdadeiras.

As operações condicionais de divisibilidade, do ponto de vista computacional, usam apenas dois operadores relacionais; ou a condição é igual a zero, ou a condição é diferente de zero. Não tente usar outra forma de representação condicional, pois pode incorrer em algum tipo de erro de lógica.

4.8 - Exercício de Aprendizagem

Como descrito no tópico 4.2, um computador pode executar ações de tomada de decisões e isso é conseguido de diversas formas: decisão simples, decisão composta, decisão sequencial, decisão encadeada, decisão por seleção. Uma tomada de decisão pode ocorrer de uma condição ou de mais de uma condição e neste caso são usados operadores lógicos para auxiliar a tarefa. É importante lembrar que condição é o estabelecimento de uma relação lógica entre dois elementos com o auxílio de um operador relacional.

Para demonstrar o tema deste capítulo seguem alguns exemplos de aprendizagem. Observe cuidadosamente cada exemplo apresentado, cada ponto diagramado e codificado, pois os detalhes indicados são importantes para a solução dos exercícios.

1º Exemplo

Elaborar um programa que efetue a entrada de um valor numérico real não negativo diferente de cinco. Em caso afirmativo, o programa deve calcular e exibir o resultado da raiz quadrada do valor fornecido; caso contrário, o programa deve apresentar o resultado da raiz cúbica do valor fornecido. Se o valor fornecido for negativo, o programa não deve executar nenhuma ação, apenas ser encerrado.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 4.26.

Entendimento

Para estabelecer a linha de entendimento adequado deste exemplo de aprendizagem é necessário tomar cuidado com o trecho do problema proposto "valor numérico real não negativo". Observe atentamente a indicação "não negativo". Isso implica no uso do operador lógico de negação **.não.** na tomada de decisão com a condição **.não. ($N < 0$)**. Inicialmente pode até parecer que tanto faz usar a condição **.não. ($N < 0$)** ou a condição **($N >= 0$)**. No entanto, ao usar a segunda opção, **($N >= 0$)**, o programa funciona, mas não cumpre o que fora realmente solicitado, ou seja, o programador comete um erro de requisito. Na verdade, o programa pede para verificar se o número fornecido é um valor *não negativo* e assim deve operar por ser uma questão de lógica e de atendimento ao que é meramente solicitado.

1. Ler um valor numérico real qualquer (variável N).
2. Verificar se o valor fornecido é não negativo e proceder da seguinte forma:
 - 2.1. Se positivo, verificar se é diferente de 5:
 - 2.1.1. Se sim, calcular a raiz quadrada (variável R).
 - 2.1.2. Se não, calcular a raiz cúbica (variável R).
 - 2.1.3. Apresentar o resultado do cálculo da raiz (variável R).
 - 2.2. Se não for positivo, não fazer nada.
3. Encerrar o programa.

Diagramação

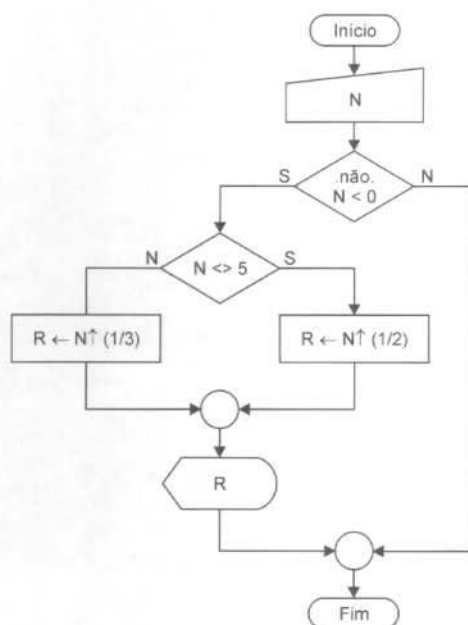


Figura 4.26 - Diagrama de blocos do programa exemplo 1.

Codificação

```

programa EXEMPLO_1
var
  N, R : real
inicio
  leia N
  se .não. (N < 0) então
    se (N <> 5) então
      R ← N ↑ (1 / 2)
    senão
      R ← N ↑ (1 / 3)
  fim_se
  escreva R
fim_se
fim
  
```

2º Exemplo

Elaborar um programa que efetue a entrada dos valores de medida de três pesos auferidos de forma aleatória. O programa deve mostrar o maior peso fornecido.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 4.27.

Entendimento

Para entender este problema, basta pegar um dos pesos e comparar com os demais. Sendo o peso menor que o comparado, deve ser descartado e ser pego o peso comparado que é maior. Este procedimento deve ser repetido quantas vezes for necessário para obter o maior peso no final.

1. Ler o primeiro peso (variável A).
2. Ler o primeiro peso (variável B).
3. Ler o primeiro peso (variável C).
4. Criar uma variável auxiliar (variável X).
5. Assumir que a variável X possui o valor da variável A.
6. Verificar se a variável X possui menor peso que a variável B:
 - 6.1. Se sim, fazer X assumir o valor da variável B.
 - 6.2. Se não, manter o valor da variável X.

7. Verificar se a variável X possui menor peso que a variável C:
 - 7.1. Se sim, fazer X assumir o valor da variável C.
 - 7.2. Se não, manter o valor da variável X.
8. Apresentar o valor da variável X que terá o maior valor de peso.

Diagramação

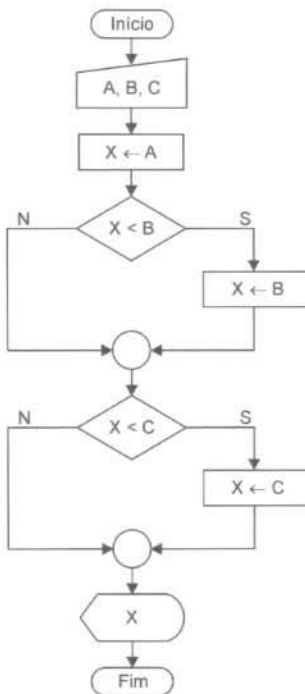


Figura 4.27 - Diagrama de blocos do programa exemplo 2.

Codificação

```

programa EXEMPLO_2
var
  A, B, C, X : inteiro
inicio
  leia A, B, C
  X ← A
  se (X < B) então
    X ← B
  fim_se
  se (X < C) então
    X ← C
  fim_se
  escreva X
fim
  
```

3º Exemplo

Elaborar um programa que leia três valores para os lados de um triângulo, considerando lados como A, B e C. Verificar se os lados fornecidos formam um triângulo, e se for esta condição verdadeira, deve ser indicado o tipo de triângulo formado: isósceles, escaleno ou equilátero. Veja o algoritmo, diagrama de blocos e a codificação em português estruturado, prestando atenção nos operadores lógicos.

Observe a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 4.28.

Entendimento

Para estabelecer esse algoritmo é necessário, em primeiro lugar, saber o que realmente é um triângulo. Caso não saiba, não conseguirá resolver o problema. Triângulo é uma forma geométrica (polígono) composta de três lados, e o valor de cada lado deve ser menor que a soma dos outros dois lados. Esta definição é uma regra (uma condição) e deve ser plenamente considerada. Assim sendo, é um triângulo quando $A < B + C$, quando $B < A + C$ e quando $C < A + B$, considerando como lados as variáveis A, B e C.

Tendo certeza de que os valores informados para os três lados formam um triângulo, deve-se então analisar os valores fornecidos para estabelecer o tipo de triângulo que será formado: isósceles, escaleno ou equilátero.

Um triângulo é isósceles quando possui dois lados iguais e um diferente, sendo $A=B$ ou $A=C$ ou $B=C$; é escaleno quando possui todos os lados diferentes, sendo $A < B$ e $B < C$ e é equilátero quando possui todos os lados iguais, sendo $A=B$ e $B=C$.

1. Ler três valores para os lados de um triângulo (variáveis A, B e C).
2. Verificar se cada lado é menor que a soma dos outros dois lados. Se sim, saber se $A=B$ e se $B=C$; sendo verdade, o triângulo é equilátero. Se não, verificar $A=B$ ou se $A=C$ ou se $B=C$; sendo verdade, o triângulo é isósceles; caso contrário, o triângulo é escaleno.
3. Caso os lados fornecidos não caracterizem um triângulo, avisar a ocorrência.

Diagramação

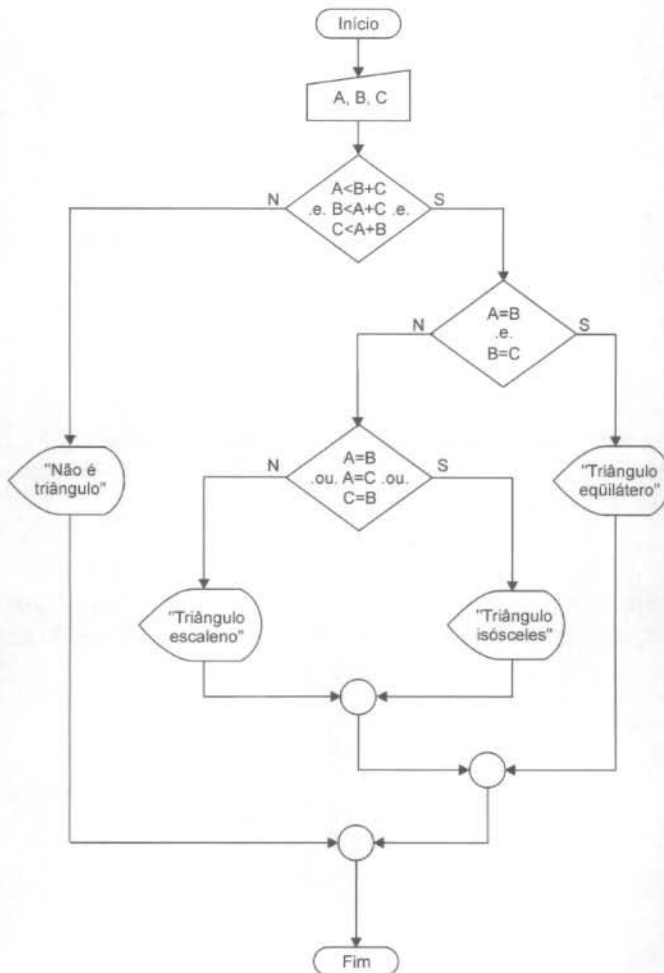


Figura 4.28 - Diagrama de blocos do programa exemplo 3.

Codificação

```

programa TRIÂNGULO
var
  A, B, C : real
início
  leia A, B, C
  se (A < B + C) .e. (B < A + C) .e. (C < A + B) então
    se (A = B) .e. (B = C) então
      escreva "Triângulo Equilátero"
    senão
      se (A = B) .ou. (A = C) .ou. (C = B) então
        escreva "Triângulo Isósceles"
      senão
        escreva "Triângulo Escaleno"
    fim_se
  fim_se
senão
  escreva "As medidas não formam um triângulo"
fim_se
fim

```

4º Exemplo

Elaborar um programa que leia um valor inteiro qualquer e apresente esse valor somente se for divisível por 2 ou somente se for divisível 3. Caso contrário, não faça nada. Em hipótese nenhuma esse valor pode ser apresentado caso seja divisível por 2 e/ou 3.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 4.29.

Entendimento

Para estabelecer esse algoritmo é necessário, após a entrada do valor, efetivar os cálculos de divisibilidade por 2 e também por 3. Em seguida é necessário usar o operador lógico de disjunção exclusiva **.xou.** para conseguir atender a restrição do que é pedido.

1. Efetuar a leitura do valor inteiro qualquer (variável N).
2. Calcular a divisibilidade da variável N por 2 (armazenar resultado na variável R2).
3. Calcular a divisibilidade da variável N por 3 (armazenar resultado na variável R3).
4. Verificar se a variável R2 é igual a zero ou exclusivamente se a variável R3 é igual a zero:
 - 4.1. Se sim, apresentar o conteúdo na variável N.
 - 4.2. Se não, abandonar o programa.

Diagramação

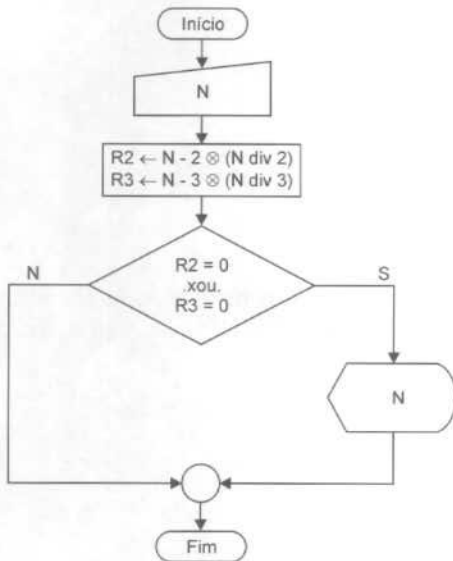


Figura 4.29 - Diagrama de blocos do programa exemplo 4.

Codificação

```

programa DIVISIVEL_OU_POR_2_OU_POR_3
var
  N, R2, R3 : inteiro
inicio
  leia N
  R2 ← N - 2 * (N div 2)
  R3 ← N - 3 * (N div 3)
  se (R2 = 0) .ou. (R3 = 0) então
    escreva N
  fim_se
fim
    
```

4.9 - Exercícios de Fixação

- Determine o resultado lógico das expressões mencionadas, assinalando se são verdadeiras ou falsas. Considere para as respostas os seguintes valores: X = 1, A = 3, B = 5, C = 8 e D = 7.
 - .não. (X > 3)
Verdadeiro () Falso ()
 - (X < 1) .e. .não. (B > D)
Verdadeiro () Falso ()
 - .não. (D < 0) .e. (C > 5)
Verdadeiro () Falso ()
 - .não. (X > 3) .ou. (C < 7)
Verdadeiro () Falso ()
 - (A > B) .ou. (C > B)
Verdadeiro () Falso ()
 - (X >= 2)
Verdadeiro () Falso ()

- g) $(X < 1) .e. (B \geq D)$
 Verdadeiro () Falso ()
- h) $(D < 0) .ou. (C > 5)$
 Verdadeiro () Falso ()
- i) $.n\tilde{a}o. (D > 3) .ou. .n\tilde{a}o. (B < 7)$
 Verdadeiro () Falso ()
- j) $(A > B) .ou. .n\tilde{a}o. (C > B)$
 Verdadeiro () Falso ()

2. Indique na linha de resposta a expressão aritmética a ser calculada a partir da tomada de decisão composta em análise. Considere os seguintes valores: A=2, B=3, C=5 e D=9. Não é necessário calcular os valores da variável X.

a) Resposta: _____

```
se .n\tilde{a}o. (D > 5) ent\~{a}o
  X ← (A + B) * D
sen\~{a}o
  X ← (A - B) / C
fim_se
escreva X
```

b) Resposta: _____

```
se (A > 2) .e. (B < 7) ent\~{a}o
  X ← (A + 2) * (B - 2)
sen\~{a}o
  X ← (A + B) / D * (C + D)
fim_se
escreva X
```

c) Resposta: _____

```
se (A = 2) .ou. (B < 7) ent\~{a}o
  X ← (A + 2) * (B - 2)
sen\~{a}o
  X ← (A + B) / D * (C + D)
fim_se
escreva X
```

d) Resposta: _____

```
se (A > 2) .ou. .n\tilde{a}o. (B < 7) ent\~{a}o
  X ← A + B - 2
sen\~{a}o
  X ← A - B
fim_se
escreva X
```


e) Resposta: _____

```

se .não. (A > 2) .ou. .não. (B < 7) então
  X ← A + B
senão
  X ← A / B
fim_se
escreva X

```

f) Resposta: _____

```

se .não. (A > 3) .e. .não. (B < 5) então
  X ← A + D
senão
  X ← D / B
fim_se
escreva X

```

g) Resposta: _____

```

se (C >= 2) .e. (B <= 7) então
  X ← (A + D) / 2
senão
  X ← D * C
fim_se
escreva X

```

h) Resposta: _____

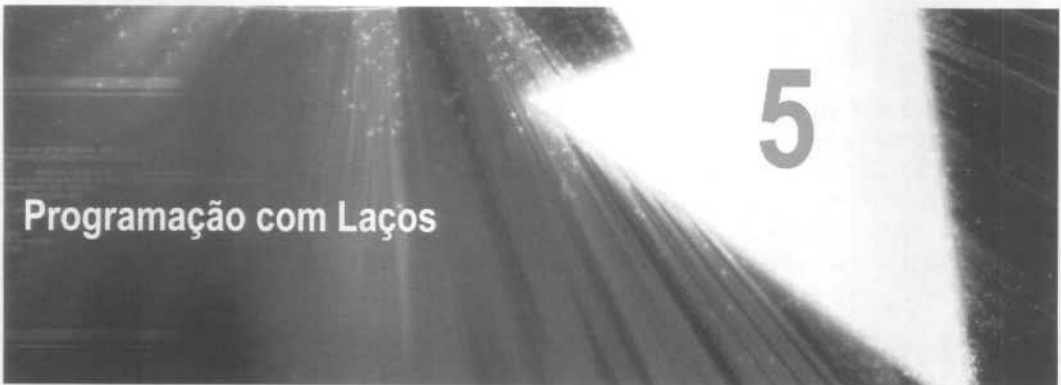
```

se (A >= 2) .ou. (C <= 1) então
  X ← (A + D) / 2
senão
  X ← D * C
fim_se
escreva X

```

3. Desenvolva os entendimentos, diagrama de blocos e código em português estruturado dos seguintes problemas computacionais:
- Efetuar a leitura de dois valores numéricos inteiros representados pelas variáveis A e B e apresentar o resultado da diferença do maior valor pelo menor valor.
 - Efetuar a leitura de um valor numérico inteiro positivo ou negativo representado pela variável N e apresentar o valor lido como sendo positivo. Dica: se o valor lido for menor que zero, ele deve ser multiplicado por -1.
 - Realizar a leitura dos valores de quatro notas escolares bimestrais de um aluno representadas pelas variáveis N1, N2, N3 e N4. Calcular a média aritmética (variável MD) desse aluno e apresentar a mensagem "Aprovado" se a média obtida for maior ou igual a 5; caso contrário, apresentar a mensagem "Reprovado". Informar também, após a apresentação das mensagens, o valor da média obtida pelo aluno.

- d) Ler os valores de quatro notas escolares bimestrais de um aluno representadas pelas variáveis N1, N2, N3 e N4. Calcular a média aritmética (variável MD1) desse aluno e apresentar a mensagem "Aprovado" se a média obtida for maior ou igual a 7; caso contrário, o programa deve solicitar a quinta nota (nota de exame, representada pela variável NE) do aluno e calcular uma nova média aritmética (variável MD2) entre a nota de exame e a primeira média aritmética. Se o valor da nova média for maior ou igual a cinco, apresentar a mensagem "Aprovado em exame"; caso contrário, apresentar a mensagem "Reprovado". Informar também, após a apresentação das mensagens, o valor da média obtida pelo aluno.
- e) Efetuar a leitura de três valores numéricos (representados pelas variáveis A, B e C) e processar o cálculo da equação completa de segundo grau, utilizando a fórmula de Bhaskara (considerar para a solução do problema todas as possíveis condições para delta: $\text{delta} < 0$ – não há solução real, $\text{delta} > 0$ – há duas soluções reais e diferentes e $\text{delta} = 0$ – há apenas uma solução real). Lembre-se de que é completa a equação de segundo grau que possui todos os coeficientes A, B e C diferentes de zero. O programa deve apresentar respostas para todas as condições estabelecidas para delta.
- f) Ler três valores inteiros representados pelas variáveis A, B e C e apresentar os valores lidos dispostos em ordem crescente. Dica: utilizar tomada de decisão sequencial e as ideias trabalhadas nos exercícios "g" (propriedade distributiva) e "f" (troca de valores) do capítulo 3.
- g) Fazer a leitura de quatro valores numéricos inteiros representados pelas variáveis A, B, C e D. Apresentar apenas os valores que sejam divisíveis por 2 e 3.
- h) Efetuar a leitura de quatro valores numéricos inteiros representados pelas variáveis A, B, C e D. Apresentar apenas os valores que sejam divisíveis por 2 ou 3.
- i) Ler cinco valores numéricos inteiros (variáveis A, B, C, D e E), identificar e apresentar o maior e o menor valores informados. Não execute a ordenação dos valores como no exercício "f".
- j) Ler um valor numérico inteiro e apresentar uma mensagem informando se o valor fornecido é par ou ímpar.
- k) Efetuar a leitura de um valor numérico inteiro que esteja na faixa de valores de 1 até 9. O programa deve apresentar a mensagem "O valor está na faixa permitida", caso o valor informado esteja entre 1 e 9. Se o valor estiver fora da faixa, o programa deve apresentar a mensagem "O valor está fora da faixa permitida".
- l) Fazer a leitura de um valor numérico inteiro qualquer e apresentá-lo caso **não seja maior que 3**. Dica: para a solução deste problema utilize apenas o operador lógico de negação.
- m) Efetuar a leitura de um nome (variável NOME) e o sexo (variável SEXO) de uma pessoa e apresentar como saída uma das seguintes mensagens: "Ilmo. Sr.", caso seja informado o sexo masculino (utilizar como valor o caractere "M"), ou "Ilma. Sra.", caso seja informado o sexo feminino (utilizar como valor o caractere "F"). Após a mensagem de saudação, apresentar o nome informado. O programa deve, após a entrada do sexo, verificar primeiramente se o sexo fornecido é realmente válido, ou seja, se é igual a "M" ou a "F". Não sendo essa condição verdadeira, o programa deve apresentar a mensagem "Sexo informado inválido".
- n) Efetuar a leitura de três valores inteiros desconhecidos representados pelas variáveis A, B e C. Somar os valores fornecidos e apresentar o resultado somente se for maior ou igual a 100.



Anteriormente foi estudado como trabalhar com tomadas de decisão nas mais variadas formas. Nesta etapa já é possível escrever a estrutura de programas mais sofisticados. Este capítulo amplia o foco do estudo, apresentando uma técnica de programação que possibilita repetir um trecho de programa, sem que seja necessário escrevê-lo exatamente o número de vezes que se deseja de fato executar. Mostra as técnicas de laços de repetição interativos e iterativos, como laço de repetição condicional pré-teste (verdadeiro e falso), laço de repetição condicional pós-teste (verdadeiro e falso), laço de repetição condicional seletivo e por fim o laço de repetição incondicional.

5.1 - Ser Programador

Antes de prosseguir, cabe apresentar mais alguns detalhes sobre a ética e o comportamento do programador de computador. Ser um programador exige alto grau de atenção e cuidado no trabalho executado. Um médico-cirurgião desatento pode matar um paciente numa cirurgia; um programador desatento pode "matar" uma empresa. Na tarefa de programar computadores o programador pode correr três riscos, descritos em seguida.

- ▶ Erro de sintaxe, cometido quando se escrevem comandos e/ou instruções de forma incorreta, sem respeitar as regras gramaticais da linguagem de programação em uso. Esse tipo de erro denota que o programador não possui experiência na escrita de programas naquela linguagem. Isso não significa que não tenha boa lógica, apenas mostra que o programador não sabe escrever o código na linguagem em uso. Para solucionar este problema é necessário aprender a escrever os códigos da linguagem pretendida. O erro de sintaxe é considerado de baixa gravidade, pois é fácil de ser corrigido.
- ▶ Erro de requisito, acontece quando o programador não atende o que é solicitado. Esse erro denota que o programador não sabe ou não quer ouvir o que lhe é pedido. Para solucionar é necessário ser mais atento. O erro de requisito é considerado de média gravidade, pois é possível corrigi-lo com certa facilidade.
- ▶ Erro de lógica, ocorre quando o programador não consegue entender e atender o que de fato é necessário fazer com o programa. Denota que o programador não soube pensar a programação de computadores. Para solucionar este problema é necessário mudar o "pensar", o que pode ser tarefa muito trabalhosa. O erro de lógica é considerado de alta gravidade, pois é difícil de ser corrigido, exigindo do programador muita disciplina.

Um programador cuidadoso, que goste de programar, é um profissional que evita correr riscos desnecessários, principalmente os mais comuns, que sabe, por meio de bom senso, ser perseverante, disciplinado e, acima de tudo, humilde em seu trabalho.

Uma forma de evitar erros, principalmente de requisitos e de lógica, é a construção de algoritmos. Entre as formas de algoritmos que podem ser usadas na área da programação de computadores, a gráfica é um instrumento que auxilia os olhos e a mente do programador a enxergar e verificar a sua própria linha de raciocínio, ou de outros programadores. Vale ressaltar as palavras do Pe. John Venn (autor dos gráficos utilizados no tópico 4.6 sobre o uso de operadores lógicos): "*os diagramas servem para auxiliar os olhos e a mente graças à natureza intuitiva de seu próprio testemunho*".

5.2 - Laços ou Malhas de Repetição (Loopings ou Loops)

A técnica denominada laços de repetição, que também pode ser referenciada como *malhas de repetição* ou pelos seus termos análogos em inglês *loopings* ou *loops*, é uma estrutura de programação que facilita repetir determinados trechos de código. Essa técnica reduz o trabalho de programação, principalmente quando é preciso repetir várias vezes alguma ação importante do programa.

Os laços de repetição podem ser classificados em duas formas, sendo *laços de repetição interativa* ou *laços de repetição iterativa*. São *interativos* quando necessitam da intervenção de um usuário do programa para repetir a próxima ação um indeterminado número de vezes, são *laços iterativos* quando executam as repetições previstas de forma automática um número de vezes.

Para executar laços de repetição são encontradas nas linguagens de programação instruções que podem ser escritas de até seis formas diferentes. Isso não significa que uma linguagem de programação precise ter o conjunto completo dos seis tipos de laços de repetição. A maior parte das linguagens de programação oferece em média três formas de laços de repetição. Podem existir linguagens que possuem até menos formas de execução de laços.

Dos seis tipos de laços de repetição existentes, cinco são condicionais e um é incondicional. Os laços condicionais podem ser classificados em três categorias, sendo *controle condicional pré-teste*, *controle condicional pós-teste* e *controle condicional seletivo*, desenhados em um diagrama de blocos com o símbolo *decision*. Nessas estruturas não se usa o símbolo *connector*. Já o laço incondicional é desenhado em um diagrama de blocos com o símbolo *preparation*.

A maior parte das linguagens de programação possui três tipos de laços de repetição, sendo oferecidos normalmente um laço com controle condicional pré-teste, um laço com controle condicional pós-teste e um laço incondicional, pois em tese não há necessidade de que uma linguagem de programação possua os seis tipos de laços de repetição. Algumas linguagens oferecem ainda a alternativa de execução de laços de repetição condicionais seletivos.

Já que o assunto são laços de repetição, existe uma certa tradição na construção dos laços, principalmente no que tange ao uso de laços iterativos. Trata-se do nome da variável de controle que executará o laço e permitirá o seu controle. É hábito, entre a maior parte dos programadores, usar essa variável com o nome *I*, de iteração em português ou *iteration* em inglês. Respeitando essa tradição, a maior parte dos laços iterativos apresentados nesta obra usa *I* como variável de controle, mas como é tradição e não uma regra, também serão escritos laços iterativos com outros nomes.

5.3 - Laço de Repetição Condicional Pré-Teste

Os laços de repetição condicionais com verificação pré-teste têm por finalidade executar as instruções subordinadas de um bloco adjacente após conferir a validade do resultado lógico de uma condição estabelecida. Sendo o resultado lógico da condição válido, são executadas as instruções subordinadas do bloco adjacente. No momento em que o resultado lógico da condição não é mais válido, o laço de repetição é automaticamente encerrado.

Os laços de repetição com controle condicional pré-teste apresentam-se de duas formas, sendo um laço de repetição para condição verdadeira e outro para condição falsa.

5.3.1 - Controle Condicional Verdadeiro

O laço de repetição com controle condicional pré-teste verdadeiro executa as instruções subordinadas de um bloco adjacente no período em que o resultado lógico da condição permanece verdadeiro. No momento em que o resultado lógico da condição se tornar falso, a execução do laço é automaticamente encerrada.

Observe na Figura 5.1 um diagrama de blocos com os rótulos **S** e **N** para a execução de um laço de repetição com controle condicional pré-teste verdadeiro com base na **CONDIÇÃO** do símbolo *decision*. Note uma linha de fluxo que pende do símbolo *decision* com uma seta apontando para baixo sinalizada com o rótulo **S** e uma segunda linha de fluxo saindo do símbolo *decision* e sinalizada com o rótulo **N**, indicando a saída do laço de repetição quando a condição se tornar falsa. Além dessas duas linhas de fluxo há também uma terceira que mostra uma ação de retorno para o próprio símbolo *decision*, indicando a ação real do laço.

O laço de repetição com controle condicional pré-teste verdadeiro, do ponto de vista da codificação em português estruturado, utiliza os comandos **enquanto**, **faça** e **fim_enquanto** para realizar a construção da instrução **enquanto...faça/fim_enquanto**. Nesse trecho de instrução são executadas as instruções subordinadas ao bloco adjacente (entre os comandos **enquanto...faça** e **fim_enquanto**) no período em que o resultado lógico da condição permanecer verdadeiro entre os comandos **enquanto** e **faça**. No momento em que o resultado lógico for falso, são executadas as eventuais instruções que estiverem após o comando **fim_enquanto**. Observe a estrutura sintática seguinte:

```
enquanto (<condição>) faça
    [instruções executadas durante o período em que a condição é verdadeira]
fim_enquanto
```

Atente para o detalhe da *indentação* para sinalizar as instruções subordinadas ao bloco adjacente da instrução **enquanto...faça/fim_enquanto**. A indicação **<condição>** entre parênteses deve ser substituída pela expressão lógica da condição a ser utilizada. Os trechos sinalizados entre colchetes representam as instruções a serem executadas pelo programa.

A título de ilustração do laço de repetição com controle condicional pré-teste verdadeiro em um contexto operacional, considere os problemas a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação. Os programas demonstram os laços de repetição nas formas iterativa e interativa.

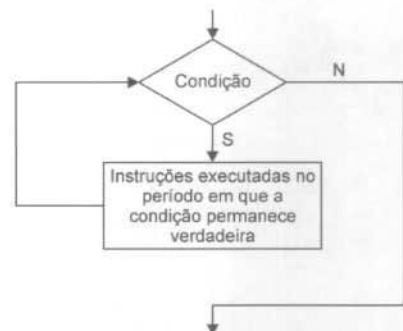


Figura 5.1 - Estrutura de laço de repetição com controle condicional pré-teste verdadeiro.

Exemplo com laço iterativo

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida processe o cálculo do valor de entrada, multiplicando-o por 3 e apresentando seu resultado. Proceder à execução dos passos anteriores cinco vezes.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.2.

Diagramação

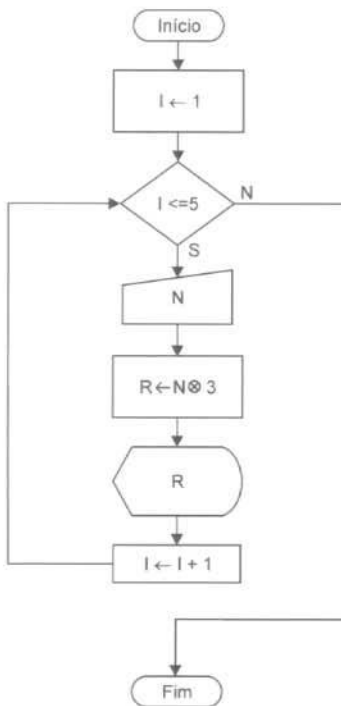


Figura 5.2 - Exemplo da instrução enquanto...faça/fim_enquanto iterativa.

Codificação

```

programa LAÇO_PRÉ_TESTE_VERDADEIRO_VA
var
  I, N, R : inteiro
inicio
  I ← 1
  enquanto (I <= 5) faça
    leia N
    R ← N * 3
    escreva R
    I ← I + 1
  fim_enquanto
fim
  
```

Entendimento

No problema exposto nada impede que se faça a entrada do valor, execute o processamento da multiplicação e em seguida apresente o resultado cinco vezes, escrevendo cinco entradas, com cinco processamentos e cinco saídas. O problema seria mais difícil de resolver se tivesse de executar a mesma tarefa mil vezes. Para solucionar este problema existem as estruturas de laços de repetição. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para servir como contador com valor inicial 1 (variável I).
2. Enquanto o valor da variável contador for menor ou igual a 5, executar os passos 3, 4 e 5; caso contrário, desviar a execução do programa para o passo 8.
3. Ler um valor inteiro qualquer (variável N).

4. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R .
5. Apresentar o valor calculado que está armazenado na variável R .
6. Acrescentar o valor 1 ao valor existente da variável I , definida no passo 1.
7. Retornar a execução do programa ao passo 2.
8. Encerrar a execução do programa.

Além das variáveis N e R , foi necessário criar uma terceira variável (I) para controlar o número de vezes que o trecho de programa deve ser executado.

Logo após o início do programa, a variável contador é atribuída com o valor 1 ($I \leftarrow 1$). Em seguida, a instrução **enquanto** ($I \leq 5$) **faça** verifica a condição estabelecida e se o resultado lógico da condição é verdadeiro. Sendo o valor da variável I igual a 1, a condição do laço de repetição é verdadeira, e por esta razão deve processar as instruções subordinadas ao bloco adjacente definido até o limite do comando **fim_enquanto**.

Sendo assim, tem início a sequência de instruções subordinadas ao bloco adjacente que estão entre as instruções **enquanto...faça** e **fim_enquanto**. Depois da primeira leitura, cálculo e apresentação do valor calculado, o programa encontra a linha $I \leftarrow I + 1$. A variável I possui neste momento o valor 1 que, somado a 1, passa a ter o valor 2, ou seja, $I \leftarrow I + 1$, sendo assim $I \leftarrow 1 + 1$ que resulta $I = 2$.

Agora que a variável I possui o valor 2, o processamento do programa volta para a instrução **enquanto** ($I \leq 5$) **faça**, uma vez que o valor da variável I é menor ou igual a 5. É executada a rotina de instruções subordinadas ao bloco adjacente. Ao final dessa sequência a variável I passa a ter o valor 3. Desta forma, o programa processa novamente a rotina de instruções, passando o valor de I para 4, que será verificado, e sendo menor ou igual a 5, será executada mais uma vez a mesma rotina de instruções.

Neste ponto, a variável I passa a ter o valor 5. A instrução **enquanto** ($I \leq 5$) **faça** verifica o valor da variável I que é 5 com a condição $I \leq 5$. Veja que 5 não é menor que 5, mas é igual. Sendo essa condição verdadeira, a rotina deve ser executada mais uma vez. Neste momento, o valor da variável I passa a ser 6, que fornece para a instrução **enquanto** um resultado lógico da condição como falso. Por conseguinte, desvia o fluxo de processamento do programa para a primeira instrução após a instrução **fim_enquanto**, que no caso é **fim**, encerrando o programa.

Exemplo com laço interativo

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor entrado, multiplicando-o por 3 e apresentando seu resultado. Ao final da apresentação do resultado o programa deve perguntar ao usuário se ele deseja novo cálculo. Se a resposta for sim, deve executar novamente as instruções subordinadas ao bloco adjacente. Se a resposta for não, o programa deve parar a execução.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 5.3.

Entendimento

No problema exposto é necessário definir um laço que vai executar a vontade do usuário. Assim sendo, não há possibilidade de saber quantas vezes o laço será executado. O número de vezes vai depender da vontade do usuário, por isso esse tipo de laço de repetição chama-se interativo. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para ser utilizada como resposta (variável RESP).
2. Enquanto o valor da variável RESP for igual a "S", executar os passos 3, 4 e 5; caso contrário, desviar a execução do programa para o passo 8.
3. Ler um valor inteiro qualquer (variável N).

4. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
5. Apresentar o valor calculado que está armazenado na variável R.
6. Perguntar para o usuário se ele deseja continuar a execução do programa.
7. Retornar a execução do programa ao passo 2.
8. Encerrar o programa.

Diagramação

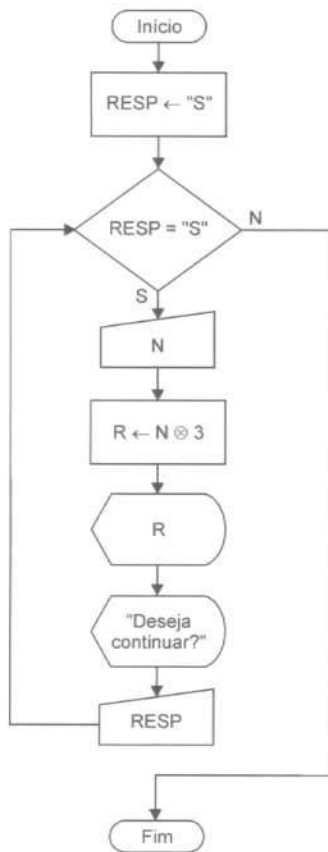


Figura 5.3 - Exemplo da instrução enquanto...faça/fim_enquanto iterativa.

Codificação

```

programa LAÇO_PRÉ_TESTE_VERDADEIRO_VB
var
  N, R : inteiro
  RESP : caractere
inicio
  RESP ← "S"
  enquanto (RESP = "S") faça
    leia N
    R ← N * 3
    escreva R
    escreva "Deseja continuar?"
    leia RESP
  fim_enquanto
fim
  
```

Nesta versão o contador foi substituído pela variável *RESP*, que enquanto possuir valor igual a "S", executa a sequência de instruções subordinadas ao bloco adjacente dos comandos **enquanto...faça** e **fim_enquanto**. Neste caso, o número de vezes que a rotina se repete será controlado pelo usuário e será encerrada somente quando alguma informação diferente de "S" for fornecida para a variável *RESP*, enviando o fluxo do programa para a primeira instrução após a instrução **fim_enquanto**.

O estilo de laço de repetição com controle condicional pré-teste verdadeiro é encontrado, por exemplo, nas linguagens PASCAL, BASIC, C, C++, JAVA, ADA, entre outras. Esta é a forma mais comum de execução de laço de repetição com controle condicional pré-teste encontrada nas linguagens de programação.

5.3.2 - Controle Condicional Falso

O laço de repetição com controle condicional pré-teste falso executa as instruções subordinadas de um bloco adjacente no período em que o resultado lógico da condição permanece falso. No momento em que o resultado lógico da condição se tornar verdadeiro, a execução do laço é automaticamente encerrada.

Observe na Figura 5.4 um diagrama de blocos com os rótulos **N** e **S** para a execução de um laço de repetição com controle condicional pré-teste falso com base na **CONDIÇÃO** do símbolo *decision*. Note uma linha de fluxo pendendo do símbolo *decision* com uma seta apontando para baixo sinalizada com o rótulo **N** e uma segunda linha de fluxo saindo do símbolo *decision* e sinalizada com o rótulo **S**, indicando a saída do laço de repetição quando a condição se tornar verdadeira. Além dessas duas linhas de fluxo há também uma terceira que mostra uma ação de retorno para o próprio símbolo *decision*, indicando a ação real do laço.

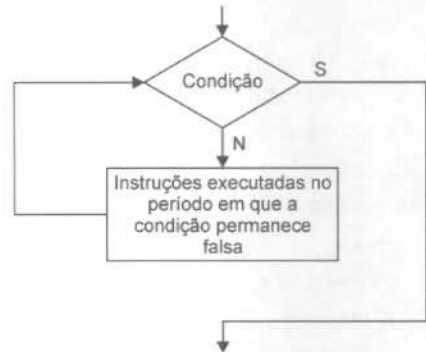


Figura 5.4 - Estrutura de laço de repetição com controle condicional pré-teste falso.

O laço de repetição com controle condicional pré-teste falso, do ponto de vista da codificação em português estruturado, utiliza os comandos **até_seja**, **efetue** e **fim_até_seja** na construção de sua instrução. Neste trecho são executadas as instruções subordinadas ao bloco adjacente (entre os comandos **até_seja...efetue** e **fim_até_seja**) no período em que o resultado lógico da condição permanecer falso entre os comandos **até_seja** e **efetue**. No momento em que o resultado lógico for verdadeiro, serão executadas as eventuais instruções que estiverem após o comando **fim_até_seja**. Observe a estrutura sintática seguinte:

```

até_seja (<condição>) efetue
  [instruções executadas durante o período em que a condição é falsa]
fim_até_seja
  
```

Atente para o detalhe da *indentação* para sinalizar as instruções subordinadas ao bloco adjacente da instrução **até_seja...efetue/fim_até_seja**. A indicação **<condição>** entre parênteses deve ser substituída pela expressão lógica da condição a ser utilizada. Os trechos sinalizados entre colchetes representam as instruções a serem executadas pelo programa.

A título de ilustração do laço de repetição com controle condicional pré-teste falso em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação. Os programas demonstram os laços de repetição nas formas iterativa e interativa.

Exemplo com laço iterativo

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor entrado, multiplicando-o por 3 e apresentando seu resultado. Proceder à execução dos passos anteriores cinco vezes.

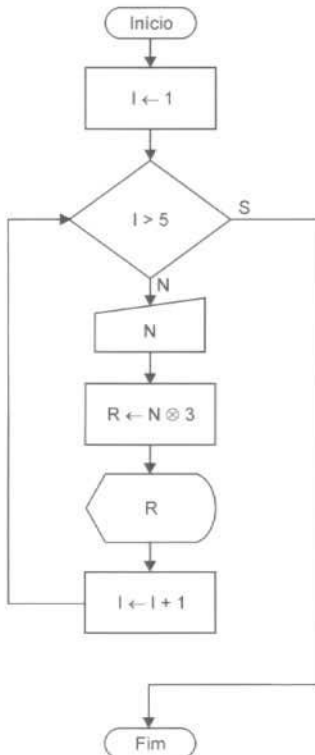
Observe a seguir a descrição das etapas básicas do problema e a representação das ações do programa na Figura 5.5.

Entendimento

O problema exposto é o mesmo do tópico anterior. A diferença é a forma de construção do laço de repetição. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para servir como contador com valor inicial 1 (variável I).
2. Até que o valor da variável contador seja maior que 5, executar os passos 3, 4 e 5; caso contrário, desviar a execução do programa para o passo 8.
3. Ler um valor inteiro qualquer (variável N).
4. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
5. Apresentar o valor calculado que está armazenado na variável R.
6. Acrescentar o valor 1 ao valor existente da variável I, definida no passo 1.
7. Retornar a execução do programa ao passo 2.
8. Encerrar o programa.

Diagramação



Codificação

```

programa LAÇO_PRÉ_TESTE_FALSO_VA
var
  I, N, R : inteiro
inicio
  I ← 1
  até_seja (I > 5) efetue
    leia N
    R ← N * 3
    escreva R
    I ← I + 1
  fim_até_seja
fim
  
```

Figura 5.5 - Exemplo da instrução durante...faça/fim_durante iterativa.

A primeira ação estabelecida é colocar a variável contador com o valor 1 ($I \leftarrow 1$). Em seguida, a instrução **até_seja** ($I > 5$) **efetue** verifica a condição estabelecida e se o resultado lógico da condição é falso. Sendo o valor da variável I igual a 1, a condição do laço de repetição é falsa, e por esta razão deve processar as instruções subordinadas ao bloco adjacente até o limite do comando **fim_até_seja**. Assim sendo, inicia-se a sequência de instruções subordinadas ao bloco adjacente que estão entre as instruções **até_seja...efetue** e **fim_até_seja**. Depois da primeira leitura, do cálculo e da apresentação do valor calculado, o programa encontra a linha $I \leftarrow I + 1$. A variável I possui neste momento o valor 1 que, somado a 1, passa a ter o valor 2.

Agora que a variável I possui o valor 2, o processamento do programa volta para a instrução **até_seja** ($I > 5$) **efetue**, uma vez que o valor da variável I não é maior que 5. É executada a rotina de instruções subordinadas do bloco adjacente. Ao final dessa sequência a variável I passa a possuir o valor 3. O programa processa novamente a rotina de instruções, passando o valor de I para 4, que será verificado, e sendo menor ou igual a 5, será executada mais uma vez a mesma rotina de instruções.

Neste ponto, a variável I passa a ter o valor 5. A instrução **até_seja** ($I > 5$) **efetue** verifica o valor da variável I que é 5 com a condição $I > 5$. Veja que 5 não é maior que 5. Sendo essa condição falsa, deve a rotina ser executada mais uma vez. Neste momento, o valor da variável I passa a ser 6, que para a instrução **até_seja** é um resultado lógico da condição como verdadeiro. Por conseguinte, desvia o fluxo de processamento do programa para a primeira instrução após a instrução **fim_até_seja**, que no caso é **fim**, encerrando o programa.

Exemplo com laço iterativo

Elaborar um programa que entre um valor numérico inteiro qualquer, em seguida calcule o valor de entrada, multiplicando-o por 3 e apresentando seu resultado. Ao final da apresentação do resultado o programa deve perguntar ao usuário se ele deseja novo cálculo. Se a resposta for sim, deve executar novamente as instruções subordinadas ao bloco adjacente. Se a resposta for não, o programa deve parar a execução.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.6.

Entendimento

No problema exposto é necessário definir um laço que vai executar a vontade do usuário. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para ser utilizada como resposta (variável RESP).
2. Até que o valor da variável RESP seja diferente de "S", executar os passos 3, 4 e 5; caso contrário, desviar a execução do programa para o passo 8.
3. Ler um valor inteiro qualquer (variável N).
4. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
5. Apresentar o valor calculado que está armazenado na variável R.
6. Perguntar para o usuário se ele deseja continuar a execução do programa.
7. Retomar a execução do programa ao passo 2.
8. Encerrar o programa.

Diagramação

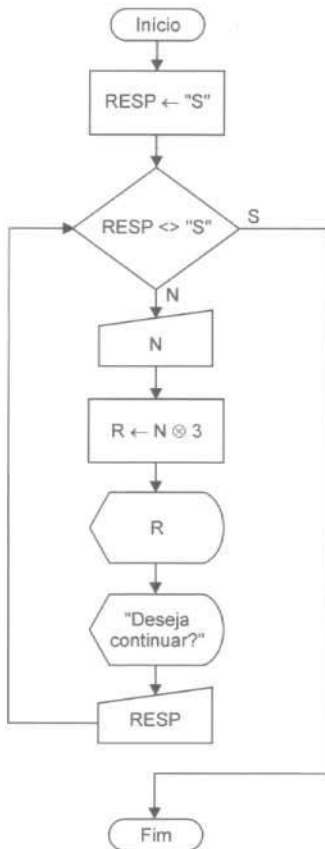


Figura 5.6 - Exemplo da instrução até_seja...faça/fim_até_seja interativa.

Codificação

```

programa LAÇO_PRÉ_TESTE_FALSO_VB
var
  N, R : inteiro
  RESP : caractere
inicio
  RESP ← "S"
  até_seja (RESP <> "S") efetue
    leia N
    R ← N * 3
    escreva R
    escreva "Deseja continuar?"
    leia RESP
  fim_até_seja
fim
  
```

Nesta versão o contador foi substituído pela variável *RESP*, e até que o valor seja diferente de "S", executará a sequência de instruções subordinadas ao bloco adjacente dos comandos **até_seja...efetue** e **fim_até_seja**. Neste caso, o número de vezes que a rotina se repete será controlado pelo usuário e encerrada somente quando alguma informação diferente de "S" for fornecida para a variável *RESP*, enviando o fluxo de execução do programa para a primeira instrução após a instrução **fim_até_seja**.

O laço de repetição com controle condicional pré-teste falso é encontrado, por exemplo, na linguagem BASIC. Essa forma de execução de laço de repetição com controle condicional pré-teste é muito rara nas linguagens de programação.

5.4 - Laço de Repetição Condicional Pós-Teste

O laço de repetição condicional com verificação pós-teste tem por finalidade executar pelo menos uma vez as instruções subordinadas de um bloco adjacente, verificando a validade do resultado lógico da condição após a execução. Se o resultado lógico da condição não for válido, o bloco adjacente de instruções

subordinadas é repetido. No momento em que o resultado lógico da condição se torna válido, a execução do laço de repetição é automaticamente encerrada.

Os laços de repetição com controle condicional pós-teste podem ser de duas formas, sendo um laço de repetição para condição falsa e outro para condição verdadeira.

5.4.1 - Controle Condicional Falso

O laço de repetição com controle condicional pós-teste falso executa no mínimo uma vez as instruções subordinadas de um bloco adjacente e mantém a execução no período em que o resultado lógico da condição permanece falso. No momento em que o resultado lógico da condição se torna verdadeiro, a execução do laço é automaticamente encerrada.

Observe na Figura 5.7 um diagrama de blocos com o rótulo **N** para a execução de um laço de repetição com controle condicional pós-teste falso com base na **CONDIÇÃO** do símbolo *decision*. A linha de fluxo do rótulo **N** indica o retorno para um determinado trecho, configurando assim a execução do laço de repetição. Já o rótulo **S** indica a saída do laço de repetição.

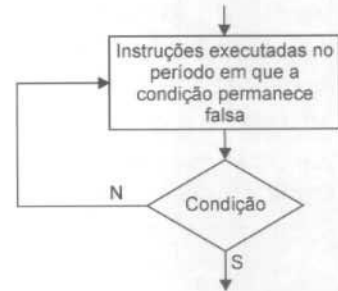


Figura 5.7 - Estrutura de laço de repetição com controle condicional pós-teste falso.

O laço de repetição com controle condicional pós-teste falso, do ponto de vista da codificação em português estruturado, utiliza os comandos **repita** e **até_que** na construção da instrução **repita/até_que**. Nesse trecho são executadas as instruções subordinadas no bloco adjacente (entre os comandos **repita** e **até_que**) durante o período em que o resultado lógico da condição permanecer falso. No momento em que o resultado lógico for verdadeiro, executam-se as eventuais instruções que estiverem após o comando **até_que**. Observe a estrutura sintática seguinte:

```

repita
  [instruções executadas durante o período em que a condição é falsa]
até_que (<condição>)
  
```

Atente para o detalhe da *indentação* para sinalizar as instruções subordinadas ao bloco adjacente da instrução **repita/até_que**. A indicação **<condição>** entre parênteses deve ser substituída pela expressão lógica da condição a ser utilizada. Os trechos sinalizados entre colchetes representam as instruções a serem executadas pelo programa.

A título de ilustração de uso de laço de repetição com controle condicional pós-teste falso em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação. Os programas demonstram os laços de repetição nas formas iterativa e interativa.

Exemplo com laço iterativo

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor entrado, multiplicando-o por 3 e apresentando seu resultado. Proceder à execução dos passos anteriores cinco vezes.

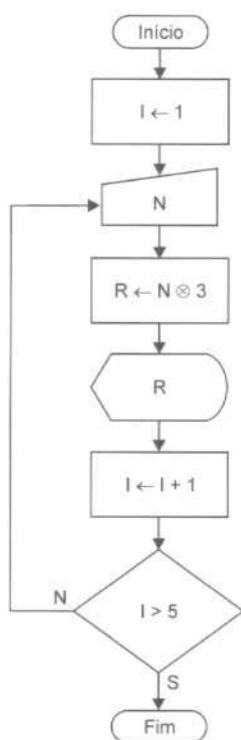
Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.8.

Entendimento

O problema é o mesmo exposto anteriormente. A diferença é a forma de construção da condição e do laço de repetição. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para servir como contador com valor inicial 1 (variável I).
2. Ler um valor inteiro qualquer (variável N).
3. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
4. Apresentar o valor calculado que está armazenado na variável R.
5. Acrescentar o valor 1 ao valor existente da variável I, definida no passo 1.
6. Verificar se a variável contador é maior que 5; sendo, executar os passos 2, 3 e 4; caso contrário, desviar o programa para o passo 7.
7. Encerrar a execução do programa.

Diagramação



Codificação

```

programa LAÇO_PÓS_TESTE_FALSO_VA
var
  I, N, R : inteiro
inicio
  I ← 1
  repita
    leia N
    R ← N * 3
    escreva R
    I ← I + 1
  até_que (I > 5)
fim
  
```

Figura 5.8 - Exemplo da instrução repita/até_que iterativa.

O bloco adjacente de instruções subordinadas entre os comandos **repita** e **até_que** se repete até que o resultado lógico da condição se torne verdadeiro, ou seja, quando a variável *I* que foi iniciada com valor 1 tiver um valor maior que 5. Quando o resultado lógico da condição for verdadeiro, o laço de repetição é encerrado, enviando o fluxo do programa para a primeira instrução após a instrução **até_que**.

Exemplo com laço interativo

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor de entrada, multiplicando-o por 3 e apresentando seu resultado. Ao final o programa deve perguntar ao usuário se ele deseja novo cálculo. Se a resposta for sim, deve executar novamente as instruções subordinadas ao bloco adjacente. Se a resposta for não, o programa deve parar a execução.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.9.

Entendimento

Por ser um problema semelhante a uma situação já exposta, basta escolher um laço que vai satisfazer a vontade do usuário do programa. Assim sendo, não é possível saber quantas vezes o laço será executado. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para ser utilizada como resposta (variável RESP).
2. Ler um valor inteiro qualquer (variável N).
3. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
4. Apresentar o valor calculado que está armazenado na variável R.
5. Perguntar para o usuário se ele deseja continuar o programa.
6. Caso o valor da variável RESP seja diferente de "S", executar os passos 2, 3 e 4; caso contrário, ir para o passo 7.
7. Encerrar o programa.

O bloco adjacente de instruções subordinadas entre os comandos **repita** e **até_que** é repetido até que o resultado lógico da condição se torne verdadeiro, ou seja, quando a variável *RESP* iniciada com o valor "S" estiver com um valor diferente de "S". Quando o resultado lógico da condição for verdadeiro, o laço de repetição é encerrado, enviando o fluxo do programa para a primeira instrução após a instrução **até_que**.

O laço de repetição com controle condicional pós-teste falso é encontrado, por exemplo, nas linguagens PASCAL e BASIC. Essa forma de laço de repetição com controle condicional pós-teste não é muito encontrada nas linguagens de programação.

Diagramação

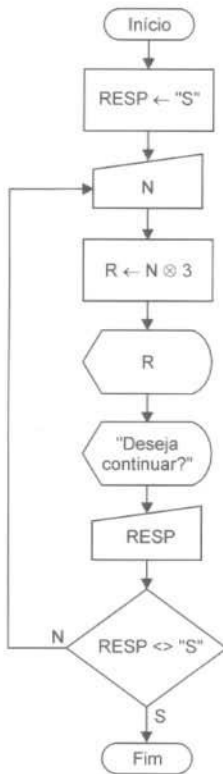


Figura 5.9 - Exemplo da instrução repita/até_que interativa.

Codificação

```

programa LAÇO_PÓS_TESTE_FALSO_VB
var
  N, R : inteiro
  RESP : caractere
inicio
  RESP ← "S"
  repita
    leia N
    R ← N * 3
    escreva R
    escreva "Deseja continuar?"
    leia RESP
  até_que (RESP <> "S")
fim
  
```

5.4.2 - Controle Condicional Verdadeiro

O laço de repetição com controle condicional pós-teste verdadeiro executa no mínimo uma vez as instruções subordinadas de um bloco adjacente e mantém a execução no período em que o resultado lógico da condição permanece verdadeiro. No momento em que o resultado lógico da condição se tornar falso, o laço é automaticamente encerrado.

Observe na Figura 5.10 um diagrama de blocos com o rótulo **S** para a execução de um laço de repetição com controle condicional pós-teste verdadeiro com base na **CONDIÇÃO** do símbolo *decision*.

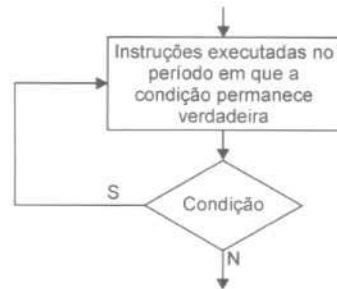


Figura 5.10 - Estrutura de laço de repetição com controle condicional pós-teste verdadeiro.

A linha de fluxo do rótulo **S** indica o retorno para um determinado trecho, configurando assim o laço de repetição. Já o rótulo **N** indica a saída do laço de repetição.

A título de ilustração de laço de repetição com controle condicional pós-teste verdadeiro em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação. Os programas demonstram os laços de repetição nas formas iterativa e interativa.

O laço de repetição com controle condicional pós-teste verdadeiro, do ponto de vista da codificação em português estruturado, utiliza os comandos **continua** e **enquanto_for** na construção da instrução **continua/enquanto_for**. Nesse trecho de instrução são executadas as instruções subordinadas ao bloco adjacente (entre os comandos **continua** e **enquanto_for**) durante o período em que o resultado lógico da condição permanecer verdadeiro. No momento em que o resultado lógico for falso, serão executadas as eventuais instruções que estiverem após o comando **enquanto_for**. Observe a estrutura sintática seguinte:

```
continua  
  [instruções executadas durante o período em que a condição é verdadeira]  
enquanto_for (<condição>)
```

Atente para o detalhe da *indentação* para sinalizar as instruções subordinadas ao bloco adjacente da instrução **continua/enquanto_for**. A indicação **<condição>** entre parênteses deve ser substituída pela expressão lógica da condição a ser utilizada. Os trechos sinalizados entre colchetes representam as instruções a serem executadas pelo programa.

Exemplo com laço iterativo

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor de entrada, multiplicando-o por 3 e apresentando seu resultado. Proceder à execução dos passos anteriores cinco vezes.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 5.11.

Entendimento

O problema exposto é o mesmo apresentado nas páginas anteriores deste capítulo. A diferença é a forma de construir e executar o laço de repetição. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para servir como contador com valor inicial 1 (variável I).
2. Ler um valor inteiro qualquer (variável N).
3. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
4. Apresentar o valor calculado que está armazenado na variável R.
5. Acrescentar o valor 1 ao valor existente da variável I, definida no passo 1.
6. Verificar se a variável contador é menor ou igual a 5; sendo, executar os passos 2, 3 e 4; caso contrário, desviar a execução do programa para o passo 7.
7. Encerrar o programa.

Diagramação

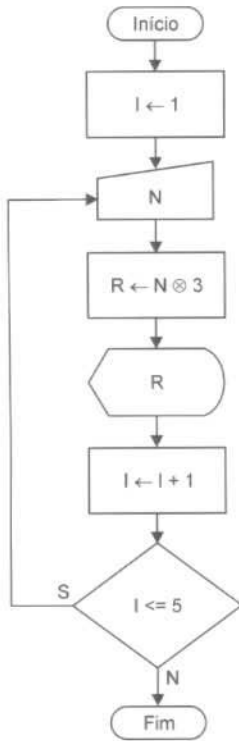


Figura 5.11 - Exemplo da instrução continua/enquanto_for iterativa.

Codificação

```

programa LAÇO_PÓS_TESTE_VERDADEIRO_VA
var
  I, N, R : inteiro
inicio
  I ← 1
  continua
  leia N
  R ← N * 3
  escreva R
  I ← I + 1
  enquanto_for (I <= 5)
fim
  
```

O bloco adjacente de instruções subordinadas entre os comandos **continua** e **enquanto_for** se repete até que o resultado lógico da condição se torne falso, ou seja, enquanto a variável *I* que foi iniciada com valor 1 permanecer com um valor menor ou igual a 5. Quando o resultado lógico da condição for falso, o laço de repetição é encerrado, enviando o fluxo do programa para a primeira instrução após a instrução **enquanto_for**.

Exemplo com laço iterativo

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor de entrada, multiplicando-o por 3 e apresentando seu resultado. Ao final o programa deve perguntar ao usuário se ele deseja novo cálculo. Se a resposta for sim, deve executar novamente as instruções subordinadas ao bloco adjacente. Se a resposta for não, o programa deve parar a execução.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.12.

Entendimento

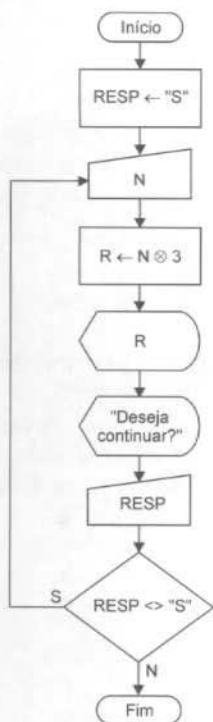
Por ser este problema semelhante a uma situação já exposta, basta definir um laço que atende a vontade do usuário do programa. Assim sendo, não é possível saber quantas vezes o laço será executado. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para ser utilizada como resposta (variável RESP).
2. Ler um valor inteiro qualquer (variável N).
3. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
4. Apresentar o valor calculado que está armazenado na variável R.
5. Perguntar para o usuário se ele deseja continuar o programa.
6. Caso o valor da variável RESP seja igual a "S", executar os passos 2, 3 e 4; caso contrário, ir para o passo 7.
7. Encerrar o programa.

O bloco adjacente de instruções subordinadas entre os comandos **continua** e **enquanto_for** se repete até que o resultado lógico da condição se tome verdadeiro, ou seja, enquanto a variável *RESP* possuir valor igual a "S". Quando o resultado lógico da condição for falso, o laço de repetição é encerrado, enviando o fluxo do programa para a primeira instrução após a instrução **enquanto_for**.

O laço de repetição com controle condicional pós-teste verdadeiro é encontrado, por exemplo, nas linguagens C, C++, JAVA, JavaScript, ActionScript, entre outras. Essa forma de laço de repetição com controle condicional pós-teste é comum encontrar nas linguagens de programação.

Diagramação



Codificação

```

programa LAÇO_PÓS_TESTE_FALSO_VB
var
  N, R : inteiro
  RESP : caractere
inicio
  RESP ← "S"
  continua
  leia N
  R ← N * 3
  escreva R
  escreva "Deseja continuar?"
  leia RESP
  enquanto_for (RESP = "S")
fim
  
```

Figura 5.12 - Exemplo da instrução **continua/enquanto_for** interativa.

5.5 - Laço de Repetição Condicional Seletivo

O laço de repetição condicional seletivo permite selecionar o local onde a decisão de saída do laço será incrementada, podendo colocá-la, por exemplo, no início, simulando uma ação condicional pré-teste, no final, simulando uma ação condicional pós-teste, como também pode-se colocar no meio dele, que é a forma mais comumente usada. Esse tipo é também referenciado como *laço localizado* (SEBESTA, 2003, p.315-317) ou *laço infinito* (THOMPSON, 2006).

A Figura 5.13 mostra um diagrama de blocos com o laço de repetição condicional seletivo. O símbolo *decision* é posicionado no meio do diagrama, mas poderia estar em qualquer outro lugar. Devido à característica de poder escolher o local onde a decisão será colocada, esse tipo de laço de repetição condicional é seletivo.

Perceba na Figura 5.13 os rótulos **S** e **N** para a verificação condicional do laço de acordo com a **CONDIÇÃO** do símbolo *decision*. A linha de fluxo sinalizada com o rótulo **S** faz o desvio do fluxo de programa para fora do laço quando o resultado lógico da condição é verdadeiro. No caso de o resultado lógico da condição ser falso, o laço permanece em execução.

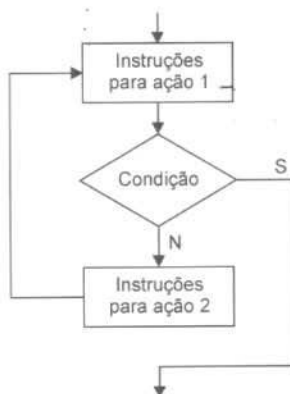


Figura 5.13 - Estrutura de laço de repetição com controle condicional seletivo.

O laço de repetição com controle condicional seletivo, do ponto de vista da codificação em português estruturado, utiliza os comandos **laço**, **saia_caso** e **fim_laço** na construção da instrução **laço/saia_caso/fim_laço** que executa o trecho de instruções subordinadas do bloco adjacente entre os comandos **laço** e **fim_laço** até o momento em que o resultado lógico da condição representada pela instrução do comando **saia_caso** torna-se verdadeiro. Nesse ponto o fluxo do programa é desviado para fora do laço de repetição. Observe a estrutura sintática seguinte:

```

laço
[instruções para ação 1]
  saia_caso (<condição>)
[instruções para ação 2]
fim_laço
  
```

Atente para o detalhe da *indentação* para sinalizar as instruções subordinadas ao bloco adjacente da instrução **laço/fim_laço**. A indicação **<condição>** após o comando **saia_caso** deve ser substituída pela expressão lógica da condição a ser utilizada. Os trechos sinalizados entre colchetes representam as instruções a serem executadas pelo programa.

A título de ilustração de laço de repetição com controle condicional seletivo em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor de entrada, multiplicando-o por 3 e apresentando seu resultado. Proceder à execução dos passos anteriores cinco vezes.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.14.

Entendimento

Para a solução do problema com o laço de repetição condicional seletivo é usada uma condição de saída um pouco diferente das formas anteriormente utilizadas. Atente para os passos descritos em seguida:

1. Criar uma variável de controle para servir como contador com valor inicial 1 (variável I).
2. Ler um valor inteiro qualquer (variável N).
3. Efetuar a multiplicação do valor de N por 3, colocando o resultado na variável R.
4. Apresentar o valor calculado que está armazenado na variável R.
5. Verificar se a variável contador é menor ou igual a 4 e executar os passos 2, 3 e 4; caso contrário, desviar o programa para o passo 7.
6. Acrescentar o valor 1 ao valor existente da variável I do passo 1 e retornar o programa ao passo 2.
7. Encerrar a execução do programa.

O bloco adjacente de instruções subordinadas entre os comandos **laço** e **fim_laço** é repetido até que o resultado lógico da condição **saia_caso (I > 4)** se torne verdadeiro, ou seja, enquanto a variável I que foi iniciada com o valor 1 tiver um valor maior que 5. Quando o resultado lógico da condição for falso, o laço de repetição é encerrado, enviando o fluxo do programa para a primeira instrução após a instrução **fim_laço**.

Apesar de ser uma estrutura de laço de repetição muitas vezes atraente, esse laço não é encontrado em muitas das linguagens de programação existentes. De forma direta é encontrado nas linguagens BASIC e ADA e de forma indireta, de modo que essa ação possa ser simulada a partir de suas habituais instruções de laço, existe nas linguagens C, C++, JAVA, PERL e PASCAL, entre outras.

A estrutura de laço de repetição condicional seletiva apresenta duas desvantagens. A primeira está relacionada ao seu uso em trabalhos com laços encadeados, tornando o encadeamento confuso, principalmente quando a condição de saída do laço está no meio da estrutura. A segunda está relacionada ao fato de não ser uma forma muito adequada de trabalhar com laços interativos, quando há outras possibilidades para executar essa tarefa, como os laços de repetição condicionais pré-teste ou pós-teste. Além dessas duas desvantagens, é uma estrutura de laço não muito elegante.

Diagramação

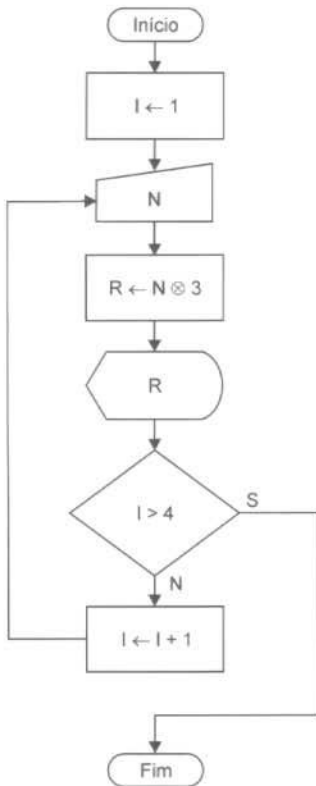


Figura 5.14 - Exemplo da instrução laço/saia_caso/fim_laço iterativa.

Codificação

```

programa LAÇO_SELETIVO
var
  I, N, R : inteiro
inicio
  I ← 1
  laço
    leia N
    R ← N * 3
    escreva R
    saia_caso (I > 4)
    I ← I + 1
  fim_laço
fim
  
```

5.6 - Laço de Repetição Incondicional

Este capítulo apresentou algumas formas de laços de repetição iterativos e interativos. Este tópico mostra um laço de repetição que realiza apenas e exclusivamente laços iterativos, por ser incondicional.

O laço de repetição incondicional é representado em um diagrama de blocos. O símbolo especificado para essa representação é o *preparation*, adotado nesta obra. Observe a indicação, dentro do símbolo *preparation*, da variável a ser controlada com a atribuição dos valores de início, fim e incremento, separados por vírgula, Figura 5.15.



Figura 5.15 - Estrutura de laço de repetição com controle incondicional.

O laço de repetição incondicional, do ponto de vista da codificação em português estruturado, utiliza os comandos **para**, **de**, **até**, **passo**, **faça** e **fim_para** para realizar a construção da instrução **para...de...até...passo...faça/fim_para**, e a estrutura desse tipo de laço tem o seu funcionamento controlado por uma variável denominada contador. Pode-se executar um determinado conjunto de instruções subordinadas a um bloco adjacente um certo número de vezes. No momento em que o valor da variável de controle atingir o valor definido no segmento de fim de contagem, serão executadas as eventuais instruções que estiverem após o comando **fim_para**. Observe a estrutura sintática seguinte:

```
para [<variável>] de [<início>] até [<fim>] passo [<incremento>] faça
  [<instruções executadas durante o ciclo de contagem da variável de controle>]
fim_para
```

A título de ilustração de laço de repetição incondicional em um contexto operacional, considere o problema a seguir, observando detalhadamente as etapas de ação de um programador de computador: entendimento, diagramação e codificação.

Elaborar um programa que efetue a entrada de um valor numérico inteiro qualquer, em seguida calcule o valor de entrada, multiplicando-o por 3 e apresentando seu resultado. Proceder à execução dos passos anteriores cinco vezes.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.16.

Diagramação

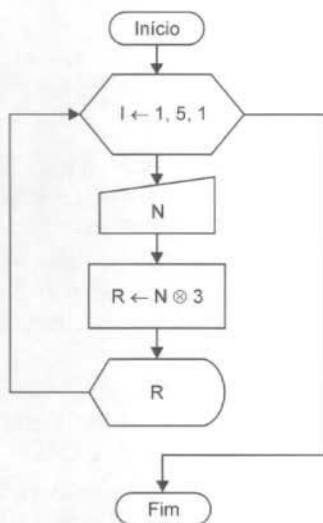


Figura 5.16 - Exemplo da estrutura para...de...até...passo...faça...fim_para.

Codificação

```
programa LAÇO_INCONDICIONAL
var
  I, N, R : inteiro
inicio
  para I de 1 até 5 passo 1 faça
    leia N
    R ← N * 3
    escreva R
  fim_para
fim
```

Entendimento

O problema exposto já é conhecido, mas desta vez sua operação será mais dinâmica. Atente para os passos descritos em seguida:

1. Definir uma variável do tipo contador (variável *I*), variando de 1 a 5, de 1 em 1.
2. Ler um valor inteiro qualquer (variável *N*).
3. Efetuar a multiplicação do valor de *N* por 3, colocando o resultado na variável *R*.
4. Apresentar o valor calculado que está armazenado na variável *R*.
5. Repetir os passos 2, 3, 4 e 5 até o valor da variável chegar a 5.
6. Encerrar a execução do programa.

O conjunto de instruções subordinadas ao bloco adjacente é executado entre os comandos **para** e **fim_para**, sendo a variável *I* (variável de controle) inicializada com valor 1 e incrementada de mais 1 por meio do comando **passo** até a variável *I* atingir o valor 5. Esse tipo de estrutura de repetição pode ser utilizado todas as vezes que houver a necessidade de repetir trechos finitos, em que se conhecem os valores do laço: inicial, final e incremento.

5.7 - Considerações entre Tipos de Laços

No decorrer deste capítulo, foram apresentadas seis estruturas de controle de laços de repetição:

- ▶ **enquanto/fim_enquanto** (laço de repetição condicional);
- ▶ **até_seja/fim_até_seja** (laço de repetição condicional);
- ▶ **repita/até_que** (laço de repetição condicional);
- ▶ **continua/enquanto_for** (laço de repetição condicional);
- ▶ **laço/fim_laço** (laço de repetição condicional);
- ▶ **para/fim_para** (laço de repetição incondicional).

Cada laço possui uma característica de processamento. Neste aspecto, deve-se notar que os laços mais versáteis são os condicionais, exceto o laço de repetição condicional seletivo, pois podem ser substituídos uns pelos outros, tanto que é comum encontrar uma ou outra forma nas linguagens de programação existentes.

Outro detalhe a ser considerado é que os laços de repetição condicionais podem ser utilizados na construção de laços interativos e iterativos, ressalva já considerada em relação ao laço de repetição condicional seletivo. O laço de repetição condicional pode substituir a ação de um laço de repetição incondicional, mas o contrário não é verdadeiro.

No capítulo 4, foi comentado o fato de ocorrer o encadeamento das estruturas de tomadas de decisão, o que se estende às estruturas de laços de repetição. Neste ponto pode ocorrer o encadeamento de uma estrutura de laço de repetição com outra. Essas ocorrências vão depender do problema a ser solucionado.

Outro fato a ser lembrado é com relação às variáveis de ação e de controle, como comentado no tópico 3.3. Nos exemplos deste capítulo esses dois conceitos ficam muito evidentes. Observe as variáveis *I* e *RESP* que nos vários exemplos efetuaram o processamento de controle dos laços, enquanto *N* e *R* são as variáveis de ação dos programas apresentados.

Os laços de repetição representam a etapa de programação mais importante, pois permitem escolher a base para que um programa tenha funcionalidade mais ampla e aplicada. Sem os laços a tarefa de programação seria muito desestimulante.

5.8 - Exercício de Aprendizagem

A seguir encontram-se alguns exemplos das estruturas de laços de repetição iterativos apresentadas neste capítulo.

Elaborar um programa que efetue o cálculo da fatorial do valor inteiro 5 e apresente o resultado dessa operação.

Para entender o problema proposto, considere que, do ponto de vista matemático, *fatorial* é o produto dos números naturais desde 1 até o limite informado, neste caso 5. Assim sendo, a fatorial do valor 5, representada matematicamente como $5!$, é a multiplicação de $1 \times 2 \times 3 \times 4 \times 5$, que resulta no valor 120.

Para efetuar essa operação, são necessárias duas variáveis, uma que controla as iterações do laço de repetição e outra que calcula a fatorial propriamente dito. Veja em seguida:

Entendimento

A descrição do entendimento e da solução do problema proposto é basicamente a mesma para qualquer forma de laço, pois é levado em consideração o uso de laço iterativo. Considere os passos seguintes:

1. Inicializar as variáveis FAT e CONT, ambas com valor inicial 1.
2. Multiplicar o valor da variável FAT sucessivamente pelo valor da variável CONT, atribuindo o resultado da multiplicação à própria variável FAT.
3. Incrementar mais 1 à variável CONT.
4. Repetir os passos 2 e 3 até o limite de 5 para a variável CONT.
5. Apresentar ao final o valor obtido na variável FAT.
6. Encerrar o programa.

Por ser necessário calcular uma fatorial de 5 ($5!$), significa que o contador deve variar de 1 a 5, e por este motivo deve ser a variável CONT inicializada com valor 1. Pelo fato de a variável FAT possuir ao final o resultado do cálculo da fatorial pretendida, ela deve ser inicializada também com valor 1. Se a variável FAT fosse inicializada com valor zero, não teria ao final o resultado da fatorial, pois qualquer valor multiplicado por zero resulta zero.

Multiplicar sucessivamente implica em multiplicar a variável CONT pelo valor atual da variável FAT por cinco vezes. Observe atentamente as etapas de cálculo indicadas na tabela de cálculo da fatorial de 5 a seguir:

Tabela de cálculo da 5!			
CONT	FAT	FAT ← FAT * CONT	Comentários
1	1	1	Valor inicial das variáveis e da fatorial
2	1	2	Cálculo da fatorial com o contador em 2
3	2	6	Cálculo da fatorial com o contador em 3
4	6	24	Cálculo da fatorial com o contador em 4
5	24	120	Cálculo da fatorial com o contador em 5

O movimento apresentado na tabela de cálculo da fatorial de 5 é o mesmo que deve ser realizado por um programa que permita ao computador executar a operação e mostrar o resultado esperado.

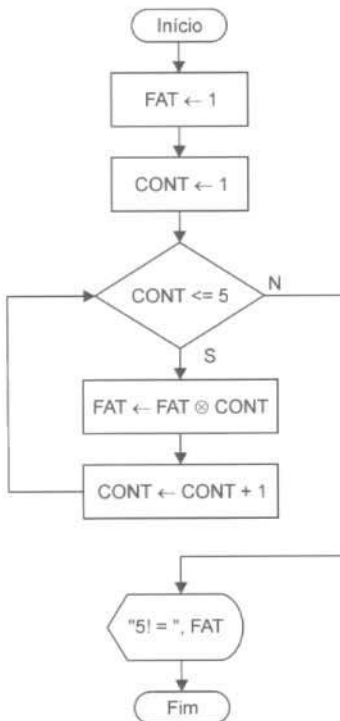
Observe na tabela a necessidade de dois contadores. O contador para a variável CONT que inicia a contagem em 1 e termina em 5 será utilizado de duas formas: como variável de controle para manter a execução do laço de repetição na frequência estabelecida, e como variável de ação para fomentar o cálculo da fatorial em si. O outro contador é a variável FAT que acumula o resultado de seu valor multiplicado pelo valor de CONT.

As variáveis CONT (primeira coluna) e FAT (segunda coluna) são iniciadas com valor 1 (como mostra a primeira linha da tabela), o que resulta no valor de FAT em 1 (terceira coluna). Observe na sequência a segunda linha da tabela, em que a variável CONT está com valor 2 e a variável FAT com valor 1. Após o cálculo da terceira coluna na segunda linha da tabela, a variável FAT passa a possuir o valor 2. Se for seguido o mesmo raciocínio linha a linha, chegar-se-á à quinta e última linha com a variável FAT com 120. Quando a variável CONT estiver com valor 5, a variável FAT estará com o valor 120.

A seguir são apresentados seis exemplos de solução do programa de cálculo da fatorial de 6, Figuras 5.17 a 5.22. Cada um dos exemplos usa as formas de laço de repetição estudadas neste capítulo e baseia-se no raciocínio descrito na tabela de cálculo da fatorial de 5. Para cada exemplo são indicados diagrama de blocos e codificação correspondente em português estruturado.

1º Exemplo - Estrutura de laço de repetição enquanto/fim_enquanto

Diagramação



Codificação

```

programa FATORIAL_EX1
var
  CONT, FAT : inteiro
inicio
  FAT ← 1
  CONT ← 1
  enquanto (CONT <= 5) faça
    FAT ← FAT * CONT
    CONT ← CONT + 1
  fim_enquanto
  escreva "5! = ", FAT
fim
  
```

Figura 5.17 - Fatorial com laço de repetição enquanto/fim_enquanto.

Observe no código do programa **FATORIAL_EX1** dois dados com o comando **escreva**, sendo um dado uma mensagem e o outro uma variável, ambas separadas por vírgula. A mensagem exibe o que representa a saída e a variável apresenta o seu valor. A junção de um dado de mensagem com um dado de variável gerou uma informação para o usuário. Ao ver a saída **5! = 120**, o usuário consegue reconhecer o significado do conteúdo apresentado.

2º Exemplo - Estrutura de laço de repetição até_seja/fim_até_seja

Diagramação

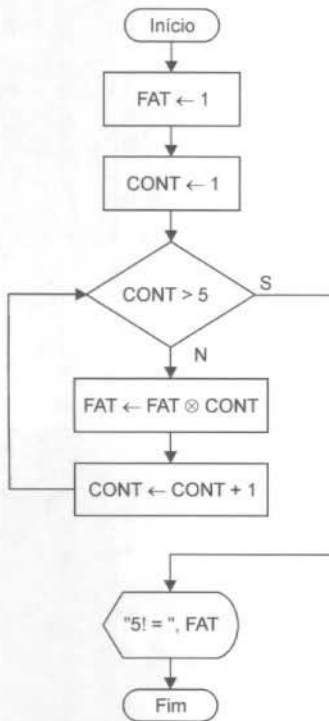


Figura 5.18 - Fatorial com laço de repetição até_seja/fim_até_seja.

Codificação

```

programa FATORIAL_EX2
var
  CONT, FAT : inteiro
inicio
  FAT ← 1
  CONT ← 1
  até_seja (CONT > 5) efetue
    FAT ← FAT * CONT
    CONT ← CONT + 1
  fim_até_seja
  escreva "5! = ", FAT
fim
  
```

3º Exemplo - Estrutura de laço de repetição repita/até_que

Diagramação

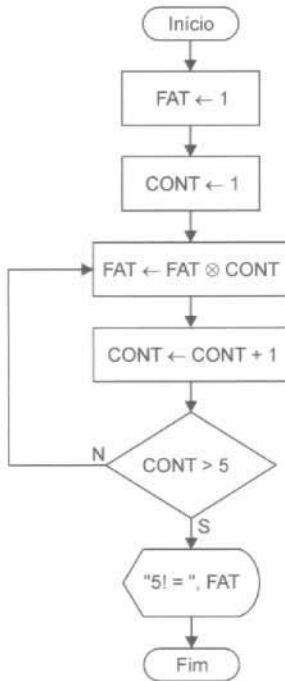


Figura 5.19 - Fatorial com laço de repetição repita/até_que.

Codificação

```

programa FATORIAL_EX3
var
  CONT, FAT : inteiro
inicio
  FAT ← 1
  CONT ← 1
  repita
    FAT ← FAT * CONT
    CONT ← CONT + 1
  até_que (CONT > 5)
  escreva "5! = ", FAT
fim
  
```

4º Exemplo - Estrutura de laço de repetição continua/enquanto_for

Diagramação

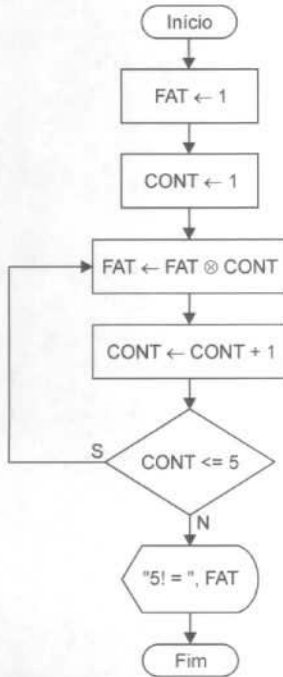


Figura 5.20 - Fatorial com laço de repetição continua/enquanto_for.

Codificação

```

programa FATORIAL_EX4
var
  CONT, FAT : inteiro
inicio
  FAT ← 1
  CONT ← 1
  continua
    FAT ← FAT * CONT
    CONT ← CONT + 1
  enquanto_for (CONT ≤ 5)
    escreva "5! = ", FAT
fim
  
```

5º Exemplo - Estrutura de laço de repetição laço/fim_laço

Diagramação

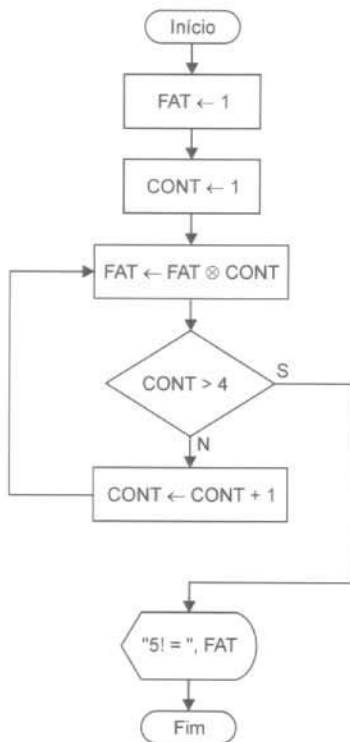


Figura 5.21 - Fatorial com laço de repetição laço/fim_laço.

Codificação

```

programa FATORIAL_EX5
var
  CONT, FAT : inteiro
inicio
  FAT ← 1
  CONT ← 1
  laço
    FAT ← FAT * CONT
    saia_caso (CONT > 4)
    CONT ← CONT + 1
  fim_laço
  escreva "5! = ", FAT
fim
  
```

6º Exemplo - Estrutura de laço de repetição para/fim_para

Diagramação

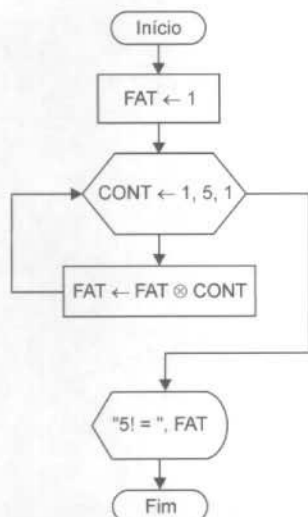


Figura 5.22 - Fatorial com laço de repetição para/fim_para.

Codificação

```

programa FATORIAL_EX6
var
  CONT, FAT : inteiro
inicio
  FAT ← 1
  para CONT de 1 até 5 passo 1 faça
    FAT ← FAT * CONT
  fim_para
  escreva "5! = ", FAT
fim
  
```

7º Exemplo

Os algoritmos (diagrama de blocos e codificação em português estruturado) apresentados anteriormente solucionam o cálculo apenas para a fatorial do número 5. Seria melhor possuir um programa mais abrangente, que permitisse o cálculo da fatorial de um número qualquer. Assim sendo, considere o seguinte problema:

Elaborar um programa que calcule e apresente o resultado do cálculo da fatorial de um número qualquer. Além dessa possibilidade, o programa deve permitir ao usuário fazer novos cálculos até o momento que decidir encerrar o programa.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 5.23.

Entendimento

Neste programa, além do cálculo da fatorial que será feito com um laço iterativo a partir da quantidade de vezes desejada para o cálculo, será utilizado um laço iterativo para executar o programa no período que o usuário desejar. Nesta situação usam-se laços encadeados.

1. Definir uma variável para controle do laço de resposta do usuário (variável RESP).
2. Indicar uma variável para controle do laço de cálculo da fatorial (variável I).
3. Definir uma variável para o limite máximo de cálculo para a fatorial (variável N).
4. Inicializar a variável RESP com valor "SIM".
5. Inicializar a variável FAT com valor 1.

6. Efetuar a entrada da variável N para definir o valor limite de cálculo.
7. Inicializar o laço da variável CONT em 1 e terminar em N de 1 em 1.
8. Calcular a expressão aritmética $FAT \leftarrow FAT * CONT$.
9. Incrementar o valor 1 à variável CONT até chegar ao limite da variável N.
10. Apresentar o resultado obtido na variável FAT.
11. Perguntar ao usuário se deseja continuar. Se a resposta for SIM, repetir os passos de 5 até 11. Se a resposta for NÃO, ir para o passo 12.
12. Encerrar o programa.

Diagramação

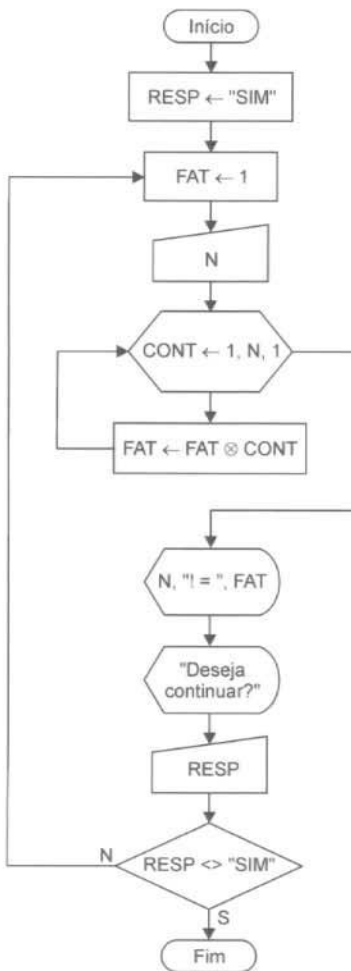


Figura 5.23 - Fatorial de um número qualquer com intervenção do usuário.

Codificação

```

programa FATORIAL_EX7
var
  CONT, FAT, N : inteiro
  RESP : cadeia
inicio
  RESP ← "SIM"
  repita
    FAT ← 1
    leia N
    para CONT de 1 até N passo 1 faça
      FAT ← FAT * CONT
    fim_para
    escreva N, "!" =, FAT
    escreva "Deseja continuar?"
    leia RESP
  até_que (RESP <> "SIM")
fim
  
```


5.9 - Exercícios de Fixação

Desenvolver os diagramas de blocos e as codificações em português estruturado dos problemas elencados de **a** até **s**, nos laços de repetição:

1. Laço de repetição condicional pré-teste verdadeiro (**enquanto/ fim_enquanto**).
2. Laço de repetição condicional pré-teste falso (**até_seja/fim_até_seja**).
3. Laço de repetição condicional pós-teste falso (**repita/até_que**).
4. Laço de repetição condicional pós-teste verdadeiro (**continua/enquanto_for**).
5. Laço de repetição condicional seletivo (**laço/fim_laço**).
6. Laço de repetição incondicional (**para/fim_para**).

É importante levar em consideração que talvez um ou outro problema não possa ser resolvido com um determinado tipo de laço de repetição. Fica a critério do professor escolher os laços e os exercícios que devem ser realizados pelo aluno.

Atente para os seguintes problemas:

- a) Elaborar um programa que apresente os quadrados dos números inteiros existentes na faixa de valores de 15 a 200.
- b) Elaborar um programa que mostre os resultados da tabuada de um número qualquer, a qual deve ser apresentada de acordo com sua forma tradicional.
- c) Construir um programa que apresente a soma dos cem primeiros números naturais ($1+2+3+\dots+98+99+100$).
- d) Elaborar um programa que apresente o somatório dos valores pares existentes na faixa de 1 até 500.
- e) Elaborar um programa que apresente todos os valores numéricos inteiros ímpares situados na faixa de 0 a 20. Sugestão: para verificar se o valor numérico é ímpar, dentro do laço de repetição, fazer a verificação lógica dessa condição com a instrução **se/fim_se** dentro do próprio laço, perguntando se o valor numérico do contador é ímpar (se é diferente de zero); sendo, mostre-o, não sendo, passe para o próximo valor numérico. Para saber se um valor numérico é ímpar, use o método de cálculo de divisibilidade do valor numérico por 2.
- f) Construir um programa que apresente todos os valores numéricos divisíveis por 4 e menores que 200. Sugestão: a variável que controla o contador do laço de repetição deve ser iniciada com valor 1.
- g) Elaborar um programa que apresente os resultados das potências do valor de base 3, elevado a um expoente que varie do valor 0 até o 15. O programa deve apresentar os valores 1, 3, 9, 27, ..., 14.348.907. Sugestão: leve em consideração as definições matemáticas do cálculo de potência, em que qualquer valor numérico elevado a zero é 1, e todo valor numérico elevado a 1 é ele próprio.
- h) Escrever um programa que apresente como resultado a potência de uma base qualquer elevada a um expoente qualquer, ou seja, de B^E , em que B é o valor da base e E o valor do expoente. Considere apenas a entrada de valores inteiros e positivos, ou seja, de valores naturais. Sugestão: não utilize o formato "base ↑ expoente", pois é uma solução muito trivial. Use a técnica de laço de repetição, em que o valor da base deve ser multiplicado o número de vezes determinado no expoente.

- i) Escrever um programa que apresente os valores da sequência numérica de Fibonacci até o décimo quinto termo. A sequência de Fibonacci é formada por 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ... etc., obtendo-se o próximo termo a partir da soma do termo atual com o anterior sucessivamente até o infinito se a sequência não for interrompida.
- j) Elaborar um programa que apresente os valores de conversão de graus Celsius em graus Fahrenheit, de dez em dez graus, iniciando a contagem em dez graus Celsius e finalizando em cem graus Celsius. O programa deve apresentar os valores das duas temperaturas.
- k) Escrever um programa que calcule e apresente o somatório do número de grãos de trigo que se pode obter num tabuleiro de xadrez, obedecendo à seguinte regra: colocar um grão de trigo no primeiro quadro e nos quadros seguintes o dobro do quadro anterior. Ou seja, no primeiro quadro coloca-se um grão, no segundo quadro colocam-se dois grãos (neste momento têm-se três grãos), no terceiro quadro colocam-se quatro grãos (tendo neste momento sete grãos), no quarto quadro colocam-se oito grãos (tendo-se então 15 grãos) até atingir o sexagésimo quarto quadro (*este exercício foi baseado numa situação exposta no capítulo 16 do livro "O Homem que Calculava" de Malba Tahan, da Editora Record*).
- l) Elaborar um programa que leia quinze valores numéricos inteiros e no final apresente o somatório da fatorial de cada valor lido.
- m) Elaborar um programa que leia dez valores numéricos reais e apresente no final o somatório e a média dos valores lidos.
- n) Elaborar um programa que leia sucessivamente valores numéricos e apresente no final o somatório, a média e o total de valores lidos. O programa deve ler os valores enquanto o usuário estiver fornecendo valores positivos. Ou seja, o programa deve parar quando o usuário fornecer um valor negativo (menor que zero).
- o) Construir um programa que apresente como resultado a fatorial dos valores ímpares situados na faixa numérica de 1 até 10.
- p) Elaborar um programa que apresente os resultados da soma e da média aritmética dos valores pares situados na faixa numérica de 50 até 70.
- q) Escrever um programa que possibilite calcular a área total em metros de uma residência com os cômodos sala, cozinha, banheiro, dois quartos, área de serviço, quintal, garagem, entre outros que podem ser fornecidos ao programa. O programa deve solicitar a entrada do nome, da largura e do comprimento de um determinado cômodo. Em seguida, deve apresentar a área do cômodo lido e também uma mensagem solicitando ao usuário a confirmação de continuar calculando novos cômodos. Caso o usuário responda "NÃO", o programa deve apresentar o valor total acumulado da área residencial.
- r) Elaborar um programa que leia valores positivos inteiros até que um valor negativo seja informado. Ao final devem ser apresentados o maior e o menor valores informados pelo usuário.
- s) Elaborar um programa que apresente o resultado inteiro da divisão de dois números quaisquer, representando o dividendo e o divisor da divisão a ser processada. Sugestão: para a elaboração do programa, não utilize o operador aritmético de divisão com quociente inteiro DIV. Use uma solução baseada em laço de repetição. O programa deve apresentar como resultado (quociente) quantas vezes o divisor cabe no dividendo.

6

Estruturas de Dados Homogêneas de Uma Dimensão

Os capítulos 3, 4 e 5 mostraram as três técnicas básicas (sequência, tomada de decisão e laço de repetição) que possibilitam o desenvolvimento de programas mais complexos para qualquer finalidade. Essas técnicas facilitaram a compreensão da escrita de programas (BOHM & JACOPINI, 1966).

Nos exemplos apresentados foram usadas variáveis consideradas simples, o que limita em muito a atividade de programação, pois elas somente armazenam um valor por vez, obrigando o programador a trabalhar com um grande conjunto de variáveis quando precisar de um volume de dados maior.

Este capítulo descreve uma técnica de programação que auxilia o agrupamento de dados do mesmo tipo em uma mesma variável indexada. Devido a essa característica referencia-se essa técnica como *estrutura de dados homogênea*.

A estrutura de dados homogênea em programação recebe diversos nomes, como variáveis indexadas, variáveis compostas, variáveis subscriptas, arranjos, vetores, matrizes, tabelas em memória, *arrays* (do inglês) ou conjuntos.

6.1 - Ser Programador

É fundamental ponderar sobre as palavras de Carlos Almeida, professor da Escola Secundária Emídio Navarro, da região de Viseu em Portugal. Ele diz que um "algoritmo não é a solução de um problema, pois, se assim fosse, cada problema teria um único algoritmo", e complementa: "algoritmo é um caminho para a solução de um problema, em geral, os caminhos que a uma solução são muitos".

O desenvolvimento de algoritmos utiliza procedimentos lógicos e de raciocínio na busca da solução de problemas. Um mesmo problema pode ser resolvido de várias formas, tanto por uma pessoa como por um conjunto de pessoas. No entanto, resolver um problema de várias formas não significa, em absoluto, escrever a solução de qualquer jeito. A escrita de algoritmos deve respeitar um certo formalismo. A base do desenvolvimento de algoritmos, do ponto de vista computacional, é o próprio computador. É necessário ao programador ter seu pensar modelado no computador, como indicado no capítulo 2, e para pensar "computador" é preciso seguir as regras apresentadas neste trabalho.

Como exposto, também no capítulo 2, programar não é trabalho fácil. Se programar um computador fosse simples, todas as pessoas o fariam e não seria necessário muito esforço para o aprendizado. A grande dificuldade está exatamente no fato de existirem vários caminhos para a solução de um problema e ter de seguir regras de trabalho. Como saber o caminho correto? Qual é o melhor algoritmo para o problema?

O professor Carlos Almeida adverte que o aprendizado de algoritmos não se faz copiando algoritmos, mas construindo e testando-os, ou seja, exercitando algoritmos. Não há outra forma de fazê-lo.

Por mais experiência que tenha um programador, em algum momento da vida profissional terá de elaborar e testar algoritmos muitas vezes complexos e diferentes da forma que esteja acostumado. O programador é um eterno estudante.

A construção de algoritmos voltados à programação de computadores exige cuidado e atenção. Assim que a solução de um certo problema computacional, por meio de um algoritmo, é elaborada, é necessário realizar testes para verificar se a linha de raciocínio é correta, chamados *testes de mesa*.

O *teste de mesa* é semelhante a tirar a "prova dos nove" para verificar se um cálculo matemático está correto. Leva este nome por ser feito numa folha de papel sobre uma mesa. Não se fazem testes de mesa em computadores.

É a verificação da linha de raciocínio do programador, o modo como o programador confere se o seu pensar está correto, se está dentro da esfera de funcionamento de um computador. O *teste de mesa* pode também ser utilizado como mecanismo de verificação da lógica de programação de outras pessoas.

De forma prática não existe oficialmente um mecanismo de elaboração de *teste de mesa*. A tarefa de verificação é muito pessoal. No entanto, o tópico 5.8 (capítulo 5) apresenta uma forma, uma sugestão, de *teste de mesa* ao descrever o processo de entendimento do cálculo da fatorial do valor cinco em uma tabela de valores.

6.2 - Matrizes de Uma Dimensão

A matriz de uma dimensão é a forma mais simples de usar tabelas de valores com apenas uma coluna e várias linhas de dados. Essa estrutura de dados fica em uma única variável dimensionada com um determinado tamanho. A dimensão de uma matriz é formada por constantes inteiras e positivas. Os nomes dados a uma variável composta (matriz) seguem as mesmas regras dos nomes dados a variáveis simples e também ao comando **var** com auxílio do comando **conjunto** na definição da matriz.

Para ter ideia de como usar uma matriz de uma dimensão em certa situação e de sua vantagem operacional de trabalho, considere o seguinte problema:

Elaborar um programa que calcule e apresente o valor da média geral das médias individuais de uma classe com oito alunos.

É necessário somar primeiramente as médias de cada um dos alunos para dividir pela quantidade de alunos da classe.

A tabela de médias escolares seguinte apresenta a quantidade de alunos da classe, suas notas bimestrais e respectivas médias. É a partir da média (coluna **Média**) de cada um dos alunos que será calculada a média da turma.

Tabela de médias escolares					
Aluno	Nota 1	Nota 2	Nota 3	Nota 4	Média
1	4,0	6,0	5,0	3,0	4,5
2	6,0	7,0	5,0	8,0	6,5
3	9,0	8,0	9,0	6,0	8,0
4	3,0	5,0	4,0	2,0	3,5
5	4,0	6,0	6,0	8,0	6,0
6	7,0	7,0	6,0	5,0	6,5

Tabela de médias escolares					
Aluno	Nota 1	Nota 2	Nota 3	Nota 4	Média
7	8,0	7,0	6,0	5,0	6,5
8	6,0	7,0	2,0	9,0	6,0

A partir dos dados da coluna **Média** da tabela, basta escrever um programa para fazer o cálculo da média geral das oito médias de cada aluno. Para representar a média do primeiro aluno será utilizada a variável **MD1**, para a média do segundo aluno a variável **MD2** e assim por diante até a oitava média associada à variável **MD8**. Considere as variáveis simples seguintes e seus respectivos valores:

MD1 ← 4.5
 MD2 ← 6.5
 MD3 ← 8.0
 MD4 ← 3.5
 MD5 ← 6.0
 MD6 ← 7.0
 MD7 ← 6.5
 MD8 ← 6.0

Com o conhecimento adquirido até o momento, deve ser elaborado um programa que leia cada uma das oito médias, efetue a soma delas e a divisão por oito e apresente o resultado da média geral obtida. Observe detalhadamente o exemplo da Figura 6.1 e sua codificação:

Diagramação

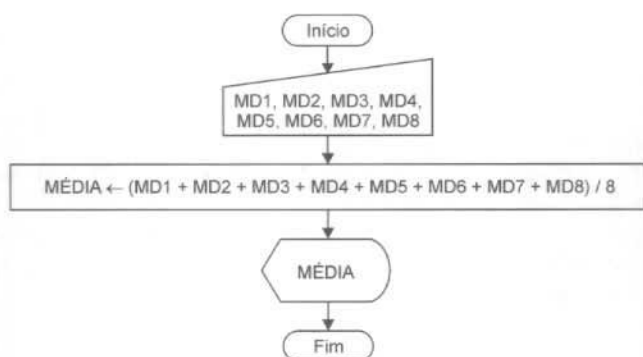


Figura 6.1 - Diagrama de bloco do programa de cálculo de média geral.

Codificação

```

programa MÉDIA_GERAL_V1
var
  MD1, MD2, MD3, MD4, MD5, MD6, MD7, MD8, MÉDIA : real
início
  leia MD1, MD2, MD3, MD4, MD5, MD6, MD7, MD8
  MÉDIA ← (MD1 + MD2 + MD3 + MD4 + MD5 + MD6 + MD7 + MD8) / 8
  escreva MÉDIA
fim
  
```

Para receber a média de cada aluno da classe, foi necessário utilizar oito variáveis simples. Com a técnica de matrizes é possível utilizar apenas uma variável composta com a capacidade de armazenar oito valores.

Uma matriz de uma dimensão será representada neste trabalho por nome de identificação e tamanho (dimensão) entre colchetes. Para o exemplo usa-se a variável composta **MD** para armazenamento das médias de cada aluno. A variável composta **MD** deve ter dimensão total de oito elementos. A dimensão será indicada entre parênteses pelo valor de índice inicial até o índice final separados por ponto e ponto, ou seja, para oito elementos usa-se a dimensão **[1..8]** escrita ao lado direito do nome da matriz. Desta forma, a matriz **MD** de dimensão **[1..8]** fica como **MD[1..8]**, indicando individualmente a existência das posições **MD[1]** (variável **MD**, índice **1**), **MD[2]** (variável **MD**, índice **2**) e assim sucessivamente até **MD[8]** (variável **MD**, índice **8**).

É pertinente lembrar o exposto no tópico 3.3 (capítulo 3), que variável é *uma região de memória utilizada para armazenar um valor por um determinado espaço de tempo*. Esta definição foi explanada, na ocasião, em relação ao uso de variáveis simples. No uso de matrizes, esta é uma variável que pode armazenar mais de um valor por vez, respeitando o limite da dimensão estabelecida, por ser dimensionada. É importante considerar que cada um dos índices, cada uma das posições de uma matriz, só pode armazenar um valor por vez, caracterizando a manipulação dos elementos de uma matriz de forma individualizada.

No exemplo do cálculo da média geral das médias individuais dos oito alunos, ter-se-á uma única variável indexada (a matriz) denominada **MD**, contendo os valores das médias de cada aluno. Isso é representado da seguinte forma:

```
MD[1] ← 4.5
MD[2] ← 6.5
MD[3] ← 8.0
MD[4] ← 3.5
MD[5] ← 6.0
MD[6] ← 7.0
MD[7] ← 6.5
MD[8] ← 6.0
```

O nome da variável indexada (matriz) é um só: **MD**. O que de fato muda é a informação indicada entre colchetes, os índices da matriz que são os endereços (a posição) em que um certo elemento (o conteúdo) está armazenado. É necessário que fique bem claro que elemento é o conteúdo, é o dado armazenado da matriz, neste caso representado pelos valores das médias de cada um dos oito alunos atribuídos a uma posição da matriz. No caso **MD[1] ← 4.5**, sendo o número **1** o valor do índice; o endereço cujo elemento **4.5** está armazenado está sendo atribuído.

Em português estruturado as matrizes usam o comando **conjunto** que indica uma matriz, tendo como sintaxe a estrutura apresentada em seguida:

```
VARIÁVEL : conjunto[<dimensão>] de <tipo de dado>
```

Em que **VARIÁVEL** é o nome da variável que será indexada, **<dimensão>** indica os valores inicial e final do tamanho da matriz unidimensional entre ponto e ponto e **<tipo de dado>** o tipo de dado operacionalizado na matriz (**real**, **inteiro**, **lógico** e **cadeia** ou **caractere**).

6.2.1 - Leitura dos Dados de uma Matriz

A leitura dos dados de uma matriz é processada passo a passo, um elemento por vez, por meio de um laço de repetição. A seguir são apresentados diagrama de blocos, Figura 6.2, e codificação em português estruturado da leitura das médias de cada um dos alunos, cálculo da média geral e a apresentação do resultado.

Diagramação

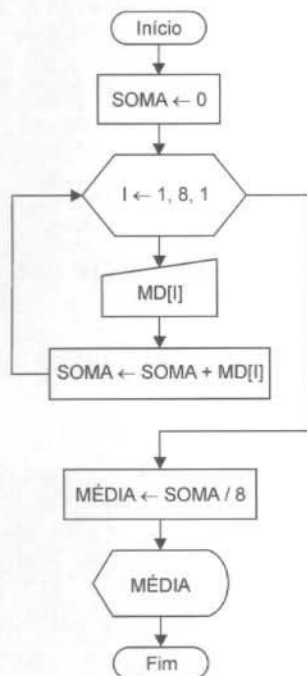


Figura 6.2 - Diagrama de blocos para leitura dos elementos de uma matriz do tipo vetor.

Codificação

```

programa MÉDIA_GERAL_V2
var
  I : inteiro
  MD : conjunto[1..8] de real
  SOMA, MÉDIA : real
início
  SOMA ← 0
  para I de 1 até 8 passo 1 faça
    leia MD[I]
    SOMA ← SOMA + MD[I]
  fim_para
  MÉDIA ← SOMA / 8
  escreva MÉDIA
fim
  
```

O programa **MÉDIA_GERAL_V2** possui maior mobilidade em relação ao programa **MÉDIA_GERAL_V1**, pois se houver a necessidade de calcular um número maior de médias individuais, basta dimensionar a matriz para o novo tamanho e mudar o valor final da instrução **para...fim_para** de oito para o valor desejado.

No programa **MÉDIA_GERAL_V2** a leitura de cada uma das médias é feita pelo laço de repetição, uma leitura por vez, oito vezes. A matriz é controlada pelo número de cada índice que faz com que cada entrada aconteça em uma posição diferente da outra. Assim sendo, a matriz passa, ao final da execução do laço de repetição, a possuir armazenadas todas as médias de cada um dos oito alunos.

Tabela da matriz: MD	
Índice	Elemento
1	4,5
2	6,5
3	8,0
4	3,5
5	6,0
6	6,5
7	6,5
8	6,0

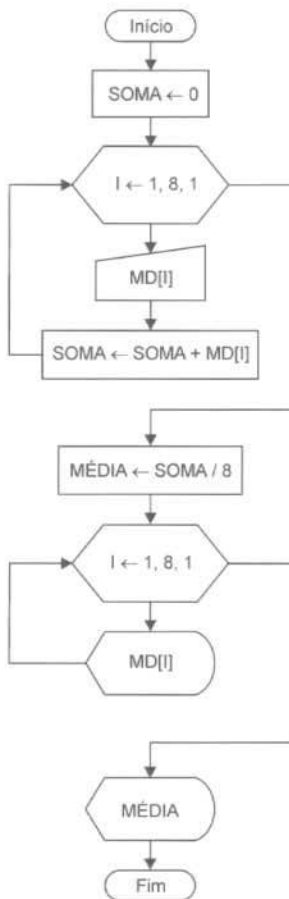
A tabela da matriz **MD** mostrou como ficam os valores armazenados na matriz na memória principal de um computador.

Não confunda índice com elemento. Índice é o endereço, o local de armazenamento de uma matriz, enquanto elemento é o conteúdo, o dado armazenado em uma determinada posição da matriz.

6.2.2 - Escrita dos Dados de uma Matriz

O processo de escrita dos dados de uma matriz é bastante parecido com o processo de leitura desses dados. Em seguida são apresentados o diagrama de blocos, Figura 6.3, e a codificação em português estruturado da escrita das médias individuais dos oito alunos antes de mostrar o resultado do cálculo da média geral.

Diagramação



Codificação

```

programa MÉDIA_GERAL_V3
var
  I : inteiro
  MD : conjunto[1..8] de real
  SOMA, MÉDIA : real
inicio
  SOMA ← 0
  para I de 1 até 8 passo 1 faça
    leia MD[I]
    SOMA ← SOMA + MD[I]
  fim_para
  MÉDIA ← SOMA / 8
  para I de 1 até 8 passo 1 faça
    escreva MD[I]
  fim_para
  escreva MÉDIA
fim
  
```

Figura 6.3 - Diagrama de blocos para escrita dos elementos de uma matriz do tipo vetor.

6.3 - Exercício de Aprendizagem

Para demonstrar a utilização de matrizes de uma dimensão em um exemplo maior, considere os problemas apresentados em seguida. Um exemplo destaca a manipulação dos índices de uma matriz e o outro a manipulação dos elementos de uma matriz.

1º Exemplo

Desenvolver um programa que leia dez elementos numéricos reais de uma matriz A do tipo vetor. Construir uma matriz B de mesmo tipo, observando a seguinte lei de formação: se o valor do índice da matriz A for par, o valor deve ser multiplicado por 5; sendo ímpar, deve ser somado com 5. Ao final, mostrar o conteúdo da matriz A e B.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 6.4.

Entendimento

Este exemplo de resolução mostra como fazer o tratamento da condição do índice.

1. Iniciar o contador de índice, variável I como 1, estendendo essa contagem até 10.
2. Ler os dez valores, um a um.
3. Verificar se o índice da matriz A é par; se sim, multiplicar o valor por 5; se não, somar 5 ao valor. Criar a matriz B e atribuir a ela os valores da matriz A devidamente calculados.
4. Apresentar o conteúdo da matriz B.

No processamento do programa pergunta-se se o valor do índice I em um determinado momento é par (ele será par quando, dividido por 2, obtiver um quociente inteiro e o resto igual a zero). Sendo a condição verdadeira, será implicada na matriz B[I] a multiplicação do elemento da matriz A[I] por 5. Caso o valor do índice I seja ímpar, será implicada na matriz B[I] a soma do elemento da matriz A[I] por 5.

2º Exemplo

Desenvolver um programa que leia cinco elementos numéricos inteiros de uma matriz A do tipo vetor. No final, apresentar o total da soma de todos os elementos da matriz A que sejam ímpares.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 6.5.

Entendimento

O primeiro exemplo pedia para verificar se o índice era par ou ímpar. Este solicita que se analise a condição do elemento e não do índice. Já foi alertado anteriormente para não confundir elemento com índice. Veja a solução:

1. Iniciar o contador de índice, variável I como 1, estendendo essa contagem até 5.
2. Ler os cinco valores, um a um.
3. Verificar se o elemento é ímpar; se sim, efetuar a soma dos elementos.
4. Apresentar o total de todos os elementos ímpares da matriz.

Representação gráfica do 1º exemplo

Diagramação

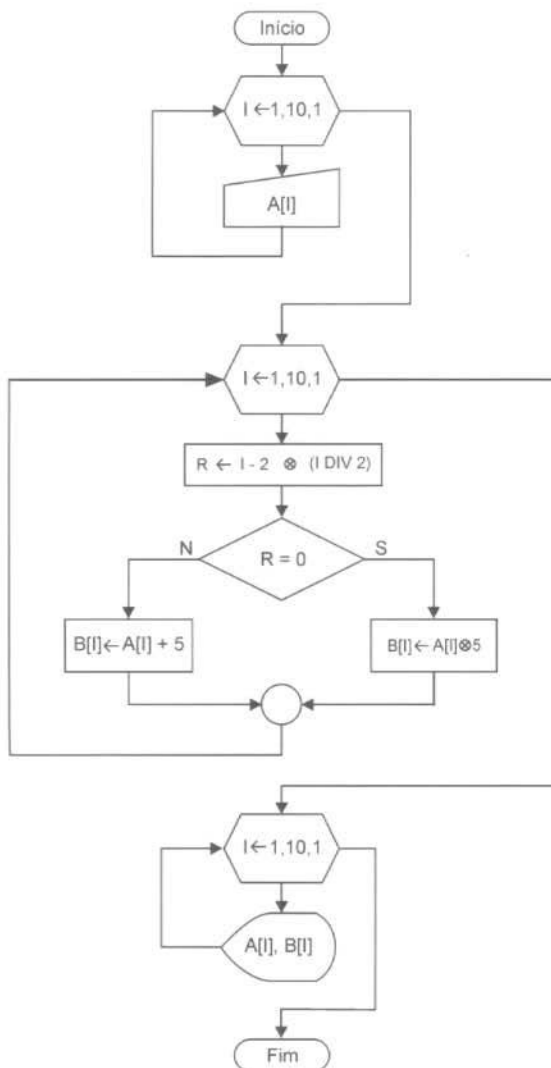


Figura 6.4 - Diagrama de blocos para o primeiro exemplo.

Codificação

```

programa ÍNDICE_PAR_OU_IMPAR
var
  I, R : inteiro
  A, B : conjunto[1..10] de real
início
  para I de 1 até 10 passo 1 faça
    leia A[I]
  fim_para
  para I de 1 até 10 passo 1 faça
    R ← I - 2 * (I DIV 2)
    se (R = 0) então
      B[I] ← A[I] * 5
    senão
      B[I] ← A[I] + 5
    fim_se
  fim_para
  para I de 1 até 10 passo 1 faça
    escreva A[I], B[I]
  fim_para
fim
  
```

Representação gráfica do 2º exemplo

Diagramação

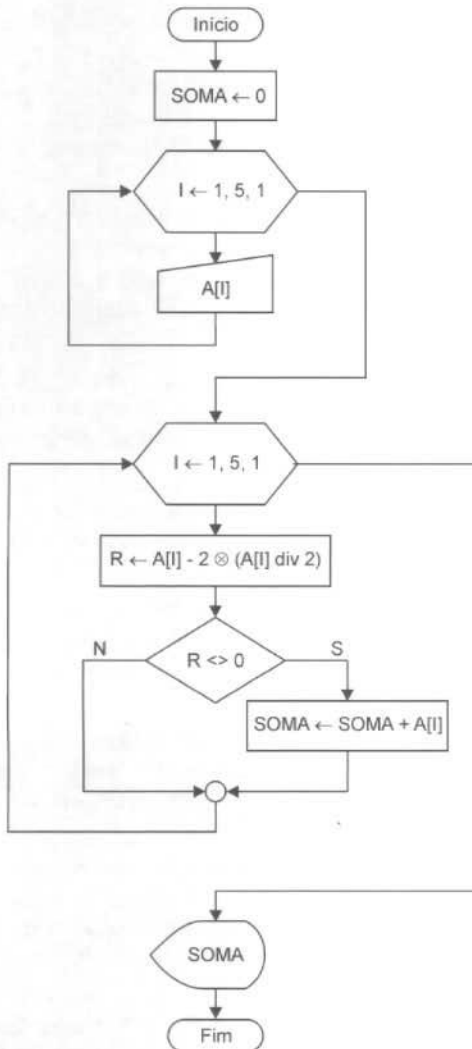


Figura 6.5 - Diagrama de blocos para o segundo exemplo.

Codificação

```

programa ELEMENTO_IMPAR
var
  R, I, SOMA : inteiro
  A : conjunto[1..5] de inteiro
inicio
  SOMA ← 0
  para I de 1 até 5 passo 1 faça
    leia A[I]
  fim_para
  para I de 1 até 5 passo 1 faça
    R ← A[I] - 2 * (A[I] div 2)
    se (R <> 0) então
      SOMA ← SOMA + A[I]
    fim_se
  fim_para
  escreva SOMA
fim
  
```

Quando se faz menção ao índice, indica-se a variável que controla o contador de índice; no caso do exemplo anterior, a variável I. Quando se faz menção ao elemento, indica-se A[I], pois desta forma pega-se o valor armazenado e não a sua posição de endereço.

6.4 - Exercícios de Fixação

1. Desenvolver os diagramas de bloco e a codificação em português estruturado dos problemas computacionais seguintes:
 - a) Elaborar um programa que efetue a leitura de dez nomes de pessoas em uma matriz A do tipo vetor e apresentá-los em seguida.
 - b) Elaborar um programa que leia oito elementos inteiros em uma matriz A do tipo vetor. Construir uma matriz B de mesma dimensão com os elementos da matriz A multiplicados por 3. O elemento B[1] deve ser implicado pelo elemento A[1] * 3, o elemento B[2] implicado pelo elemento A[2] * 3 e assim por diante, até 8. Apresentar a matriz B.
 - c) Escrever um programa que leia duas matrizes (denominadas A e B) do tipo vetor com 20 elementos reais. Construir uma matriz C, sendo cada elemento da matriz C a subtração de um elemento correspondente da matriz A com um elemento correspondente da matriz B, ou seja, a operação de processamento deve estar baseada na operação $C[i] \leftarrow A[i] - B[i]$. Ao final, apresentar os elementos da matriz C.
 - d) Elaborar um programa que leia 15 elementos inteiros de uma matriz A do tipo vetor. Construir uma matriz B de mesmo tipo, observando a seguinte lei de formação: "todo elemento da matriz B deve ser o quadrado do elemento da matriz A correspondente". Apresentar os elementos das matrizes A e B.
 - e) Elaborar um programa que leia uma matriz A do tipo vetor com 15 elementos inteiros. Construir uma matriz B de mesmo tipo, e cada elemento da matriz B deve ser o resultado da fatorial correspondente de cada elemento da matriz A. Apresentar as matrizes A e B.
 - f) Construir um programa que leia duas matrizes A e B do tipo vetor com 15 elementos quaisquer inteiros. Construir uma matriz C, sendo esta o resultado da junção das matrizes A e B. Desta forma, a matriz C deve ter o dobro de elementos em relação às matrizes A e B, ou seja, a matriz C deve possuir 30 elementos. Apresentar a matriz C.
 - g) Elaborar um programa que leia duas matrizes do tipo vetor para o armazenamento de nomes de pessoas, sendo a matriz A com 20 elementos e a matriz B com 30 elementos. Construir uma matriz C, sendo esta a junção das matrizes A e B. Desta forma, a matriz C deve ter a capacidade de armazenar 50 elementos. Apresentar os elementos da matriz C.
 - h) Elaborar um programa que leia 20 elementos do tipo real em uma matriz A unidimensional e construir uma matriz B de mesma dimensão com os mesmos elementos armazenados na matriz A, porém de forma invertida. Ou seja, o primeiro elemento da matriz A passa a ser o último da matriz B, o segundo elemento da matriz A passa a ser o penúltimo da matriz B e assim por diante. Apresentar os elementos das matrizes A e B.
 - i) Escrever um programa que leia três matrizes (A, B e C) de uma dimensão do tipo vetor com cinco elementos cada que sejam do tipo real. Construir uma matriz D, sendo esta o resultado da junção das três matrizes (A, B e C). Desta forma, a matriz D deve ter o triplo de elementos das matrizes A, B e C, ou seja, 15 elementos. Apresentar os elementos da matriz D.
 - j) Elaborar um programa que leia uma matriz A do tipo vetor com 20 elementos inteiros. Construir uma matriz B do mesmo tipo e dimensão da matriz A, sendo cada elemento da matriz B o somatório de 1 até o valor do elemento correspondente armazenado na matriz A. Se o valor do elemento da matriz A[1] for 5, o elemento correspondente da matriz B[1] deve ser 15, pois o somatório do elemento da matriz A é $1+2+3+4+5$. Apresentar os elementos da matriz B.

- k) Elaborar um programa que leia uma matriz A do tipo vetor com dez elementos inteiros positivos. Construir uma matriz B de mesmo tipo e dimensão, em que cada elemento da matriz B deve ser o valor negativo do elemento correspondente da matriz A. Desta forma, se em A[1] estiver armazenado o elemento 8, deve estar em B[1] o valor -8 e assim por diante. Apresentar os elementos da matriz B.
- l) Elaborar um programa que leia uma matriz A do tipo vetor com dez elementos inteiros. Construir uma matriz B de mesmo tipo, em que cada elemento deve ser a metade exata de cada um dos elementos existentes da matriz A. Apresentar os elementos das matrizes A e B.
- m) Construir um programa que calcule a tabuada de um valor qualquer de 1 até 10 e armazene os resultados em uma matriz A de uma dimensão. Apresentar os elementos da matriz A.
- n) Elaborar um programa que leia 20 elementos (valores reais) para temperaturas em graus Celsius e armazene esses valores em uma matriz A de uma dimensão. O programa ao final deve apresentar a menor, a maior e a média das temperaturas lidas.
- o) Escrever um programa que leia 25 elementos (valores reais) para temperaturas em graus Celsius e armazene esses valores em uma matriz A de uma dimensão do tipo vetor. Construir uma matriz B de mesmo tipo e dimensão, em que cada elemento da matriz B deve ser a conversão da temperatura em graus Fahrenheit do elemento correspondente da matriz A. Apresentar os elementos das matrizes A e B.
- p) Elaborar um programa que leia 12 elementos inteiros para uma matriz A de uma dimensão do tipo vetor. Construir uma matriz B de mesmo tipo e dimensão, observando a seguinte lei de formação: "todo elemento da matriz A que for ímpar deve ser multiplicado por 2; caso contrário, o elemento da matriz A deve permanecer constante". Apresentar os elementos da matriz B.
- q) Elaborar um programa que leia 15 elementos reais para uma matriz A de uma dimensão do tipo vetor. Construir uma matriz B de mesmo tipo e dimensão, observando a seguinte lei de formação: "todo elemento da matriz A que possuir índice par deve ter seu elemento dividido por 2; caso contrário, o elemento da matriz A deve ser multiplicado por 1.5". Apresentar os elementos da matriz B.
- r) Elaborar um programa que leia seis elementos (valores inteiros) para as matrizes A e B de uma dimensão do tipo vetor. Construir as matrizes C e D de mesmo tipo e dimensão. A matriz C deve ser formada pelos elementos de índice ímpar das matrizes A e B e a matriz D deve ser formada pelos elementos de índice par das matrizes A e B. Apresentar os elementos das matrizes C e D.
- s) Elaborar um programa que leia duas matrizes A e B de uma dimensão com seis elementos. A matriz A deve aceitar apenas a entrada de valores pares, enquanto a matriz B deve aceitar apenas a entrada de valores ímpares. A entrada das matrizes deve ser validada pelo programa e não pelo usuário. Construir uma matriz C que seja o resultado da junção das matrizes A e B, de modo que a matriz C contenha 12 elementos. Apresentar os elementos da matriz C.
- t) Escrever um programa que leia duas matrizes A e B de uma dimensão com dez elementos. A matriz A deve aceitar apenas a entrada de valores divisíveis por 2 e 3, enquanto a matriz B deve aceitar apenas a entrada de valores múltiplos de 5. A entrada das matrizes deve ser validada pelo programa e não pelo usuário. Construir uma matriz C que seja o resultado da junção das matrizes A e B, de modo que contenha 20 elementos. Apresentar os elementos da matriz C.
- u) Elaborar um programa que leia duas matrizes A e B de uma dimensão com 12 elementos. A matriz A deve aceitar apenas a entrada de valores divisíveis por 2 ou 3, enquanto a matriz B deve aceitar apenas a entrada de valores que não sejam múltiplos de 5. A entrada das matrizes

- deve ser validada pelo programa e não pelo usuário. Construir uma matriz C que seja o resultado da junção das matrizes A e B, de forma que contenha 24 elementos. Apresentar os elementos da matriz C.
- v) Construir um programa que leia uma matriz A de uma dimensão do tipo vetor com 30 elementos do tipo inteiro. Ao final do programa, apresentar a quantidade de valores pares e ímpares existentes na referida matriz.
 - w) Elaborar um programa que leia duas matrizes A e B de uma dimensão do tipo vetor com dez elementos inteiros cada. Construir uma matriz C de mesmo tipo e dimensão que seja formada pelo quadrado da soma dos elementos correspondentes nas matrizes A e B. Apresentar os elementos da matriz C.
 - x) Elaborar um programa que leia uma matriz A de uma dimensão com seis elementos do tipo real. Construir uma matriz B, em que cada posição de índice ímpar da matriz B deve ser atribuída com um elemento de índice par existente na matriz A e cada posição de índice par da matriz B deve ser atribuída com um elemento de índice ímpar existente na matriz A. Apresentar os elementos das duas matrizes.
 - y) Escrever um programa que leia uma matriz A de uma dimensão com 15 elementos numéricos inteiros. Apresentar o total de elementos pares existentes na matriz.
 - z) Elaborar um programa que leia uma matriz A de uma dimensão com dez elementos numéricos inteiros. Apresentar o total de elementos ímpares existentes na matriz e também o percentual do valor total de números ímpares em relação à quantidade total de elementos armazenados na matriz.

7

Aplicações Básicas com Matrizes de Uma Dimensão

A utilização de matrizes em programação é bastante ampla, que vai desde o uso direto, quando o programador constrói a estrutura de dados e faz o seu controle operacional, até o uso indireto, quando o programador usa um sistema de gerenciamento de banco de dados para auxiliar a construção e controle das operações em tabelas de dados.

O capítulo anterior apresentou uma técnica de programação que possibilita o uso e o armazenamento de um grande conjunto de dados em uma única variável de estrutura composta, denominada matriz.

Este capítulo traz exemplos práticos da técnica de programação baseada no uso de matrizes de uma dimensão. Mostra três algoritmos clássicos, muito utilizados, um de classificação de elementos e outros dois de pesquisa de elementos.

7.1 - Ser Programador

A partir da segunda metade do século XX e início do século XXI, muitas pessoas passaram a ter muita pressa de realizar seus compromissos diários. Parece que se não agirem assim, o mundo vai acabar, que estão fora de moda, e que esse "vírus" contaminou ou está contaminando a todos, principalmente profissionais da área de desenvolvimento de programas e sistemas para computadores, que acabam esquecendo o lado bom da profissão em detrimento da pressa dos outros, tornando o trabalho de programação algo terrível.

Peter Norvig publicou um artigo em seu sítio pessoal (<http://norvig.com/21-days.html>) que parece ter fomentado a discussão sobre a pressa exacerbada. O artigo intitulado *Teach Yourself Programming in Ten Years* (aprenda a programar em dez anos), dirigido aos profissionais de desenvolvimento de programas e sistemas de computadores, foi traduzido para diversos idiomas, entre eles o português (veja as referências bibliográficas). Nesse artigo Peter Norvig apresenta as consequências da banalização de uma indústria de produção e publicação de livros e revistas sensacionalistas que acabaram por prejudicar, em vez de ajudar, o estado refinado da ciência da computação.

Peter Norvig defende a ideia de que se aprende a programar em longo prazo, não em apenas algumas horas, dias ou meses. Justifica esse apelo com exemplos reais de personalidades conhecidas que levaram dez anos ou mais até atingirem sucesso.

Peter Norvig descreve algumas recomendações a serem observadas pelas pessoas que desejam aprender a programar computadores e ter sucesso nessa empreitada, podendo-se destacar alguns pontos apresentados em seguida:

- ▶ Interesse-se por programação, e faça porque é legal. Tenha certeza de que continue legal para dedicar dez anos a isso, o que é muito tempo. Assegure-se de que gosta de programar, pois descobrir depois de alguns anos que você não gosta é realmente um atraso de vida.
- ▶ *Converse com outros programadores; leia outros programas. Isso é mais importante do que qualquer livro ou curso de treinamento.* Isto posto, não significa jogar fora livros e instruções aprendidas em cursos e treinamentos, mas sim aproveitar a experiência de outros como mais uma fonte de aprendizado.
- ▶ *Trabalhe em projetos com outros programadores. Seja o melhor programador em alguns projetos, seja o pior em outros. Quando você é o melhor, você testa suas habilidades (...).* Quando você é o pior, aprende o que os mestres ensinam e o que não gostam de fazer. Estar envolvido em equipes de projetos faz com que você aprenda a aceitar as ideias de outros e apresente o seu ponto de vista.
- ▶ *Trabalhe em projetos após outros programadores.* Esteja envolvido em entender um programa escrito por outro. Veja o que é preciso para entender e consertar quando o programador original não está por perto. Pense em como desenvolver seu programa para que ele seja fácil para quem for mantê-lo após você. Esse compromisso exige que se aprenda a trabalhar respeitando regras e normas.
- ▶ Aprenda pelo menos meia dúzia de linguagens de programação. Inclua na lista uma linguagem orientada a objetos (como Java ou C++), uma que seja de abstração funcional (como Lisp ou ML), uma que suporte abstração sintática (como Lisp), uma que suporte especificação declarativa (como Prolog ou C++ com templates), uma que suporte corrotinas (como Icon ou Scheme) e uma que suporte paralelismo (como Sisal). Conheça uma linguagem que dê suporte à programação estruturada (como Pascal, C ou C++). O programador deve ser um profissional poliglota. Saber apenas uma ou duas linguagens de programação não lhe proporciona espaço adequado em um mercado de trabalho altamente competitivo.
- ▶ Envolve-se na padronização de uma linguagem, pode ser o comitê ANSI C++, ou na padronização de programação na sua empresa, se utilizaram indentação com dois ou quatro espaços. Em qualquer caso, você aprende o que outras pessoas gostam em uma linguagem, o quanto gostam e por que gostam. A dedicação e o esforço auxiliam na obtenção do grau de disciplina que um desenvolvedor de programação necessita como uma das virtudes indicadas pelo professor Guerreiro e citadas no tópico 4.1 do capítulo 4.

Educação (não importa o âmbito em que esta palavra está inserida) é um processo, e como processo é algo que leva certo tempo para ser aprendido e absorvido. Não tenha pressa em aprender, não seja afobado, siga seu ritmo. Mas não demore muito, pois sempre há alguém prestes a ultrapassá-lo.

7.2 - Classificação de Elementos

Uma das operações mais importantes executadas com matrizes é, sem dúvida, a organização dos dados armazenados. Uma das formas de organizar dados é proceder à classificação (ordenação) nas ordens numérica, alfabética ou alfanumérica, respeitando a ordem estabelecida na tabela ASCII apresentada no tópico 1.1.3 do capítulo 1.

A classificação numérica de dados pode ser efetuada na ordem crescente (do menor valor numérico para o maior) ou na ordem decrescente (do maior valor numérico para o menor).

A classificação alfabética de dados pode ser efetuada na ordem ascendente (de "A" até "Z" ou de "a" até "z") ou na ordem descendente (de "Z" até "A" ou de "z" até "a"). Primeiramente ocorre a ordenação dos caracteres alfabéticos maiúsculos e depois a ordenação dos caracteres minúsculos.

A classificação alfanumérica de dados pode ser feita na ordem ascendente ou descendente. Esse tipo de classificação considera dados numéricos, alfabéticos e demais caracteres gráficos previstos na tabela ASCII.

Para classificar dados em uma matriz de uma dimensão (ou mesmo matrizes com mais dimensões) não há necessidade de um programador desenvolver algoritmos próprios, pois já existe um conjunto de algoritmos para essa finalidade. Basta conhecer e escolher aquele que melhor atende a uma necessidade em específico. Entre as técnicas (os métodos) de programação para classificação de dados existentes, pode-se destacar as categorias (AZEREDO, 1996):

- ▶ Classificação por inserção (método da inserção direta, método da inserção direta com busca binária, método dos incrementos decrescentes - *shellsort*).
- ▶ Classificação por troca (método da bolha - *bubblesort*, método da agitação - *shakesort*, método do pente - *combsort*, método de partição e troca - *quicksort*).
- ▶ Classificação por seleção (método da seleção direta, método da seleção em árvore - *heapsort*, método de seleção em árvore amarrada - *threadedheapsort*).
- ▶ Classificação por distribuição de chaves (método de indexação direta - *radixsort*).
- ▶ Classificação por intercalação (método da intercalação simples - *mergesort*, método de intercalação de seqüências naturais).
- ▶ Classificação por cálculo de endereços (método das listas de colisão, método da solução postergada das colisões).

Dos algoritmos de classificação de dados existentes, o algoritmo apresentado neste livro encontra-se na categoria *classificação por troca* e não consta do trabalho de Azeredo (1996), sendo um método muito simples de classificação de dados. Apesar de eficaz, sua eficiência de velocidade é de certa forma questionável, pois é um método de ordenação lento, sendo útil para ordenar um conjunto pequeno de dados. No entanto, é um método de fácil entendimento e serve de base para entender os demais métodos existentes.

O método de classificação de dados apresentado baseia-se no algoritmo de propriedade distributiva (exercício de fixação 6, g, do capítulo 3) combinado com o algoritmo de troca de valores (permuta) entre variáveis (exercício de fixação 6, f, do capítulo 3), que possibilitou o resultado do exercício de fixação 3, f, do capítulo 4 com variáveis simples. Neste contexto será usado o mesmo método, só que aplicado a variáveis compostas.

A título de ilustração, imagine a necessidade de colocar em ordem crescente cinco valores numéricos inteiros, representados na tabela de valores.

Tabela de valores	
Índice	Elemento
1	9
2	8
3	7
4	5
5	3

Os elementos estão armazenados na ordem 9, 8, 7, 5 e 3, respectivamente, para os índices 1, 2, 3, 4 e 5 e devem ser apresentados na ordem 3, 5, 7, 8 e 9, respectivamente, para os índices 1, 2, 3, 4 e 5. Os elementos da matriz (tabela) devem ser trocados de posição no sentido de apresentarem a ordem desejada. Convertendo a tabela no formato matricial, ter-se-á então:

$$A[1] = 9 \mid A[2] = 8 \mid A[3] = 7 \mid A[4] = 5 \mid A[5] = 3$$

Para o processo de troca, é necessário aplicar o método de propriedade distributiva. O elemento que estiver em $A[1]$ deve ser comparado com os elementos que estiverem em $A[2]$, $A[3]$, $A[4]$ e $A[5]$. Depois, o elemento que estiver em $A[2]$ não precisa ser comparado com o elemento que estiver em $A[1]$, pois foi anteriormente comparado, passando a ser comparado somente com os elementos que estiverem em $A[3]$, $A[4]$ e $A[5]$. Na sequência, o elemento que estiver em $A[3]$ é comparado com os que estiverem em $A[4]$ e $A[5]$ e, por fim, o elemento que estiver em $A[4]$ é comparado com o que estiver em $A[5]$.

Seguindo este raciocínio, basta comparar o valor do elemento armazenado em $A[1]$ com o valor do elemento armazenado em $A[2]$. Se o valor do primeiro elemento for maior que o valor do segundo, efetua-se a troca dos elementos em relação às suas posições de índice. Assim sendo, considere o seguinte:

$$A[1] = 9 \mid A[2] = 8 \mid A[3] = 7 \mid A[4] = 5 \mid A[5] = 3$$

Como a condição de troca é verdadeira, o elemento 9 de $A[1]$ é maior que o elemento 8 de $A[2]$, passa-se para $A[1]$ o elemento 8 e para $A[2]$ passa-se o elemento 9. Desta forma, os valores dentro da matriz passam a ter a seguinte formação:

$$A[1] = 8 \mid A[2] = 9 \mid A[3] = 7 \mid A[4] = 5 \mid A[5] = 3$$

Seguindo a regra de aplicação de propriedade distributiva, o atual valor de $A[1]$ deve ser comparado com o próximo valor após a sua última comparação. Sendo assim, ele deve ser comparado com o valor existente em $A[3]$.

$$A[1] = 8 \mid A[2] = 9 \mid A[3] = 7 \mid A[4] = 5 \mid A[5] = 3$$

O atual valor do elemento de $A[1]$ é maior que o valor do elemento de $A[3]$, ou seja, 8 é maior que 7, efetua-se então a sua troca, ficando $A[1]$ com o elemento de valor 7 e $A[3]$ com o elemento de valor 8. Os valores da matriz passam a ter a seguinte formação:

$$A[1] = 7 \mid A[2] = 9 \mid A[3] = 8 \mid A[4] = 5 \mid A[5] = 3$$

Agora devem ser comparados os valores dos elementos armazenados nas posições $A[1]$ e $A[4]$.

$$A[1] = 7 \mid A[2] = 9 \mid A[3] = 8 \mid A[4] = 5 \mid A[5] = 3$$

O elemento 7 de $A[1]$ é maior que o elemento 5 de $A[4]$. Eles são trocados, passando $A[1]$ a possuir o elemento 5 e $A[4]$ a possuir o elemento 7. A matriz passa a ter a seguinte formação:

$$A[1] = 5 \mid A[2] = 9 \mid A[3] = 8 \mid A[4] = 7 \mid A[5] = 3$$

Os elementos comparados foram trocados de posição, estando agora em $A[1]$ o elemento de valor 5. Devem ser comparados os valores dos elementos armazenados nas posições $A[1]$ e $A[5]$.

$$A[1] = 5 \mid A[2] = 9 \mid A[3] = 8 \mid A[4] = 7 \mid A[5] = 3$$

O elemento 5 de $A[1]$ é comparado como o elemento 3 de $A[5]$. Como $A[1]$ possui valor de elemento maior que o valor de elemento de $A[5]$, os seus valores são trocados. Desta forma, a matriz passa a ter a seguinte formação:

$$A[1] = 3 \mid A[2] = 9 \mid A[3] = 8 \mid A[4] = 7 \mid A[5] = 5$$

A partir deste ponto, o elemento de valor 3 armazenado em $A[1]$ não precisa mais ser comparado com os demais elementos. É necessário pegar o atual valor do elemento da posição $A[2]$ e efetuar sucessivamente sua comparação com os outros elementos restantes. Desta forma, o valor do elemento armazenado em $A[2]$ deve ser comparado com os elementos armazenados em $A[3]$, $A[4]$ e $A[5]$, segundo a regra da aplicação de propriedade distributiva. Agora devem ser comparados os valores dos elementos armazenados nas posições $A[2]$ e $A[3]$.

$$A[1] = 3 \mid A[2] = 9 \mid A[3] = 8 \mid A[4] = 7 \mid A[5] = 5$$

Comparando o elemento 9 da posição $A[2]$ com o elemento 8 da posição $A[3]$ e sendo essa condição verdadeira, efetua-se a troca de forma que o elemento 8 esteja em $A[2]$ e o elemento 9 esteja em $A[3]$. A matriz passa a ter a seguinte formação:

$$A[1] = 3 \mid A[2] = 8 \mid A[3] = 9 \mid A[4] = 7 \mid A[5] = 5$$

Em seguida, o atual valor do elemento de $A[2]$ que é 8 deve ser comparado com o valor do elemento de $A[4]$ que é 7.

$$A[1] = 3 \mid A[2] = 8 \mid A[3] = 9 \mid A[4] = 7 \mid A[5] = 5$$

Pelo fato de o elemento 8 de $A[2]$ ser maior que o elemento 7 de $A[4]$, eles são trocados, ficando $A[2]$ com o elemento 7 e $A[4]$ com o elemento 8. Desta forma, a matriz passa a ter a seguinte formação:

$$A[1] = 3 \mid A[2] = 7 \mid A[3] = 9 \mid A[4] = 8 \mid A[5] = 5$$

Então continua-se o processo de comparação e troca. O atual valor do elemento na posição $A[2]$ é 7 e será comparado com o valor do elemento $A[5]$ que é 5.

$$A[1] = 3 \mid A[2] = 7 \mid A[3] = 9 \mid A[4] = 8 \mid A[5] = 5$$

Por ser a condição verdadeira, são trocados. A posição $A[2]$ fica com o elemento 5 e a posição $A[5]$ fica com o elemento 7, conforme indicado no esquema:

$$A[1] = 3 \mid A[2] = 5 \mid A[3] = 9 \mid A[4] = 8 \mid A[5] = 7$$

Até este ponto $A[2]$ foi comparada com todas as posições subsequentes, não tendo mais nenhuma comparação para ela. Agora se efetua a comparação da próxima posição com o restante, no caso, de $A[3]$ com $A[4]$ e $A[5]$. O elemento 9 da posição $A[3]$ será comparado com o elemento 8 da posição $A[4]$.

$$A[1] = 3 \mid A[2] = 5 \mid A[3] = 9 \mid A[4] = 8 \mid A[5] = 7$$

Por ser a condição verdadeira, são trocados. A posição $A[3]$ fica com o elemento 8 e a posição $A[4]$ fica com o elemento 9, conforme indicado no esquema:

$$A[1] = 3 \mid A[2] = 5 \mid A[3] = 8 \mid A[4] = 9 \mid A[5] = 7$$

A seguir, compara-se o elemento 8 da posição $A[3]$ com o elemento 7 da posição $A[5]$, como indicado:

$$A[1] = 3 \mid A[2] = 5 \mid A[3] = 8 \mid A[4] = 9 \mid A[5] = 7$$

Por ser o valor do elemento 8 maior que o valor do elemento 7, ocorre a troca. Desta forma, A[3] passa a ter o elemento 7 e A[5] passa a ter o elemento 8, como indicado em seguida:

$$A[1] = 3 \mid A[2] = 5 \mid \mathbf{A[3] = 7} \mid A[4] = 9 \mid \mathbf{A[5] = 8}$$

Efetuada todas as comparações de A[3] com A[4] e A[5], fica disponível apenas a última comparação que é A[4] com A[5], em que o elemento 9 de A[4] será comparado com o elemento 8 de A[5].

$$A[1] = 3 \mid A[2] = 5 \mid A[3] = 7 \mid \mathbf{A[4] = 9} \mid \mathbf{A[5] = 8}$$

Pelo fato de o elemento 9 de A[4] ser maior que o elemento 8 de A[5] eles são trocados, passando A[4] a possuir o elemento 8 e A[5] a possuir o elemento 9, como mostrado em seguida:

$$A[1] = 3 \mid A[2] = 5 \mid A[3] = 7 \mid \mathbf{A[4] = 8} \mid \mathbf{A[5] = 9}$$

Pode-se notar que a referida classificação foi feita. Apresentando os elementos da matriz em ordem crescente, ter-se-á a seguinte situação:

$$A[1] = \mathbf{3} \mid A[2] = \mathbf{5} \mid A[3] = \mathbf{7} \mid A[4] = \mathbf{8} \mid A[5] = \mathbf{9}$$

Para dados do tipo caractere ou cadeia o processo de classificação é idêntico, uma vez que cada letra possui um valor diferente. Por exemplo, a letra "A" tem valor menor que a letra "B" e assim por diante. Se a letra "A" maiúscula for comparada com a letra "a" minúscula, elas terão valores diferentes, como pode ser constatado na tabela ASCII.

A seguir, como exemplo é apresentado um programa completo com a parte da entrada de dados na matriz, o processamento da ordenação e a apresentação dos dados ordenados.

Elaborar um programa que leia os nomes de 20 pessoas em uma variável composta, processar a ordenação ascendente desses nomes e apresentar a listagem dos nomes em ordem alfabética.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 7.1.

Entendimento

Com base na necessidade de ordenar 20 nomes em uma matriz de uma dimensão do tipo vetor, deve-se observar os seguintes passos, dando maior atenção ao quarto passo que deve ser implementado de acordo com o exposto anteriormente.

1. Definir uma variável do tipo inteiro para controlar a execução do laço de repetição.
2. Definir a variável matriz NOME do tipo cadeia para acesso a 20 elementos.
3. Iniciar o programa e efetuar a entrada dos 20 nomes.
4. Colocar em ordem ascendente os elementos da matriz.
5. Apresentar os 20 nomes que devem estar classificados.

A Figura 7.1 exibe o diagrama de blocos que representa a entrada dos 20 nomes, o processamento da ordenação (classificação ascendente) dos nomes e a apresentação dos nomes de forma ordenada.

O trecho do diagrama de blocos que faz o processamento da ordenação possui dois laços de repetição encadeados para procederem ao funcionamento da aplicação de propriedade distributiva.

Note o uso do segundo laço de repetição de uma segunda variável para controlar o índice subsequente no processo de ordenação, no caso a variável *J*. Atente para o fato de a variável *I* ser inicializada com valor numérico 1, estendendo-se até o valor numérico 19, e a variável *J* (encadeada à variável *I*) ser inicializada com o valor numérico *I* + 1, estendendo-se até o valor 20. Isso implica na seguinte seqüência de valores ao longo da execução do trecho de programa que opera a classificação dos elementos da matriz NOME:

Diagramação

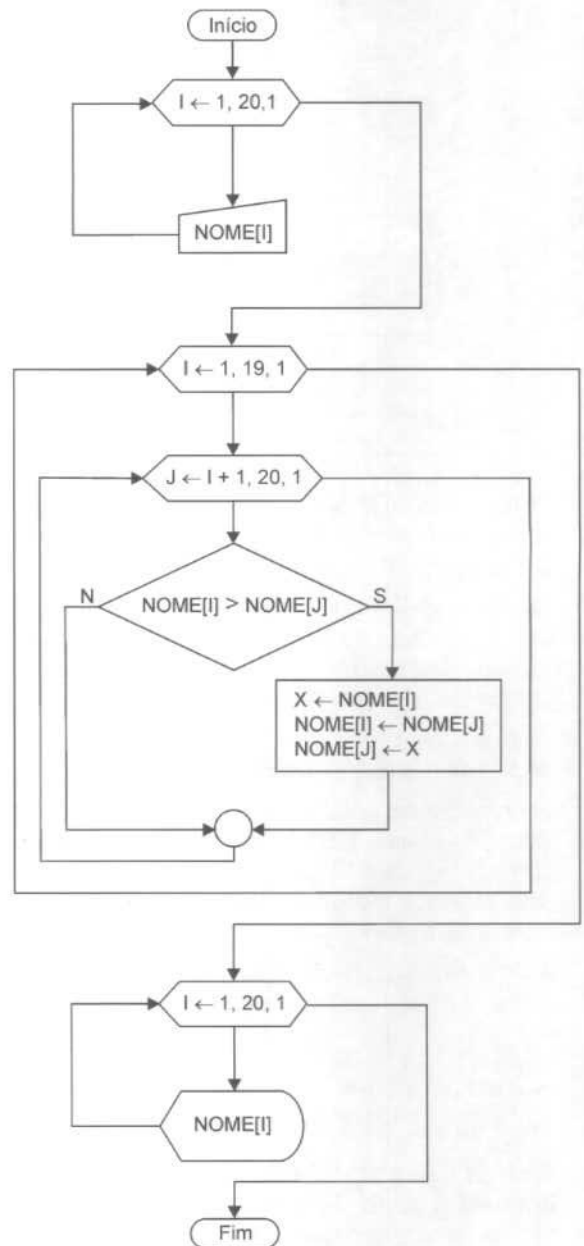


Figura 7.1 - Diagrama de blocos para a classificação dos nomes.

Tabela de valores das variáveis I e J	
Quando I for	J será
1	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
2	3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
3	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
4	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
5	6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

15	16, 17, 18, 19, 20
16	17, 18, 19, 20
17	18, 19, 20
18	19, 20
19	20

Somente quando a variável **J** atinge o valor 20 é que seu laço de repetição se encerra, retornando ao laço de repetição da variável **I**, acrescentando o valor um à variável **I** até que a variável **I** atinja o seu limite e ambos os laços de repetição sejam encerrados.

Quando a variável **I** for 1, a variável **J** será 2 e contará até 20. Ao final desse ciclo, a variável **I** é acrescida de 1, tornando-se 2; assim sendo, a variável **J** passa a ter o valor 3. Quando a variável **J** voltar a ser 20, a variável **I** passa a possuir o valor 3 e a variável **J** o valor 4. Este ciclo será executado até que, por fim, a variável **I** tenha o valor 19 e a variável **J** tenha o valor 2.

Outro detalhe a ser observado é a utilização do algoritmo de troca de valores na instrução de tomada de decisão **se NOME[I] > NOME[J] então**.

Por exemplo, considere o elemento **NOME[I]** com o valor cadeia "CARLOS" e o elemento **NOME[J]** com o valor cadeia "ALBERTO". Após verificar a condição pela tomada de decisão, o elemento **NOME[I]** deve estar com o valor cadeia "ALBERTO" e o elemento **NOME[J]** com o valor cadeia "CARLOS". Lembre-se de que, para conseguir efetuar a troca de valores, é necessário uma variável simples de apoio, a qual está representada pela variável **X**.

Após a verificação da condição como verdadeira, ou seja, sendo o primeiro nome maior que o segundo, efetua-se então a sua troca com o algoritmo:

```

X ← NOME[I]
NOME[I] ← NOME[J]
NOME[J] ← X

```

Para que o elemento **NOME[I]** fique livre para receber o valor do elemento **NOME[J]**, a variável **X** deve ser implicada pelo valor do elemento **NOME[I]**. A variável **X** passa a possuir o valor cadeia "CARLOS". Em seguida, implica-se o valor do elemento **NOME[J]** no elemento **NOME[I]**. Assim sendo, o elemento **NOME[I]** passa a possuir o valor cadeia "ALBERTO". Na sequência o elemento **NOME[J]** é implicado pelo valor cadeia que está armazenado na variável **X**. Devido ao mecanismo do algoritmo de troca ao final, ter-se-á **NOME[I]** com "ALBERTO" e **NOME[J]** com "CARLOS".

Codificação

Observe em seguida a codificação do programa de ordenação de valores em português estruturado. Atente para o detalhe das instruções de laço de repetição utilizadas pelo comando **para** em formato encadeado na fase de processamento da ordenação. Lembre-se de que, no caso de encadeamento, será executado primeiramente o laço de repetição mais interno, no caso o trecho de contagem da variável **J**, passando o processamento para a rotina mais externa controlada pela variável **I**, quando o trecho mais interno fecha o seu ciclo operacional.

```
programa LISTA_NOME_ORDENADA
var
  I, J : inteiro
  NOME : conjunto[1..20] de cadeia
  X : cadeia
início

  {Trecho de entrada de dados}

  para I de 1 até 20 passo 1 faça
    leia NOME[I]
  fim_para

  {Trecho de processamento de ordenação}

  para I de 1 até 19 passo 1 faça
    para J de I + 1 até 20 passo 1 faça
      se (NOME[I] > NOME[J]) então
        X ← NOME[I]
        NOME[I] ← NOME[J]
        NOME[J] ← X
      fim_se
    fim_para
  fim_para

  {Trecho de saída com dados ordenados}

  para I de 1 até 20 passo 1 faça
    escreva NOME[I]
  fim_para

fim
```

Um detalhe deste programa exemplo são os comentários de ilustração colocados entre chaves. Comentários desse tipo servem para auxiliar a documentação interna de código de um programa, facilitando a interpretação de um determinado trecho.

São utilizados três trechos de comentários indicando as etapas de ação de um computador (entrada, processamento e saída). Note que cada trecho sinalizado do código de programa se enquadra com os trechos equivalentes no diagrama de blocos da Figura 7.1.

7.3 - Métodos de Pesquisa de Elementos

As matrizes permitem o uso de grandes tabelas de valores. Quanto maior for a tabela, maior a dificuldade de localizar um elemento na matriz de forma rápida. Imagine uma matriz com 400.000 elementos (400.000 nomes de pessoas). Será que você conseguiria encontrar rapidamente um elemento de forma manual? Dependendo da posição em que o elemento estiver, certamente não. Para solucionar este problema existem algoritmos

específicos, entre os quais se destacam os métodos de pesquisa linear, binário, árvore binária, largura, profundidade, entre vários outros existentes. Nesta obra são apresentados os algoritmos de pesquisa sequencial (linear) e de pesquisa binária.

7.3.1 - Pesquisa Sequencial

Esse método consiste em buscar a informação desejada a partir do primeiro elemento, avançando sequencialmente cada posição da matriz até chegar ao último elemento. Se ao longo da pesquisa ocorrer a localização da informação, ela é apresentada. Caso chegue ao final da matriz sem localizar o elemento, indica que ele não está armazenado na matriz.

Esse método de pesquisa é lento, porém eficiente nos casos em que uma matriz encontra-se com os elementos desordenados. Para exemplificar a utilização desse tipo de pesquisa, imagine a tabela ao lado.

A tabela em questão representa uma matriz unidimensional com sete elementos do tipo **cadeia**. Deseja-se fazer uma pesquisa de um dos elementos. No caso, será escolhido o elemento **Frederico**, que se encontra na posição de índice 3.

Índice	Nomes
1	André
2	Carlos
3	Frederico
4	Golias
5	Silvia
6	Silvio
7	Waldir

O processo de pesquisa sequencial consiste em verificar se o conteúdo da posição de índice 1 é igual ao conteúdo pesquisado, ou seja, se "André" é igual a "Frederico". Se o resultado for verdadeiro, a pesquisa é encerrada, pois o conteúdo foi localizado. Se não, posiciona-se no próximo índice da tabela e efetua-se nova verificação. Num segundo momento o conteúdo "Carlos" da posição de índice 2 é verificado com o conteúdo de pesquisa "Frederico". Se o resultado dessa verificação for verdadeiro, a pesquisa é encerrada; caso não seja, o processamento de pesquisa é repetido sequencialmente até chegar ao fim da matriz, se este for o caso. No caso da situação exposta o processamento de pesquisa será encerrado quando a verificação chegar ao índice 3 da tabela.

No sentido de demonstrar a técnica de pesquisa sequencial, considere o exemplo seguinte:

Elaborar um programa que leia dez nomes e apresente pelo método de pesquisa sequencial os nomes que porventura estejam armazenados na matriz e que coincidam com o nome de entrada de pesquisa. Além de apresentar o nome, o programa deve indicar em que posição da matriz ele está armazenado. Caso o nome pesquisado não seja encontrado, deve informar que o nome pesquisado não foi localizado.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 7.2.

Entendimento

O algoritmo indicado em seguida descreve a entrada de dez nomes e a apresentação de nomes que podem ser solicitados na fase de pesquisa sequencial.

1. Iniciar um laço de repetição e pedir a leitura de dez nomes.
2. Criar um laço de repetição que faça a pesquisa sequencial enquanto o usuário desejar. Na fase de pesquisa deve ser solicitada a informação a ser pesquisada, a qual deve ser comparada com o primeiro elemento. Sendo igual, mostra; caso contrário, avança para o próximo. Se não achar em toda lista o conteúdo pesquisado, informar que não existe o elemento pesquisado; se existir, deve ser mostrado.
3. Encerrar a pesquisa e o programa, quando desejado.

Diagramação

O diagrama de blocos da Figura 7.2 apresenta apenas o trecho do processamento da pesquisa sequencial. O trecho referente à entrada de dados foi omitido por já ser conhecido.

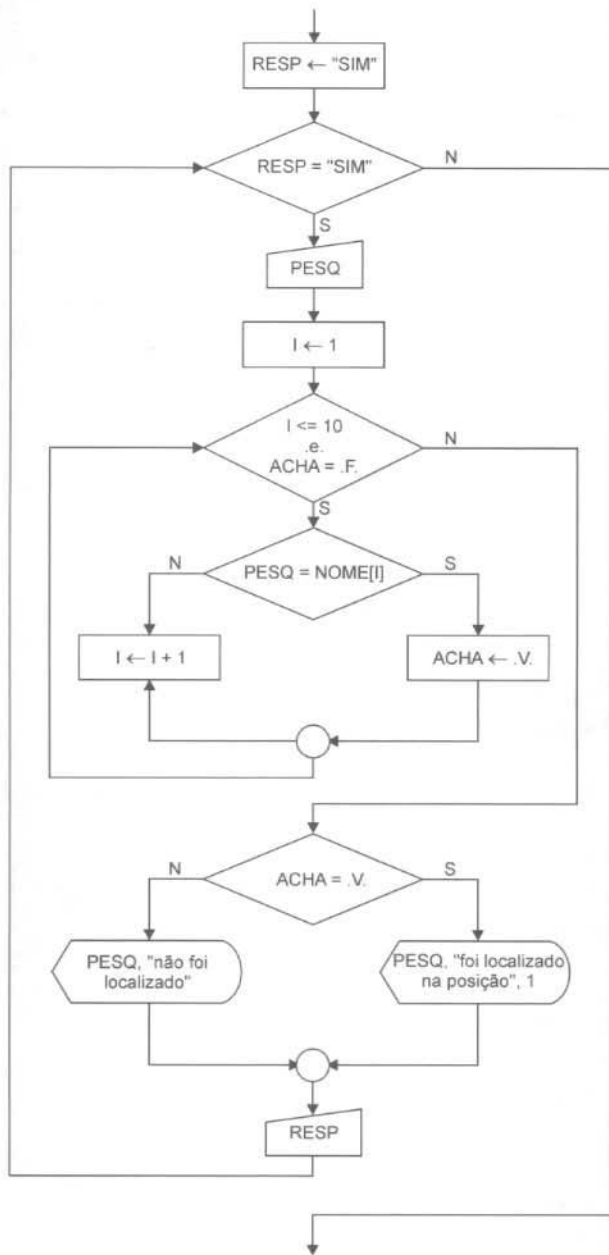


Figura 7.2 - Diagrama de blocos para pesquisa sequencial.

Codificação

O programa seguinte demonstra o método de pesquisa sequencial em um contexto prático.

```

programa PESQUISA_SEQÜENCIAL

var
  NOME : conjunto[1..10] de cadeia
  I : inteiro
  PESQ, RESP : cadeia
  ACHA : lógico
inicio

  para I de 1 até 10 passo 1 faça
    leia NOME[I]
  fim_para

  (** inicio do trecho de pesquisa sequencial **)

  RESP ← "SIM"
  enquanto (RESP = "SIM") faça
    escreva "Entre o nome a ser pesquisado: "
    leia PESQ
    I ← 1
    ACHA ← .Falso.
    enquanto (I <= 10) .e. (ACHA = .Falso.) faça
      se (PESQ = NOME[I]) então
        ACHA ← .Verdadeiro.
      senão
        I ← I + 1
      fim_se
    fim_enquanto
    se (ACHA = .Verdadeiro.) então
      escreva PESQ, " foi localizado na posição ", I
    senão
      escreva PESQ, " não foi localizado"
    fim_se
    escreva "Deseja continuar? (SIM/NÃO): "
    leia RESP
  fim_enquanto

  (** fim do trecho de pesquisa sequencial **)

fim

```

Anteriormente foi montada a rotina de pesquisa empregada em um contexto. Observe o trecho seguinte que faz a pesquisa com seus comentários:

```

1 - leia PESQ
2 - I ← 1
3 - ACHA ← .Falso.
4 - enquanto (I <= 10) .e. (ACHA = .Falso.) faça
5 -   se (PESQ = NOME[I]) então
6 -     ACHA ← .Verdadeiro.
7 -   senão
8 -     I ← I + 1
9 -   fim_se
10 - fim_enquanto
11 - se (ACHA = .Verdadeiro.) então

```

```
12 - escreva PESQ, "foi localizado na posição", I
13 - senão
14 - escreva PESQ, "não foi localizado"
15 - fim_se
```

Na linha 1, é solicitado o nome a ser pesquisado na variável PESQ, em seguida, na linha 2, é setado o valor do contador de índice como 1 e na linha 3, a variável **ACHA** é setada como tendo um valor falso. A linha 4 apresenta a instrução **enquanto** indicando que, enquanto o valor da variável I for menor ou igual a 10 e, simultaneamente, o valor da variável **ACHA** for falso, deve ser processado o conjunto de instruções situadas nas linhas 5, 6, 7, 8 e 9.

Neste ponto, a instrução **se** da linha 5 verifica se o valor da variável PESQ é igual ao valor da variável indexada NOME[1], e se for igual, é sinal de que o nome foi encontrado. Neste caso, a variável **ACHA** passa a ser verdadeira, forçando a execução da linha 11, uma vez que uma das condições do laço **enquanto** da linha 4 tornou-se falsa. Na linha 11, é confirmado se a variável **ACHA** está mesmo com o valor verdadeiro. Sendo essa condição verdadeira, é apresentada a mensagem da linha 12.

Caso na linha 5 seja verificado que o valor de PESQ não é igual a NOME[1], é incrementado 1 à variável I. É executada a próxima verificação de PESQ com NOME[2] e assim por diante. Caso o processamento chegue até o final e não encontre nada, a variável **ACHA** permanece com valor falso. Quando analisada pela linha 11, será então falsa e apresenta a mensagem da linha 14.

A variável **ACHA** exerce um papel importante na rotina de pesquisa, pois serve como um pivô, estabelecendo um valor verdadeiro quando uma determinada informação é localizada. Esse tratamento de variável é conhecido pelo nome de *flag* (bandeira). A variável **ACHA** é o *flag*, podendo dizer que, ao começar a rotina, a bandeira estava "abaixada" - falsa; quando a informação é encontrada, a bandeira é "levantada" - verdadeira, indicando a localização da informação desejada.

7.3.2 - Pesquisa Binária

O método de pesquisa binária é, em média, mais rápido que o método de pesquisa sequencial, no entanto exige que a matriz esteja previamente ordenada. O método de pesquisa binária simula em uma matriz a mesma estratégia usada para consultar um dicionário ou uma lista telefônica. Ele "divide" a matriz em duas partes (por isso chama-se pesquisa binária) e "procura" saber se a informação a ser pesquisada está na posição central da matriz, após a divisão da tabela. Se a informação em pesquisa estiver na posição central, é apresentada. Caso o elemento da posição central seja diferente do valor pesquisado, o método de pesquisa verifica a possibilidade de essa informação estar posicionada na primeira ou na segunda parte da matriz.

Neste ponto, se a possibilidade de localizar a informação estiver na primeira parte da tabela, a posição do ponto central até a última posição da segunda parte da tabela será desprezada. A partir dessa ocorrência o final da tabela será o valor da posição central menos um. Se a possibilidade de localizar a informação estiver na segunda parte da tabela, a posição do ponto central até a primeira posição da primeira parte da tabela será desprezada. A partir dessa ocorrência o início da tabela será o valor da posição central mais um.

O processo de pesquisa binária será executado de forma repetitiva enquanto a informação não for localizada. Se ao longo da pesquisa a informação for localizada, ela é apresentada. Caso chegue ao início ou ao final da matriz sem localizar o elemento, indica que ele não está armazenado na matriz.

Devido a essa característica de divisão sucessiva da matriz em duas partes, o volume de dados a ser verificado sempre diminui pela metade. Por esta razão esse método recebe a denominação de pesquisa binária e é, em média, mais rápido que o método sequencial. Para exemplificar a utilização desse tipo de pesquisa, imagine a seguinte tabela:

Pesquisa binária	
Índice	Nomes
1	André
2	Carlos
3	Frederico
4	Golias
5	Silvia
6	Silvio
7	Waldir

A tabela em questão representa uma matriz unidimensional ordenada com sete elementos do tipo **cadeia**. Deseja-se fazer uma pesquisa de um dos seus elementos. No caso, será escolhido o elemento **Waldir**, que é o último da tabela, situado na posição de índice 7.

O processo de pesquisa binária consiste em pegar o número total de elementos e dividir por 2. Sendo assim, soma-se o índice do primeiro elemento, valor 1, com o índice do último elemento, valor 7. Desta forma, $1 + 7$ resulta 8. Em seguida, divide-se o valor 8 por 2 e obtém-se o valor 4, que é a posição central da tabela.

É preciso verificar se o conteúdo da posição central, neste caso, posição de índice 4, é igual ao conteúdo que se está pesquisando, ou seja, verificar se "Golias" é igual a "Waldir". Se o resultado for verdadeiro, a pesquisa é encerrada. Se não, verifica-se em qual das partes da tabela a partir da posição central pode ser encontrado o elemento "Waldir", ou seja, a possibilidade de localizar "Waldir" está na segunda parte (parte inferior) da tabela em relação à posição de índice 4.

A tabela fica dividida em duas partes, a partir da posição central. A primeira parte (parte superior) delimitada entre os índices 1 e 3 e a segunda parte

(parte inferior) delimitada entre os índices 5 e 6, como mostra o exemplo seguinte.

Parte superior	
Índice	Nomes
1	André
2	Carlos
3	Frederico

Parte central	
Índice	Nomes
4	Golias

Parte inferior	
Índice	Nomes
5	Silvia
6	Silvio
7	Waldir

Como há possibilidade de encontrar "Waldir" na parte de baixo, despreza-se por completo a parte da tabela acima da posição de índice 4, ou seja, o trecho entre os índices 1 a 3, e assume-se o trecho de índices 5 a 6 como porção válida. Observe que o novo início da tabela é a posição de índice 5, obtida a partir da posição central mais uma unidade de posição a partir do meio da tabela.

Caso necessite pegar a porção de cima a partir do ponto central, deve-se definir a nova posição de fim da tabela. Considerando a posição central como índice 4, a nova posição de fim será o índice 3, ou seja, o novo final da tabela é o valor do índice da posição do meio menos uma unidade em relação à posição central da tabela.

O novo começo será a posição de índice 5 (parte inferior da matriz a partir de seu meio representado pelo índice 4), ou seja, o começo é a posição central acrescida de uma unidade em relação à posição central. De acordo com o exemplo exposto, é preciso pegar a porção inferior da tabela. Assim sendo, considera-se que a tabela tem seu início marcado na posição de índice 5, desprezando a primeira parte, que é menor que 5. Como são três

elementos restantes, soma-se o valor da posição 5 com o valor da posição 7; tem-se então o valor 12 que, dividido por 2, resulta no valor do meio 6.

Pesquisa binária	
Índice	Nomes
5	Silvia
6	Silvio
7	Waldir

Verifica-se então se o conteúdo da posição de índice 6 é igual ao conteúdo que se está pesquisando, ou seja, se "Waldir" é igual a "Silvio". Se o conteúdo for igual, a pesquisa termina aqui. Se não, verifica-se em qual parte da tabela a partir da posição central "Waldir" pode estar. Novamente a tabela é dividida em duas partes. Desconsiderando a posição central que já fora avaliada, tem-se:

Parte superior	
Índice	Nomes
5	Silvia

Parte inferior	
Índice	Nomes
7	Waldir

Nesta etapa verifica-se se o conteúdo a ser pesquisado está na parte superior ou inferior da tabela. A probabilidade de encontrar "Waldir" está indicada na parte inferior da tabela. Assim sendo, assume-se a parte inferior da tabela, em que o valor do novo começo é o valor do índice do meio mais uma posição, ou seja, o meio está neste momento com valor de índice 6 que, somando a 1, resulta num começo a partir do índice 7. Resta apenas um elemento a ser verificado. Para tanto, soma-se o valor do índice de começo que é 7 com o valor do índice de final da tabela que é 7. Obtém-se o valor 14 que, dividido por 2, resulta 7. Verifica-se se o conteúdo da posição de índice 7 é igual ao conteúdo que se está pesquisando, ou seja, se "Waldir" é

igual a "Waldir", então o elemento pesquisado foi localizado e a busca é encerrada.

Imagine que a posição de índice 7 tenha outro nome que não "Waldir", o que aconteceria? Haveria novamente a decisão de pegar a parte superior ou inferior. Neste caso, por ser esse ponto o extremo da tabela, na tentativa de subir ou descer, ocorre um estouro do começo ou do final da tabela (dependendo da parte em uso), indicando que a busca não é mais possível.

No sentido de demonstrar a técnica de pesquisa binária, considere o exemplo seguinte:

Elaborar um programa que leia dez nomes e apresente por meio do método de pesquisa binária os nomes que porventura estejam armazenados na matriz e que coincidam com o nome de entrada de pesquisa. Além de apresentar o nome, o programa deve indicar em que posição da matriz ele está armazenado. Caso o nome pesquisado não seja encontrado, deve informar que o nome pesquisado não foi localizado.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações a serem efetuadas pelo programa na Figura 7.3.

Entendimento

O algoritmo indicado em seguida descreve a entrada de dez nomes e a apresentação de nomes que podem ser solicitados durante a fase de pesquisa binária.

1. Iniciar um laço de repetição, pedir a leitura de dez nomes e colocá-los em ordem alfabética.
2. Criar um laço de repetição que faça a pesquisa enquanto o usuário desejar. Durante a fase de pesquisa deve ser solicitada a informação a ser pesquisada, a qual deve ser comparada pelo método de pesquisa binária. Sendo igual, mostra; caso contrário, avança para o próximo. Se não achar em toda a lista, informar que não existe o elemento pesquisado; se existir, deve mostrá-lo.
3. Encerrar a pesquisa, quando desejado.

Diagramação

O diagrama de blocos da Figura 7.3 apresenta apenas o trecho do processamento da pesquisa binária. O trecho referente à entrada de dados foi omitido por já ser conhecido.

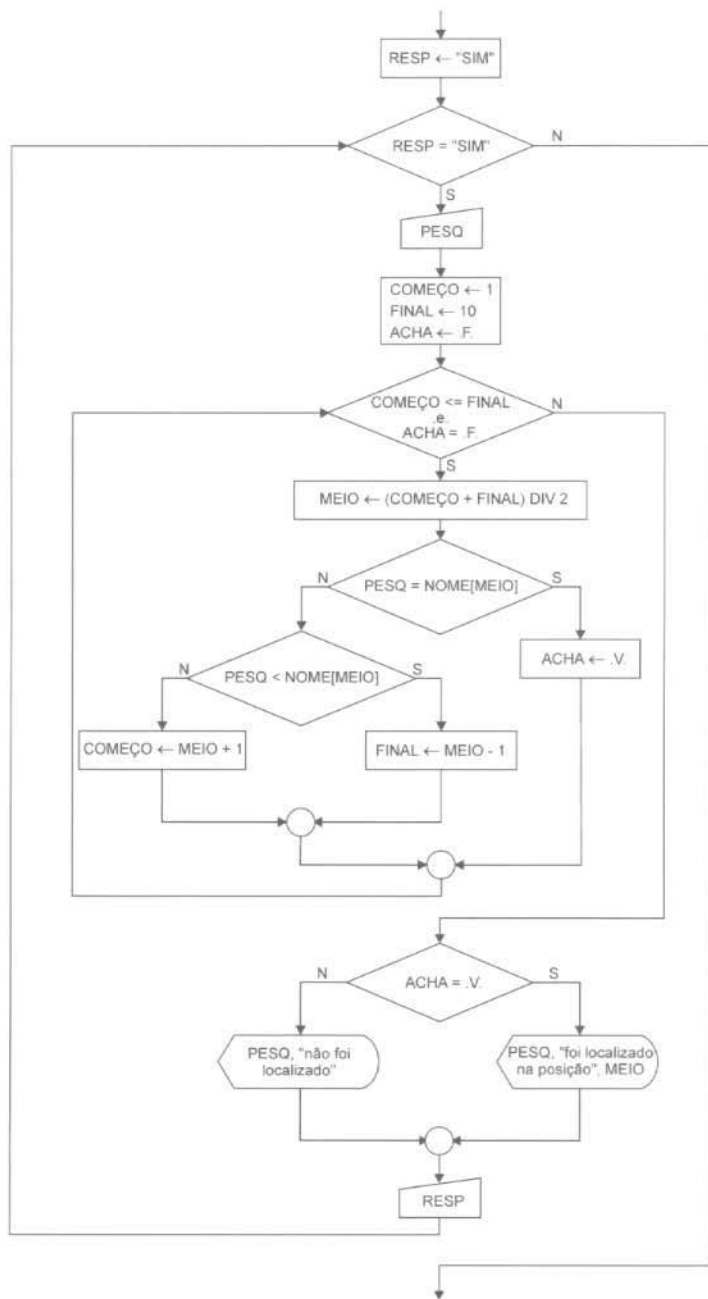


Figura 7.3 - Diagrama de blocos para pesquisa binária.

Codificação

O programa seguinte demonstra o processamento do método de pesquisa binária em um contexto prático.

```

programa PESQUISA_BINÁRIA

var
  NOME : conjunto[1..10] de cadeia
  I, J, COMEÇO, FINAL, MEIO : inteiro
  PESQ, RESP, X : cadeia
  ACHA : lógico
início

  para I de 1 até 10 passo 1 faça
    leia NOME[I]
  fim_para

  (** início trecho de ordenação **)

  para I de 1 até 9 passo 1 faça
    para J de I + 1 até 10 passo 1 faça
      se (NOME[I] > NOME[J]) então
        X ← NOME[I]
        NOME[I] ← NOME[J]
        NOME[J] ← X
      fim_se
    fim_para
  fim_para

  (** fim trecho de ordenação **)

  (** início trecho de pesquisa binária **)

  RESP ← "SIM"
  enquanto (RESP = "SIM") faça
    escreva "Entre o nome a ser pesquisado: "
    leia PESQ
    COMEÇO ← 1
    FINAL ← 10
    ACHA ← .Falso.
    enquanto (COMEÇO <= FINAL) .e. (ACHA = .Falso.) faça
      MEIO ← (COMEÇO + FINAL) div 2
      se (PESQ = NOME[MEIO]) então
        ACHA ← .Verdadeiro.
      senão
        se (PESQ < NOME[MEIO]) então
          FINAL ← MEIO - 1
        senão
          COMEÇO ← MEIO + 1
        fim_se
      fim_se
    fim_enquanto
    se (ACHA = .Verdadeiro.) então
      escreva PESQ, " foi localizado na posição: ", MEIO
    senão
      escreva PESQ, " não foi localizado"
    fim_se
    escreva "Deseja continuar? (SIM/NÃO): "
    leia RESP
  fim_enquanto

  (** fim trecho de pesquisa binária **)

fim

```

Anteriormente foi montada a rotina de pesquisa empregada em um contexto. Observe o trecho que executa a pesquisa com seus comentários:

```

1 - leia PESQ
2 - COMEÇO ← 1
3 - FINAL ← 10
4 - ACHA ← .Falso.
5 - enquanto (COMEÇO <= FINAL) .e. (ACHA = .Falso.) faça
6 -   MEIO ← (COMEÇO + FINAL) div 2
7 -   se (PESQ = NOME[MEIO]) então
8 -     ACHA ← .Verdadeiro.
9 -   senão
10 -    se (PESQ < NOME[MEIO]) então
11 -      FINAL ← MEIO - 1
12 -    senão
13 -      COMEÇO ← MEIO + 1
14 -    fim_se
15 -  fim_enquanto
16 - fim_enquanto
17 - se (ACHA = .Verdadeiro.) então
18 -   escreva PESQ, "foi localizado na posição", MEIO
19 - senão
20 -   escreva PESQ, "não foi localizado"
21 - fim_se

```

Na linha 1, é solicitado o dado a ser pesquisado. As linhas 2 e 3 inicializam as variáveis de controle **COMEÇO** com 1 e **FINAL** com 10. A linha 4 inicializa o flag **ACHA**. A linha 5 apresenta a instrução que manterá a pesquisa em execução enquanto o **COMEÇO** for menor ou igual ao **FINAL** e, simultaneamente, o flag **ACHA** for falso. O processamento divide a tabela ao meio, conforme instrução na linha 6, em que 1, que é o começo da tabela, é somado com 10 que é o fim da tabela, resultando 11 que, dividido por 2, resulta 5 que é o meio da tabela.

Neste ponto, a tabela está dividida em duas partes. A instrução da linha 7 verifica se o valor fornecido para **PESQ** é igual ao valor armazenado na posição **NOME[5]**. Se for, o flag é setado como verdadeiro, sendo em seguida apresentada a mensagem da linha 18. Se a condição de busca não for igual, pode ocorrer uma de duas situações.

Na primeira situação a informação pesquisada está numa posição acima da atual, no caso **NOME[5]**, ou seja, o valor da variável **PESQ** é menor que o valor de **NOME[5]** (linha 10). Neste caso, deve a variável **FINAL** ser implicada pelo valor da variável **MEIO** subtraída de 1 (linha 11), ficando a variável **FINAL** com valor 4. Se for esta a situação ocorrida, será processada a linha 5 que efetua novamente o laço de repetição pelo fato de o valor 1 da variável **COMEÇO** ser menor ou igual ao valor 4 da variável **FINAL**. A instrução da linha 6 divide a primeira parte da tabela ao meio. Desta forma, 1 é somado com 4, resultando 5, e dividido por 2 resulta 2 (sendo considerada a parte inteira do resultado da divisão), que é o meio da primeira parte da tabela.

A segunda situação pode ocorrer caso a informação pesquisada esteja abaixo do meio da tabela, ou seja, o valor da variável **PESQ** é maior que o valor de **NOME[5]** (linha 10). Neste caso, deve a variável **COMEÇO** ser implicada pelo valor da variável **MEIO** somado com 1 (linha 13), ficando a variável **COMEÇO** com valor 6. Se for esta a situação ocorrida, será processada a linha 5 que efetua novamente o laço de repetição pelo fato de o valor 6 da variável **COMEÇO** ser menor ou igual ao valor 10 da variável **FINAL**. A instrução da linha 6 divide a segunda parte da tabela ao meio. Desta forma, 6 é somado com 10, resultando 16 que, dividido por 2, resulta 8, que é o meio da segunda parte da tabela.

Tanto na primeira como na segunda situação, sempre se utiliza uma das metades da tabela. A vantagem desse método está exatamente na metade desprezada, pois ela não é verificada novamente.

7.4 - Utilização de Matrizes Dinâmicas

Os exemplos apresentados anteriormente fizeram uso de matrizes estáticas, ou seja, matrizes em que se conhece de antemão o número de elementos que serão nelas armazenados. No entanto, nem sempre é possível trabalhar com essa abordagem. Há situações em que há a necessidade de se trabalhar com matrizes sem saber de antemão quantos elementos elas devem armazenar. Assim sendo, é necessário fazer uso de matrizes dinâmicas.

A definição de matrizes dinâmicas em português estruturado segue a sintaxe seguinte:

```
VARIÁVEL : conjunto[] de <tipo de dado>
```

Em que **conjunto** é seguido de um par de colchetes em branco, o qual deixa em aberto para o programa a quantidade de elementos a ser definida dentro do próprio programa.

O processo de operação de entrada, processamento e saída de dados em um programa que usa matrizes dinâmicas é semelhante aos códigos de programas já utilizados para manipulação de matrizes estáticas, com a diferença de que o número de elementos a ser usado é definido dentro do código do programa pelo próprio usuário do programa. Assim sendo, considere como exemplo o seguinte requisito:

Elaborar um programa que leia uma matriz dinâmica A com N elementos do tipo cadeia para representar os nomes de algumas pessoas e em seguida apresentar a lista de nomes que foram inseridos na matriz A.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa na Figura 7.4.

Entendimento

1. Efetuar a leitura do número de elementos a ser inserido na matriz dinâmica A.
2. Ler os N elementos para a matriz dinâmica A.
3. Apresentar os elementos que foram inseridos na matriz dinâmica A.

Diagramação

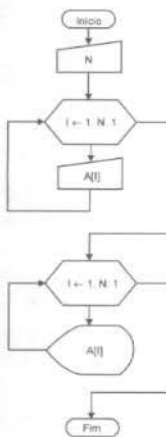


Figura 7.4 - Diagrama de blocos para programa com matriz dinâmica.

Codificação

```

programa MATRIZ_DINÂMICA
var
  I, N : inteiro
  A : conjunto[] de cadeia
início

  leia N

  para I de 1 até N passo 1 faça
    leia A[I]
  fim_para

  para I de 1 até N passo 1 faça
    escreva A[I]
  fim_para

fim
  
```

7.5 - Exercício de Aprendizagem

Apesar de os exemplos anteriores demonstrarem com bastante propriedade as técnicas aplicadas neste capítulo, faz-se necessário para o iniciante a apresentação de mais exemplos que ilustrem essas técnicas. A seguir veja três situações que ilustram as técnicas apresentadas e servem de base à solução dos exercícios de fixação.

1º Exemplo

Elaborar um programa que leia dez elementos numéricos inteiros em uma matriz A de uma dimensão do tipo vetor. Construir uma matriz B de mesma dimensão e tipo com os elementos da matriz A divididos por cinco. Apresentar os elementos armazenados na matriz B na ordem decrescente.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa nas Figuras 7.5 (a) e 7.5 (b).

Diagramação

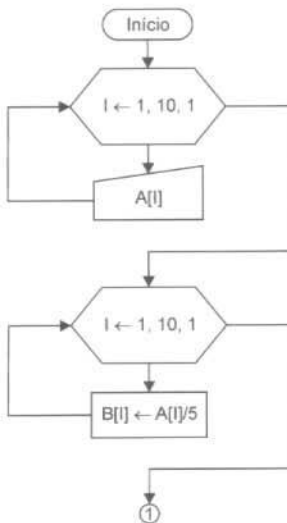


Figura 7.5 (a) - Diagrama de blocos para demonstração do primeiro exemplo.

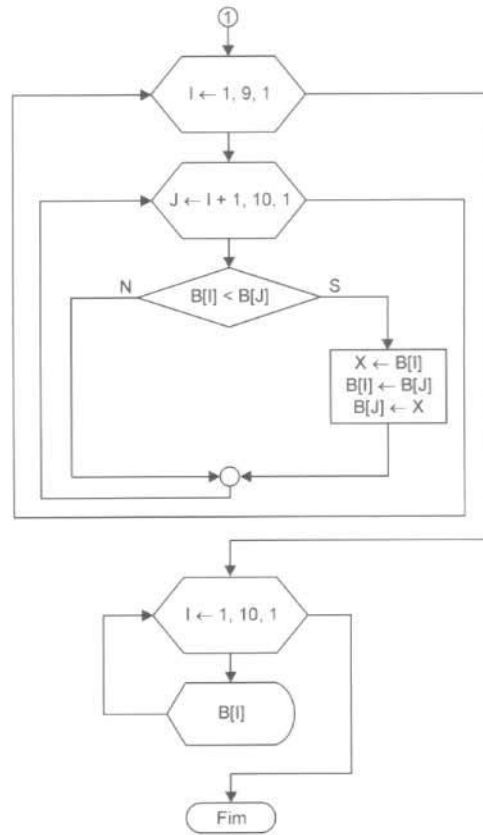


Figura 7.5 (b) - Diagrama de blocos para demonstração do primeiro exemplo.

Entendimento

1. Iniciar um laço de repetição e pedir a leitura dos dez valores inteiros para a matriz A.
2. Criar uma matriz B (do tipo real) que conterà em cada uma de suas posições os valores dos elementos da matriz A divididos por cinco.
3. Ordenar de forma decrescente os valores da matriz B.
4. Apresentar os valores da matriz B.

Codificação

```

programa EXEMPLO_1
var
  I, J : inteiro
  A : conjunto[1..10] de inteiro
  A : conjunto[1..10] de real
  X : real
inicio
  {Trecho de entrada de dados}

  para I de 1 até 10 passo 1 faça
    leia A[I]
  fim_para

  {Trecho de criação da matriz B}

  para I de 1 até 10 passo 1 faça
    B[I] ← A[I] / 5
  fim_para

  {Trecho de processamento de ordenação}

  para I de 1 até 9 passo 1 faça
    para J de I + 1 até 10 passo 1 faça
      se (B[I] < B[J]) então
        X ← B[I]
        B[I] ← B[J]
        B[J] ← X
      fim_se
    fim_para
  fim_para

  {Trecho de saída com dados ordenados}

  para I de 1 até 10 passo 1 faça
    escreva B[I]
  fim_para

fim

```

2º Exemplo

Elaborar um programa que leia duas matrizes A e B de uma dimensão do tipo vetor com 20 elementos inteiros cada. Construir uma matriz C de mesmo tipo e dimensão que seja formada pela subtração de cada um dos elementos da matriz A de cada elemento correspondente da matriz B. Montar o trecho de pesquisa sequencial para pesquisar os elementos existentes na matriz C.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa nas Figuras 7.6 (a) e 7.6 (b).

Entendimento

1. Efetuar a leitura dos 20 elementos inteiros da matriz A.
2. Ler os 20 elementos inteiros da matriz B.
3. Criar uma matriz C que conterà o valor da subtração da matriz A em relação à matriz B.
4. Apresentar os valores da matriz C a partir da pesquisa sequencial.

Diagramação

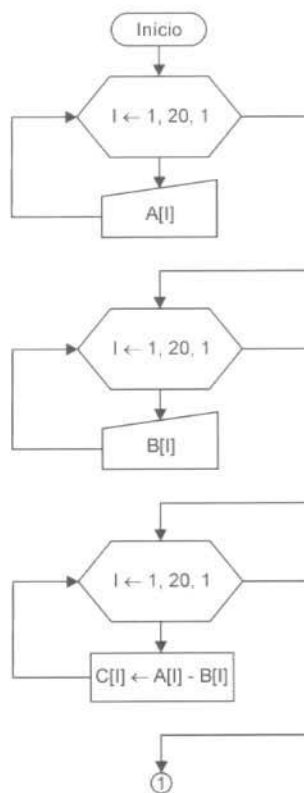


Figura 7.6 (a) - Diagrama de blocos para demonstração do segundo exemplo.

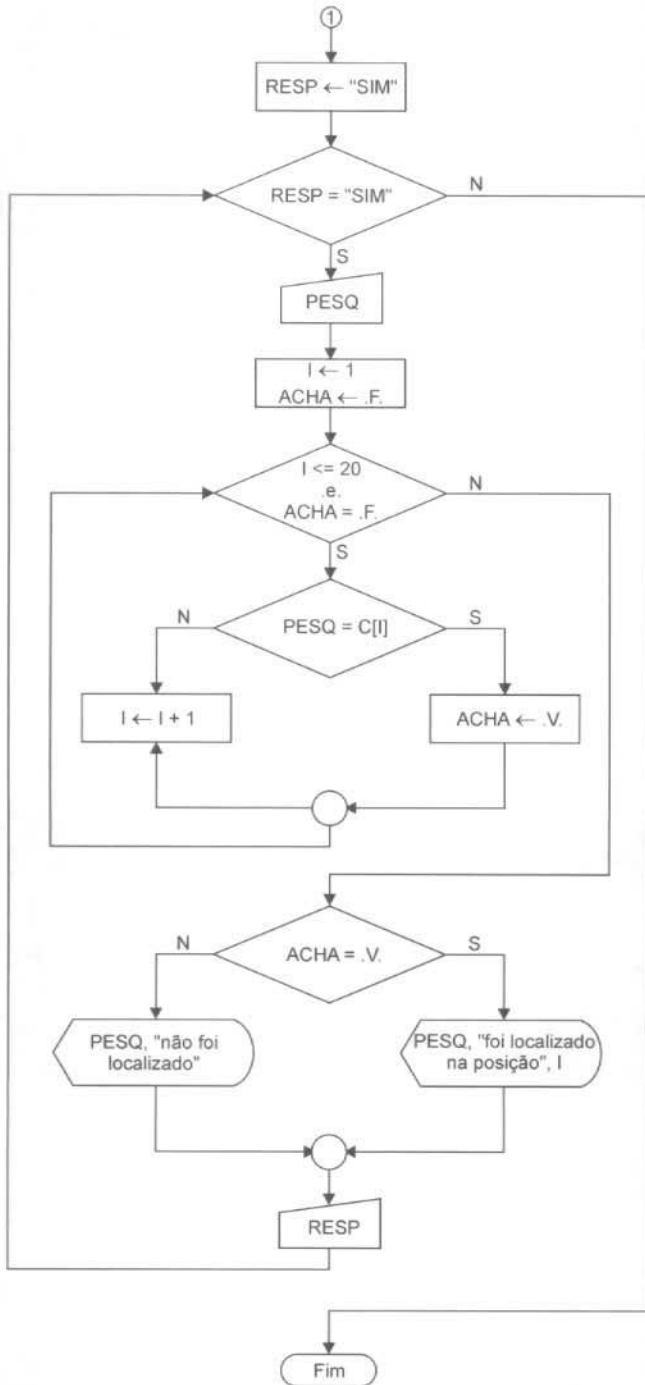


Figura 7.6 (b) - Diagrama de blocos para demonstração do segundo exemplo.

Codificação

```

programa EXEMPLO_2
var
  A, B, C : conjunto[1..20] de inteiro
  RESP : cadeia
  ACHA : lógico
  I, PESQ : inteiro
inicio

  {Trecho de entrada de dados da matriz A}

  para I de 1 até 20 passo 1 faça
    leia A[I]
  fim_para

  {Trecho de entrada de dados da matriz B}

  para I de 1 até 20 passo 1 faça
    leia B[I]
  fim_para

  {Trecho de criação da matriz C}

  para I de 1 até 20 passo 1 faça
    C[I] ← A[I] - B[I]
  fim_para

  {Trecho de pesquisa seqüencial}

  RESP ← "SIM"
  enquanto (RESP = "SIM") faça
    leia PESQ
    I ← 1
    ACHA ← .Falso.
    enquanto (I <= 20) .e. (ACHA = .Falso.) faça
      se (PESQ = C[I]) então
        ACHA ← .Verdadeiro.
      senão
        I ← I + 1
      fim_se
    fim_enquanto
    se (ACHA = .Verdadeiro.) então
      escreva PESQ, " foi localizado na posição ", I
    senão
      escreva PESQ, " não foi localizado"
    fim_se
    leia RESP
  fim_enquanto

fim

```

3º Exemplo

Elaborar um programa que leia uma matriz A com 15 elementos inteiros e uma matriz B com 35 elementos inteiros. Construir uma matriz C de mesmo tipo e dimensão que seja formada pela junção dos elementos das matrizes A e B de forma que possa armazenar 50 elementos. Montar o trecho de pesquisa binária para pesquisar os elementos existentes na matriz C.

Observe a seguir a descrição das etapas básicas de entendimento do problema e a representação das ações do programa nas Figuras 7.7 (a), 7.7 (b) e 7.7 (c).

Entendimento

1. Efetuar a leitura dos 15 elementos inteiros da matriz A.
2. Efetuar a leitura dos 35 elementos inteiros da matriz B.
3. Criar a matriz C com 50 elementos como junção das matrizes A e B.
4. Apresentar os valores da matriz C a partir da pesquisa binária.

Diagramação

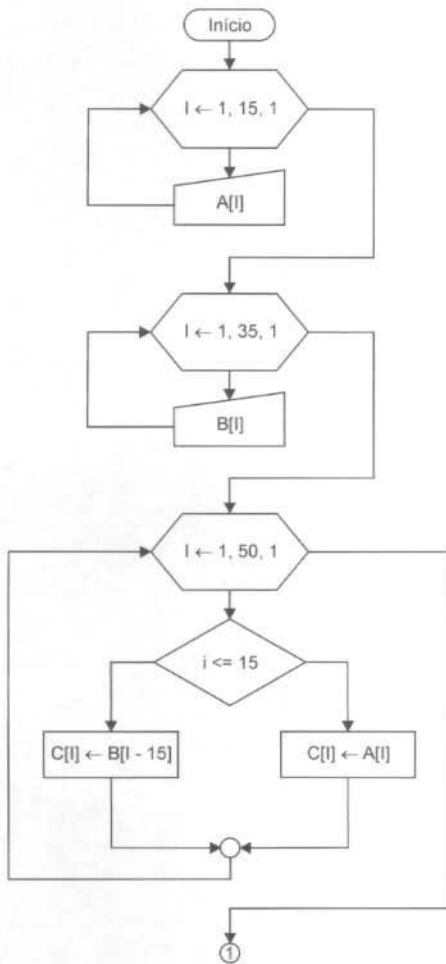


Figura 7.7 (a) - Diagrama de blocos para demonstração do terceiro exemplo.

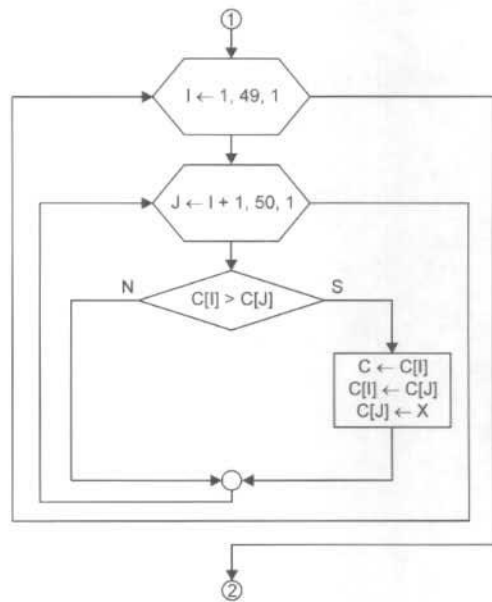


Figura 7.7 (b) - Diagrama de blocos para demonstração do terceiro exemplo.

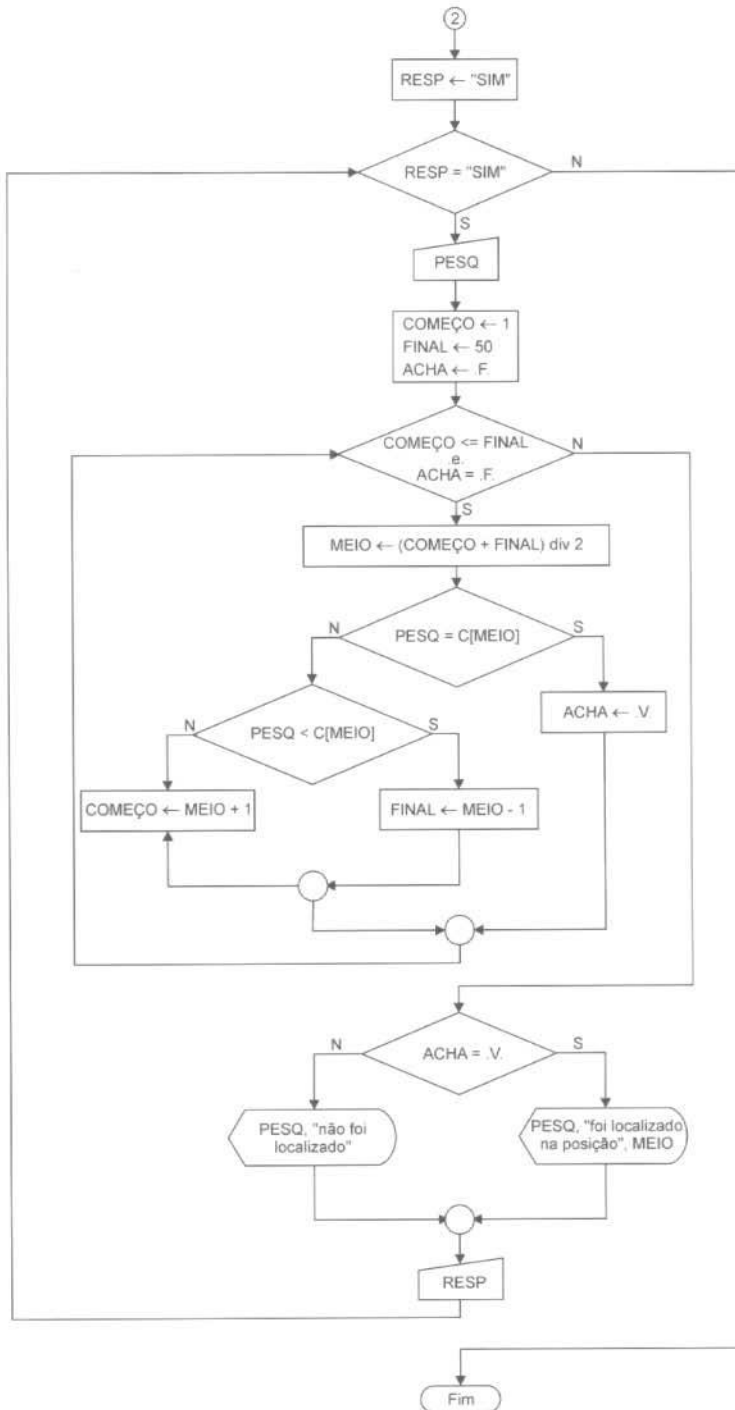


Figura 7.7 (c) - Diagrama de blocos para demonstração do terceiro exemplo.