rede e-Tec Brasil

Lógica de Programação

Rogério da Silva Batista

Curso Técnico em Informática




INSTITUTO FEDERAL DE EDUCAÇÃO CIÊNCIA E TECNOLOGIA PIAUÍ

UNIVERSIDADE FEDERAL DE SANTA CATARINA

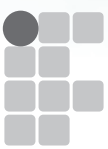
PDE | PRONATEC
PROGRAMA NACIONAL DE ACESSO AO ENSINO TÉCNICO E EMPREGO

Ministério da Educação

rede e-Tec Brasil

Lógica de Programação

Rogério da Silva Batista



INSTITUTO FEDERAL DE EDUCAÇÃO CIÊNCIA E TECNOLOGIA
PIAUÍ

Teresina – PI
2013

Presidência da República Federativa do Brasil

Ministério da Educação

Secretaria de Educação Profissional e Tecnológica

Este Caderno foi elaborado em parceria entre o Instituto Federal de Educação, Ciência e Tecnologia do Piauí e a Universidade Federal de Santa Catarina para a Rede e-Tec Brasil.

Equipe de Elaboração
Instituto Federal de Educação, Ciência e Tecnologia do Piauí – IFPI

Coordenação Institucional
Francieric Alves de Araujo/IFPI

Coordenação do Curso
Thiago Alves Elias da Silva/IFPI

Professor-autor
Rogério da Silva Batista/IFPI

Comissão de Acompanhamento e Validação
Universidade Federal de Santa Catarina – UFSC

Coordenação Institucional
Araci Hack Catapan/UFSC

Coordenação do Projeto
Silvia Modesto Nassar/UFSC

Coordenação de Design Instrucional
Beatriz Helena Dal Molin/UNIOESTE e UFSC

Coordenação de Design Gráfico
Juliana Tonietto/UFSC

Design Instrucional
Juliana Leonardi/UFSC

Web Master
Rafaela Lunardi Comarella/UFSC

Web Design
Beatriz Wilges/UFSC
Mônica Nassar Machuca/UFSC

Diagramação
Bárbara Zardo/UFSC
Breno Takamine/UFSC
Juliana Tonietto/UFSC
Roberto Gava Colombro/UFSC

Projeto Gráfico
e-Tec/MEC

Catalogação na fonte pela Biblioteca Universitária
Universidade Federal de Santa Catarina

B333l Batista, Rogério da Silva
Lógica de programação / Rogério da Silva Batista. – Teresina : Instituto Federal de Educação, Ciência e Tecnologia do Piauí, 2013.
158 p. : il., tabs.

Inclui bibliografia
ISBN: 978-85-67082-05-9

1. Programação (Computadores). 2.Programação lógica. I. Título.

CDU 681.31.06

Apresentação e-Tec Brasil

Bem-vindo a Rede e-Tec Brasil!

Você faz parte de uma rede nacional de ensino, que por sua vez constitui uma das ações do Pronatec - Programa Nacional de Acesso ao Ensino Técnico e Emprego. O Pronatec, instituído pela Lei nº 12.513/2011, tem como objetivo principal expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional e Tecnológica (EPT) para a população brasileira propiciando caminho de o acesso mais rápido ao emprego.

É neste âmbito que as ações da Rede e-Tec Brasil promovem a parceria entre a Secretaria de Educação Profissional e Tecnológica (SETEC) e as instâncias promotoras de ensino técnico como os Institutos Federais, as Secretarias de Educação dos Estados, as Universidades, as Escolas e Colégios Tecnológicos e o Sistema S.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

A Rede e-Tec Brasil leva diversos cursos técnicos a todas as regiões do país, incentivando os estudantes a concluir o ensino médio e realizar uma formação e atualização contínuas. Os cursos são ofertados pelas instituições de educação profissional e o atendimento ao estudante é realizado tanto nas sedes das instituições quanto em suas unidades remotas, os polos.

Os parceiros da Rede e-Tec Brasil acreditam em uma educação profissional qualificada – integradora do ensino médio e educação técnica, - é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação
Março de 2013

Nosso contato
etecbrasil@mec.gov.br

Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.

Atenção: indica pontos de maior relevância no texto.

Saiba mais: oferece novas informações que enriquecem o assunto ou "curiosidades" e notícias recentes relacionadas ao tema estudado.

Glossário: indica a definição de um termo, palavra ou expressão utilizada no texto.

Mídias integradas: sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.

Atividades de aprendizagem: apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

Sumário

Palavra do professor-autor

Caro (a) estudante,

Bem-vindo (a) à disciplina Lógica de Programação.

Este é nosso "Material Impresso", elaborado com o objetivo de contribuir para o desenvolvimento de seus estudos e para a ampliação de seus conhecimentos acerca da citada disciplina.

Apresentação da disciplina

Este texto é destinado aos estudantes aprendizes que participam do programa Escola Técnica Aberta do Brasil (e-Tec Brasil), vinculado à Escola Técnica Aberta do Piauí (ETAPI) do Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI), com apoio da Prefeitura Municipal dos respectivos polos: Alegrete do Piauí, Batalha, Monsenhor Gil e Valença do Piauí. O texto é composto de nove (09) Aulas assim distribuídas:

Na Aula 1 introduzimos o conceito de Lógica de Programação e a necessidade do uso dessa Lógica, a aplicabilidade da lógica no desenvolvimento de programas através do conceito de Algoritmos e Conceitos de Programas de Computador.

Na Aula 2: Formas de Representação de Algoritmos, apresentando a Descrição Narrativa, o Fluxograma, o Diagrama de Chapin e o Pseudocódigo.

Na Aula 3: A Manipulação de Dados apresenta os Tipos de Dados, Variáveis e Constantes abordando seus conceitos, utilidades e exemplos de uso.

Na Aula 4: Operadores e Expressões abordam o conceito de Operadores e Expressões e como estes estão classificados e seus exemplos de uso.

Na Aula 5: Instruções Primitivas, vamos conhecer e utilizar os comandos de entrada e saída de dados e Identificar e aplicar os conceitos de entrada, processamento e saída na resolução de algoritmos.

Na Aula 6: Estruturas de Controle e tomadas de decisões, Vamos abordar as características da estruturas de decisão simples e compostas aplicá-las em exemplos práticos.

Na Aula 7: Estruturas de Repetição, vamos abordar as características das estruturas de repetição e em que situações é mais apropriado usá-las.

Na Aula 8: Estruturas Homogêneas: Vetores e Matrizes, vamos abordar os conceitos das estruturas homogêneas e compreender suas utilizações em diversas situações práticas.

Na Aula 9: Sub-Rotinas, Vamos entender o mecanismo de funcionamento de uma sub-rotina dentro de um algoritmo e diferenciar os tipos de sub-rotinas existentes e aplicá-las dentro de um algoritmo através de chamadas.

Projeto instrucional

Disciplina: Lógica de Programação (carga horária: 120h).

Ementa: Abordagem contextual quanto à Lógica de Programação. Formas de Representação de Algoritmos. Estruturas de Tomadas de decisões e Repetições. As Estruturas de Dados Homogêneas (Vetores e Matrizes). Programação Modular (funções e Procedimentos).

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
1. Introdução à Lógica de Programação	Aplicar a lógica formal e matemática na Programação de Computadores. Definir o conceito de lógica e suas aplicações diárias. Conceituar Algoritmos descrevendo passo a passo o processo de construção de um algoritmo para resolução de um problema clássico.	Caderno. Vídeo-aulas. *Web* conferência.	8
2. Formas de representação de algoritmos	Conhecer e identificar as diversas formas de representação de algoritmos existentes. Aplicar a melhor forma de representação de acordo com o algoritmo fornecido.	Caderno. Vídeo-aulas. *Web* conferência.	10
3. Manipulação de dados	Entender os conceitos de variável e de constante e suas utilizações. Utilizar adequadamente cada tipo de dado disponível.	Caderno. Vídeo-aulas. *Web* conferencia.	10
4. Operadores e expressões	Identificar, utilizar e avaliar corretamente as expressões aritméticas e lógicas. Identificar e utilizar os principais operadores utilizados na programação de computadores.	Caderno. Vídeo-aulas. *Web* conferência.	10
5. Instruções primitivas	Identificar uma instrução e utilizar blocos de instruções no desenvolvimento de algoritmos. Conhecer e utilizar os comandos de entrada e saída de dados. Identificar e aplicar os conceitos de entrada, processamento e saída na resolução de algoritmos. Aplicar corretamente o teste de mesa nos algoritmos a fim de verificar se a lógica aplicada está correta.	Caderno. Vídeo-aulas. *Web* conferência.	10

Continua

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
6. Estruturas de controle: A tomada de decisões	Conhecer as características da estruturas de decisão simples e aplicá-las em exemplos práticos. Conhecer as características das estruturas de decisão compostas e aplicá-las em exemplos práticos. Aplicar a combinação destas estruturas em exemplos práticos.	Caderno. Vídeo-aulas. *Web* conferência.	18
7. Estruturas de repetição.	Conhecer as características das estruturas de repetição e em que situações é mais apropriado usá-las. Entender passo-a-passo a sintaxe e como funciona cada estrutura de repetição (Enquanto, Repita e Para) e estudar como um problema pode ser resolvido utilizando-se mais de um algoritmo com estruturas de repetição diferentes. Construir algoritmos utilizando estruturas de repetição combinadas.	Caderno. Vídeo-aulas. *Web* conferência.	18
8. Estruturas homogêneas: vetores e matrizes	Conceituar as estruturas homogêneas e compreender suas utilizações em diversas situações práticas. Aprender a declarar, preencher, atribuir valores e mostrar os elementos de um vetor ou matriz. Utilizar vetores e matrizes na construção de algoritmos de acordo com os problemas apresentados	Caderno. Vídeo-aulas. *Web* conferência.	18
9. Sub-rotinas	Conceituar as sub-rotinas e compreender as vantagens de sua utilização na construção de algoritmos. Entender o mecanismo de funcionamento de uma sub-rotina dentro de um algoritmo. Diferenciar os tipos de sub-rotinas existentes e aplicá-las dentro de um algoritmo através de chamadas. Utilizar parâmetros nas sub-rotinas como meio de comunicação entre o programa principal e outras sub-rotinas. Diferenciar parâmetros reais e formais. Diferenciar passagem de parâmetros por valor e por referencia e entender o que isso afeta dentro de um algoritmo.	Caderno. Vídeo-aulas. *Web* conferência.	18
			Conclusão

Aula 1 - Introdução à lógica de programação

Objetivos

Aplicar a lógica formal e matemática na programação de computadores.

Definir o conceito de lógica e suas aplicações diárias.

Conceituar algoritmos, descrevendo passo a passo o processo de construção de um algoritmo para resolução de um problema clássico.

1.1 Conceituação

Sem duvida, o computador é uma das maiores invenções do homem e tem se mostrado uma ferramenta versátil, rápida e segura para a manipulação de informações.

Para muitos, essa invenção é responsável pela intensificação da mecanização e descobertas científicas na vida moderna. Esta afirmação dá um caráter autônomo ao computador, como se o mesmo fizesse tudo sozinho. Entretanto cabe esclarecer que o computador não é criativo e nem inteligente, na verdade, apenas reproduz o que lhe é ordenado por meio de seus programas de computador, os quais são construídos para resolver algum problema específico e a solução adotada é sempre uma solução lógica. Podemos entender essa "solução lógica" como uma espécie de receita de bolo a ser adotada para a solução do problema. A Lógica de Programação é então o passo inicial para a construção de um programa de computador.

Para usar a lógica, é necessário ter domínio sobre o pensamento, bem como saber pensar, ou seja, possuir 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 aristotélico-tomista, também conhecida por Lógica Formal.

Em contraposição a esse conceito de Lógica Formal, surgiu um outro – o de Lógica Material – para designar o estudo do raciocínio no que ele depende quanto ao seu conteúdo ou matéria.

Esta distinção entre lógica formal e lógica material permite-nos agora perceber porque, tendo em vista a sua forma, o raciocínio é correto ou incorreto (válido ou invalido). Mas se atendermos à sua matéria, a conclusão pode ser verdadeira ou falsa. Exemplo:

1. Nenhum homem sabe dançar;
2. Este dançarino é homem;
3. Este dançarino não sabe dançar.

Este raciocínio é formalmente correto, uma vez que a conclusão está corretamente deduzida. Mas a conclusão é falsa, uma vez que é falsa a primeira proposição ("Nenhum homem sabe dançar"). Estamos perante um raciocínio que tem validade formal, mas não tem validade material. Logo temos que concluir que é falso.

Desde a sua criação a lógica tem registrado enormes aperfeiçoamentos, sobretudo, a partir de meados do século XIX. É costume dividir-se a sua história em três períodos: período Clássico, período Moderno e período Contemporâneo.

A Lógica Matemática ou Simbólica (surgida no Período Moderno), é usada para expressar o conteúdo do pensamento simbólico e defende a ideia de que o raciocínio é visto como cálculo matemático. A Lógica Matemática exerceu uma influência decisiva em muitos domínios, principalmente na Eletrônica, Cibernética, Informática, Inteligência Artificial, dentre outros.

Mesmo com essa multiplicidade de conceitos, a Lógica pode ser vista como uma ciência que procura encontrar as leis em relação às quais o nosso pensamento deve obedecer para que possa ser considerado válido.

No contexto da informática, a Lógica de Programação é a técnica de encadear pensamentos para atingir determinado objetivo previamente definido, ou seja, é a técnica que nos permite expressar o que deve ser feito e em que ordem para que a solução seja alcançada.

1.2 A necessidade do uso da lógica

Sempre que pensamos estamos exercitando o uso da lógica. Toda vez que falamos também estamos fazendo uso da lógica uma vez que a fala é apenas uma representação do que pensamos. Quantas vezes em nosso cotidiano, dissemos afirmações do tipo: "Isso é lógico!", "... Não tem lógica alguma.", "Não vejo lógica nisso.". Saber o que é lógico, ou saber identificar uma estrutura lógica em um contexto linguístico, é algo que nos é transmitido por meio da nossa educação.

Além dessa lógica linguística, aplicamos outros tipos de raciocínio lógico em nosso dia-a-dia. Um bom exemplo seria porque não colocamos nossa mão em uma superfície quente. Isso parece lógico, não é? Na verdade, isso acontece porque nosso cérebro processa sentenças lógicas como:

1. A pele humana não suporta altas temperaturas (ou algo mais simples como: "queimei minha pele no último contato com uma superfície quente");
2. A minha mão é coberta de pele;
3. Logo, a minha mão não suporta altas temperaturas.

Esse tipo de pensamento lógico se repete várias e várias vezes ao dia e graças à lógica, evitamos certos tipos de problemas.

Isso deixa claro que nós pensamos de forma lógica o tempo todo. No entanto, temos uma grande dificuldade em formalizar este raciocínio lógico, pois não somos acostumados a formalizar nosso pensamento.

O uso da lógica é um fator a ser considerado por todos, principalmente pelos profissionais da área de Tecnologia da Informação (programadores, analistas de sistemas), 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. Vale ressaltar que ninguém ensina ninguém a pensar, pois todas as pessoas normais possuem esse "dom". O objetivo deste material é mostrar como desenvolver a aperfeiçoar melhor essa técnica para dizer ao computador o que deve ser feito para atingir a solução de um determinado problema.

1.3 Algoritmos: aplicabilidade da lógica no desenvolvimento de programas

A construção de algoritmos é o primeiro passo para o desenvolvimento de programas de computador. É uma das tarefas mais complexas da programação de computadores, mas também uma das mais desafiadoras e empolgantes.

Muitas definições podem ser dadas à palavra algoritmo. Atualmente, tem-se associado algoritmo à computação, mas este não é um termo restrito à computação ou que tenha nascido com ela. Na realidade, a palavra algoritmo vem do nome do matemático iraniano Abu Abdullah Mohammad Ibn Musa al-Khawarizmi, nascido em Khawarizmi (Kheva), ao sul do mar de Aral, que viveu no século XVII. A influência de Khawarizmi no crescimento da ciência em geral, particularmente na matemática, astronomia e geografia, é bastante reconhecida.

O termo algoritmo também é utilizado em outras áreas como: engenharia, administração, entre outras. Algumas definições de algoritmo:

- Uma sequência de passos que visa a atingir um objetivo definido.
- Um procedimento passo a passo para a solução de um problema.
- Uma sequência detalhada de ações a serem executadas para realizar alguma tarefa.

Um exemplo clássico de algoritmo é uma receita culinária. Veja o exemplo a seguir de um bolo de chocolate.

Ingredientes:

4 xícaras (chá) de farinha de trigo.
2 xícaras (chá) de açúcar cristal.
2 xícaras (chá) de achocolatado.
2 colheres (sopa) de fermento em pó.
1 pitada de sal.
3 ovos.
2 xícaras (chá) de água morna.
1 xícara (chá) de óleo.
Óleo para untar.
Farinha de trigo para polvilhar.

Modo de preparo: Numa vasilha, misture 4 xícaras (chá) de farinha de trigo, 2 xícaras (chá) de açúcar cristal, 2 xícaras (chá) de achocolatado, 2 colheres (sopa) de fermento em pó e 1 pitada de sal. Junte 3 ovos, 2 xícaras (chá) de água morna e 1 xícara (chá) de óleo. Misture bem. Unte uma forma retangular de 25 cm x 37 cm com óleo e polvilhe farinha de trigo e despeje a massa. Asse em temperatura média (de 170°C a 180°C) por 30 minutos.

A receita tem todas as características de um algoritmo. Ela tem uma sequência detalhada de passos, descrita no modo de preparo. Apresenta a tarefa a ser realizada, que no caso é o bolo de chocolate. Além disto, podemos identificar na receita entradas (no caso, os ingredientes) e uma saída, que é o próprio bolo.

Poderíamos, então, nos perguntar por que a palavra algoritmo ficou tão associada à computação? Para compreendermos melhor os motivos, é preciso entender, mesmo que superficialmente, o funcionamento dos programas de computador. A Figura 1.1 mostra uma representação gráfica dos elementos de programa de computador.

Figura 1.1: Esquema dos elementos de um programa de computador
Fonte: Elaborada pelo autor

O algoritmo não é a solução do problema, mas uma forma de solucioná-lo. Assim, para um mesmo problema, podemos criar diferentes algoritmos usando diferentes abordagens. Em outras palavras, podemos usar diferentes sequências de instruções para resolver o mesmo problema. Em alguns casos, até mesmo diferentes instruções.

A construção de um algoritmo deve observar todos os passos necessários à execução da atividade e evitar que passos desnecessários sejam executados ou que passos interdependentes sejam executados fora de ordem.

Durante a construção de um algoritmo são realizadas constantes revisões a fim de identificar novas situações ou exceções a serem tratadas. Quando temos um problema e precisamos construir um algoritmo para resolvê-lo, devemos passar pelas seguintes etapas:

a) definir o problema;

b) realizar um estudo da situação atual e verificar qual(is) a(s) forma(s) de resolver o problema;

c) terminada a fase de estudo, descrever o algoritmo que deverá, a princípio, resolver o problema;

d) analisar junto aos usuários se o problema será resolvido. Se a solução não foi encontrada, ou surgirem exceções a serem tratadas, deverá ser retomado para a fase de estudo para descobrir onde está a falha.

A fim de entender como um algoritmo é construído, vamos analisar a construção de um algoritmo para o seguinte problema clássico, discutido por vários autores, inclusive em Forbellone e Eberspacher (2005, p. 4), por ser um problema cotidiano e que não exige conhecimentos específicos: "trocar uma lâmpada queimada". A partir desse exemplo, será possível evidenciar o processo de encadeamento de ideias até a solução final do problema. Inicialmente, poderíamos construir o seguinte algoritmo básico para solucionar o problema proposto:

1. Pegue uma escada;
2. Posicione-a embaixo da lâmpada;
3. Busque uma lâmpada nova;
4. Suba na escada;
5. Retire a lâmpada velha;
6. Coloque a lâmpada nova.

Se examinarmos esse algoritmo, veremos que ele permite solucionar o problema da troca de uma lâmpada queimada. Entretanto, se considerarmos as situações nas quais o algoritmo poderá ser aplicado, perceberemos que o mesmo não irá tratar a situação em que a lâmpada não esteja queimada. Na verdade, mesmo que a lâmpada esteja funcionando, esse algoritmo irá trocá-la por uma nova.

Para solucionar esse problema, é necessário alterar o algoritmo de modo que a lâmpada seja testada antes de efetuar a troca. Para isso, basta ligar o interruptor e verificar se a lâmpada está funcionando ou não. Intuitivamente, faríamos a seguinte alteração no algoritmo:

1. Pegue uma escada;
2. Posicione-a embaixo da lâmpada;
3. Busque uma lâmpada nova;
4. Ligue o interruptor;
5. Se a lâmpada não acender, então:

a) suba na escada;

b) retire a lâmpada velha;

c) coloque a lâmpada nova.

À primeira vista, o algoritmo agora está correto. Ao introduzirmos um teste seletivo, uma condição, no passo V, impomos que os passos a, b e c só deverão ser executados se o resultado do teste for verdadeiro. Nesse caso, se a lâmpada não acender. Os testes seletivos nos permitem tratar as exceções e garantir que determinados passos não sejam executados em vão, otimizando o algoritmo.

Esse mesmo algoritmo, apesar de funcional, ainda pode ser otimizado. Se o reexaminarmos, perceberemos que, no caso de a lâmpada não estar queimada, teremos executado os passos 1, 2 e 3 em vão.

Para tratar essa situação, bastaria reposicionar os passos 1, 2 e 3 para que ocorram somente após o teste seletivo, uma vez que só serão úteis caso a lâmpada tenha de ser realmente trocada, ou seja, que ela esteja queimada. Desse modo, o algoritmo seria novamente alterado, ficando:

6. Ligue o interruptor;
7. Se a lâmpada não acender, então:

a) pegue uma escada;

b) posicione-a embaixo da lâmpada;

c) busque uma lâmpada nova;

d) suba na escada;

e) retire a lâmpada velha;

f) coloque a lâmpada nova.

Apesar de parecer completo, ou seja, permitindo solucionar o problema da troca da lâmpada queimada, este algoritmo não leva em consideração a possibilidade de a nova lâmpada não funcionar. Pois, nesse caso, seria necessário repetir os passos "e" e "f". Daí surge a seguinte questão: pode ser que a outra lâmpada também não funcione, sendo necessário repetir mais uma vez esta sequência de passos. Então, quando deveremos parar de repetir?

Para toda repetição em um algoritmo devemos estabelecer uma condição de parada, ou seja, um limite para a quantidade de vezes em que os passos deverão ser repetidos. Caso não seja estabelecido esta condição de parada ocorre o que chamamos de laço infinito - mais conhecido por sua designação em inglês loop infinito - no qual os passos se repetem indefinidamente.

No caso do nosso algoritmo, uma condição de parada adequada seria repetir enquanto a lâmpada não acender. Assim o algoritmo ficaria da seguinte forma:

7. Ligue o interruptor;
8. Se a lâmpada não acender, então:

    a) pegue uma escada;

    b) posicione-a embaixo da lâmpada;

    c) busque uma lâmpada nova;

    d) suba na escada;

    e) retire a lâmpada velha;

    f) coloque a lâmpada nova;

    g) enquanto a lâmpada não acender:

        - retire a lâmpada;
        - coloque outra lâmpada.

O passo "g" apresenta o que chamamos de fluxo repetitivo no qual um conjunto de passos pode se repetir n vezes. Na prática não é importante ter-se apenas um algoritmo, mas sim, um bom algoritmo. O mais importante de um algoritmo é a sua correção, isto é, se ele resolve realmente o problema proposto e o faz exatamente.

Para se ter um algoritmo, é necessário:

1. Que se tenha um número finito de passos.
2. Que cada passo esteja precisamente definido, sem possíveis ambiguidades.
3. Que existam zero ou mais entradas tomadas de conjuntos bem definidos.
4. Que existam uma ou mais saídas.
5. Que exista uma condição de fim sempre atingida para quaisquer entradas e num tempo finito.

1.4 Programas de computador

O computador, a princípio, não executa nada. Para que ele faça uma determinada tarefa, calcular uma folha de pagamento, por exemplo, é necessário que ele execute um programa. Um programa é um conjunto de milhares de instruções que indicam ao computador, passo a passo, o que ele tem que fazer. Logo, um programa nada mais é do que um algoritmo computacional descrito em uma linguagem de programação. Uma linguagem de programação contém os comandos que fazem o computador escrever algo na tela, realizar cálculos aritméticos, receber uma entrada de dados via teclado, e milhares de outras coisas, mas estes comandos precisam estar em uma ordem lógica. A Figura 1.2 mostra as fases da programação.

Fases da Programação

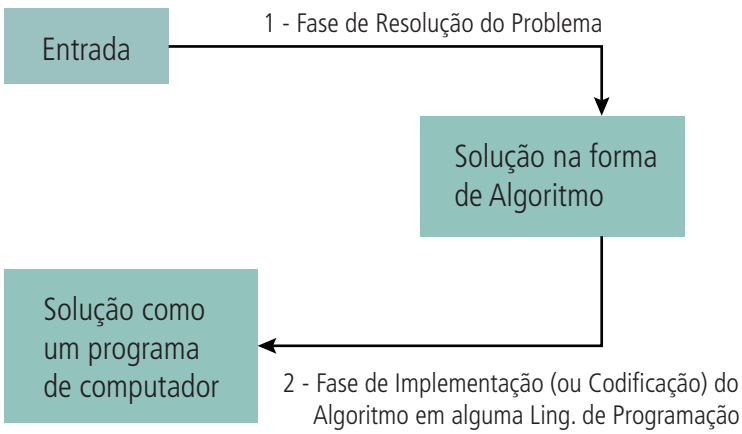


Figura 1.2: Fases da Programação
Fonte: Elaborada pelo autor

Todo programa fundamentalmente opera sobre um conjunto de entrada que representa os dados iniciais necessários à resolução do problema. Essas entradas são então processadas a partir de um conjunto de regras já definidas e, ao final, o programa gera um conjunto de saídas que representa o resultado do processamento.

Resumo

Nesta aula abordamos conceitos iniciais sobre Lógica de Programação e sua relação com a lógica matemática. Vimos a necessidade e a importância do uso da lógica em diversas áreas principalmente na área de Tecnologia da Informação. A construção de algoritmos é o primeiro passo para o desenvolvimento de programas de computador. É importante observar as etapas do desenvolvimento de um algoritmo e entender que um algoritmo não é

a solução do problema, mas uma forma de solucioná-lo e para um mesmo problema, podemos criar diferentes algoritmos usando diferentes abordagens. Sobre Programas de Computador é importante ressaltar que um programa nada mais é do que um algoritmo computacional descrito em uma linguagem de programação.

Atividades de aprendizagem

1. Analise as seguintes sentenças, assinalando (V) para Verdadeiro e (F) para Falso.

a) ( ) A Lógica de Programação é necessária para que o computador realize todas as tarefas de forma autônoma e criativa, sem a necessidade da intervenção humana.

b) ( ) A ordem em que os pensamentos são encadeados para se chegar a solução de um problema é relevante para a Lógica de Programação.

c) ( ) Um algoritmo representa uma sequência lógica de instruções.

d) ( ) Um problema pode ser solucionado por mais de um algoritmo.

e) ( ) Um algoritmo é criado para uma única Linguagem de Programação.

f) ( ) Um programa pode ser considerado como a codificação de um algoritmo em uma linguagem de programação.

2. Identifique os dados de entrada, do processamento e os dados de saída para o algoritmo da "troca da lâmpada" visto neste capítulo.

3. Construa um algoritmo simples para solucionar o problema de "trocar um pneu furado". Identifique os dados de entrada, processamento e saída.

4. Construa uma sequência de instruções (processamento) de acordo com os dados de entrada e saída abaixo:

Dados de entrada: ovo, gordura, sal
Dados de saída: ovo frito.

Aula 2 – Formas de representação de algoritmos

Objetivos

Conhecer e identificar as diversas formas de representação de algoritmos existentes.

Aplicar a melhor forma de representação de acordo com o algoritmo fornecido.

Com o passar do tempo e de estudos dos algoritmos, foram desenvolvidas inúmeras formas de se representar um algoritmo de modo a facilitar o seu entendimento e, mais tarde, a sua tradução para uma linguagem de programação específica. Entre as formas de representação de algoritmos mais conhecidas, podemos citar:

- Descrição Narrativa.
- Fluxograma.
- Diagrama de Chapin.
- Pseudocódigo, também conhecido como Português Estruturado ou Portugol.

2.1 Descrição narrativa

Nesta forma de representação os algoritmos são expressos diretamente em **linguagem natural**. Nessa forma de representação, os algoritmos são expressos diretamente em linguagem natural. Esta forma de representação é a mesma utilizada em algoritmos não computacionais, como receitas culinárias. Dessa forma, as instruções são descritas livremente, entretanto devemos tomar alguns cuidados para manter a clareza do algoritmo. Para escrever um algoritmo, precisamos descrever a sequência de instruções, de maneira simples e objetiva. Para isso devemos obedecer algumas regras básicas:

- Usar somente um verbo por frase;
- Imaginar que você está desenvolvendo um algoritmo para pessoas que não trabalham com informática;
- Usar frases curtas e simples;

- Ser objetivo;
- Procurar usar palavras que não tenham sentido dúbio.

Como exemplo, têm-se os algoritmos seguintes:

Troca de um pneu furado:

1. Afrouxar ligeiramente as porcas.
2. Suspender o carro.
3. Retirar as porcas e o pneu.
4. Colocar o pneu reserva.
5. Apertar as porcas.
6. Abaixar o carro.
7. Dar o aperto final nas porcas.

Cálculo da média de um aluno:

1. Obter as notas da primeira e da segunda prova.
2. Calcular a média aritmética entre as duas notas.
3. Se a média for maior ou igual a 7, o aluno foi aprovado, senão ele foi reprovado.

Esta representação é pouco usada na prática porque o uso de linguagem natural muitas vezes dá oportunidade a más interpretações, ambiguidades e imprecisões. Por exemplo, a instrução "afrouxar ligeiramente as porcas" no algoritmo da troca de pneus está sujeita a interpretações diferentes por pessoas distintas. Uma instrução mais precisa seria: "afrouxar a porca, girando-a de 30° no sentido anti-horário".

2.2 Fluxograma

Essa é a forma gráfica de representar um algoritmo mais conhecida e utilizada. O fluxograma nos permite mostrar graficamente a lógica de um algoritmo, enfatizando passos individuais e o fluxo de execução.

Para muitos autores, o fluxograma é a forma universal de representação, pois se utiliza de figuras geométricas padronizadas para ilustrar os passos a serem seguidos para a resolução de problemas. E o ideal é que um algoritmo seja entendido da mesma forma por diferentes pessoas que o utilizarem. Para que um fluxograma seja interpretado corretamente, é necessário entender sua **sintaxe** e sua **semântica**.

Sobre sintaxe de um fluxograma, devemos entender como a correta utilização dos seus símbolos gráficos mostrados na Figura 2.1 e das expressões que podem ser escritas no seu interior. Já a semântica diz respeito ao significado de cada símbolo, ou seja, como interpretá-lo corretamente. Com isso é possível entender e simular o algoritmo representado. A interpretação de um fluxograma, via de regra, se dá de cima para baixo e da esquerda para a direita. É importante seguir essa regra, pois garante que o fluxo de instruções seja entendido corretamente.

Por se tratar de uma representação gráfica, o fluxograma não se mostra adequado para a representação de algoritmos maiores e/ou mais complexos. Nesses casos, é comum o uso do pseudocódigo como forma de representação (Figura 2.1).

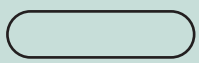
FIGURA	SIGNIFICADO
	Figura para definir início e fim do algoritmo
	Figura usada no processamento de cálculo, atribuições e processamento de dados em geral
	Figura utilizada na representação de entrada de dados
	Figura utilizada para representação da saída de dados
	Figura que indica o processo seletivo ou condicional, possibilitando o desvio no caminho do processamento
	Símbolo geométrico usado como conector
	Símbolo que identifica o sentido do fluxo de dados, permitindo a conexão entre as outras figuras existentes

Figura 2.1: Principais símbolos utilizados em um fluxograma
Fonte: adaptado de Sampaio, 2008

2.2.1 Construindo um fluxograma

Para entendermos como um fluxograma é criado para representar um algoritmo, vamos discutir passo-a-passo a criação de um fluxograma para representar um algoritmo que permita calcular a média final de um aluno considerando que todo aluno realiza três provas no semestre. O aluno é considerado "aprovado" se a sua média for igual ou superior a 7 (sete) , senão é considerado "reprovado".

Antes de iniciarmos a construção do fluxograma, devemos formalizar o algoritmo para solucionar o problema. É evidente que este é um problema de pouca complexidade e que envolve poucas regras. Para resolver esse problema, fica claro que precisaremos conhecer as três notas do aluno, que deverão ser informadas pelo usuário e, em seguida, calcular a média por meio do cálculo de média aritmética simples para a qual devemos aplicar a fórmula média = (nota1+nota2+nota3)/3. Após o cálculo da média, é necessário testar se o seu valor é igual ou superior ao valor 7 (sete) e, nesse caso, deverá ser listada a mensagem "Aprovado". Em caso contrário deverá ser listada a mensagem "Reprovado". Agora que conhecemos o algoritmo, é possível representá-lo como um fluxograma.

Inicialmente a forma mínima de um fluxograma é dada pela junção de seus terminadores de início e fim, como mostrado na Figura 2.2. Na verdade, essa construção mínima executa absolutamente nada. Os símbolos de INICIO e FIM, na verdade, não representam instruções de fato, mas são marcadores essenciais que permitem a correta interpretação do fluxograma (Figura 2.2).

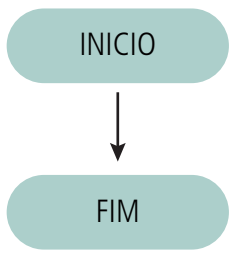


Figura 2.2: Fluxograma mínimo
Fonte: Adaptado de: Sampaio, 2008

Entretanto, para nosso algoritmo, será necessário ler os dados de entrada, nesse caso, as três notas do aluno. Com isso, é necessário expandir o nosso fluxograma, como mostra a Figura 2.3.

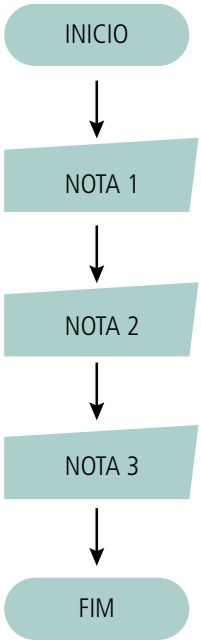


Figura 2.3: Fluxograma com dados de entrada
Fonte: Adaptado de Sampaio, 2008

Por motivos didáticos, para cada variável foi utilizado um símbolo de entrada manual. Na prática, entretanto, é comum a utilização de apenas um símbolo contendo todas as variáveis a serem lidas. De posse do valor de cada uma das três notas, é possível processar a média do aluno. Vamos representar esse processamento utilizando o símbolo de processo, como mostra a Figura 2.4.

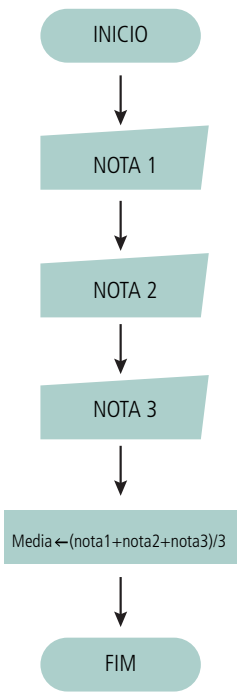


Figura 2.4: Fluxograma com cálculo da média
Fonte: Adaptado de Sampaio, 2008

Em um fluxograma, devemos sempre representar a atribuição de valor a uma variável por meio do símbolo "←", na forma **variável = valor**. Em um fluxograma, a expressão **variável = valor** representa o teste se o valor da variável é igual ao valor informado do lado direito da expressão e não uma atribuição.

Agora que o valor da média já foi calculado, é hora de testar o seu valor a fim de definir se o aluno foi aprovado ou reprovado. Nesse momento, sentimos a necessidade de controlar o fluxo de instruções, pois caso o valor da média seja superior a 7 (sete) devemos executar a instrução de impressão da mensagem "Aprovado", senão devemos apresentar a mensagem "Reprovado". Para fazer isso é necessário utilizar o símbolo de decisão, como mostrado na Figura 2.5. Como podemos notar, dependendo do resultado da condição **media >=7** , o fluxo de instruções é devidamente desviado.

Agora o fluxograma mostrado na Figura 2.4 reflete o algoritmo criado, representando cada passo de forma gráfica. Como já discutido na aula anterior, na maioria das vezes precisamos executar um conjunto de passos repetidas vezes, sem alterações no conjunto de instruções. Para representar esses casos em um fluxograma devemos construir um desvio no fluxo de instruções que permita testar uma condição de parada e, dependendo do seu resultado, retomar a um passo anterior a fim de repetir o conjunto de instruções desejado.

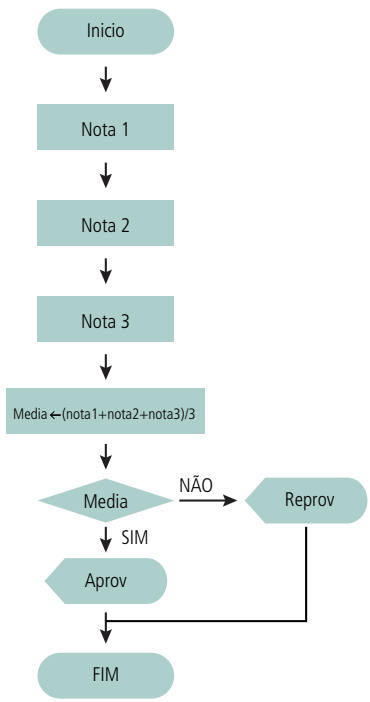


Figura 2.5: Fluxograma final
Fonte: Adaptado de Sampaio, 2008

A fim de compreender como tratar esse tipo de situação, vamos tomar como exemplo a representação de um algoritmo que imprime todos os números pares positivos e menores do que um valor N informado. O fluxograma resultante é mostrado na Figura 2.6.

Nesse fluxograma é possível notar que, após a leitura o valor N, este é então testado para saber se ainda é positivo. Em caso negativo, o programa já é encerrado, pois não haverá valores pares positivos menores do que N.

Caso a condição de teste seja satisfeita, ou seja, retome o valor verdadeiro, esse valor é testado para saber se é um valor par. Para isso, é testado o resto da divisão do valor N por dois. Caso essa expressão retorne o valor zero, o número é par, senão é impar. Caso a condição seja satisfeita - N é par – o valor de N é então exibido e, em seguida, decrementado de uma unidade.

Após essa instrução, o fluxo é então desviado para o momento anterior ao conjunto de passos que deverá ser novamente repetido. Com isso conseguimos garantir que esses passos serão executados até que a condição N > O retorne o valor falso.

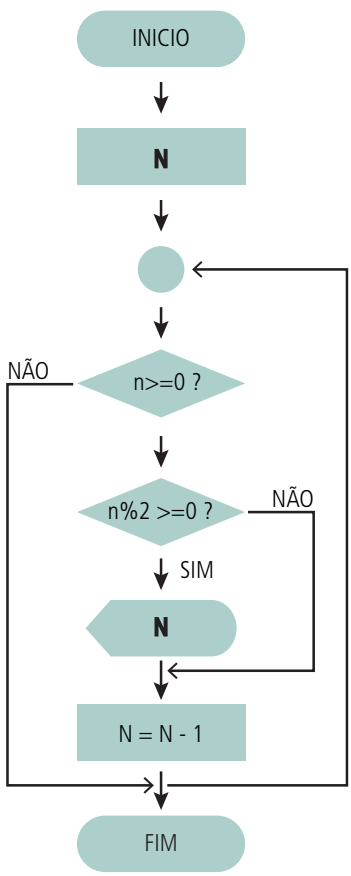


Figura 2.6: Fluxograma para impressão dos números pares e menores que um numero N.
Fonte: Adaptado de: (SAMPAIO, 2008)

2.3 Diagrama de Chapin

O diagrama de Chapin foi criado por Ned Chapin a partir de trabalhos de Nassi-Shneiderman, os quais resolveram substituir o fluxograma tradicional por um diagrama que apresenta uma visão hierárquica e estruturada da lógica do programa. A grande vantagem de usar este tipo de diagrama é a representação das estruturas que tem um ponto de entrada e um ponto de saída e são compostas pelas estruturas básicas de controle de sequência, seleção e repetição. Embora o diagrama de Chapin ofereça uma representação muito clara do algoritmo, à medida que os algoritmos vão se tornando mais complexos, fica difícil realizar os desenhos necessários numa única página, prejudicando a sua visualização. A Figura 2.7 apresenta um exemplo do tipo de diagrama de Chapin para o algoritmo de cálculo da média de um aluno.

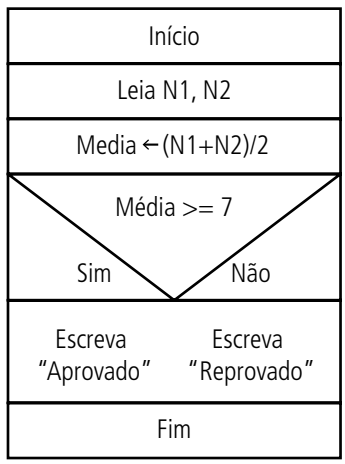


Figura 2.7: Diagrama de Chapin para o cálculo da média das notas de um aluno
Fonte: Elaborado pelo autor

2.4 Pseudocódigo

Esta forma de representação de algoritmos, também conhecida como português estruturado ou portugol, é bastante rica em detalhes e, por assemelhar-se bastante à forma em que os programas são escritos, tem muita aceitação, sendo portanto a forma de representação de algoritmos mais utilizada.

O pseudocódigo é um código de escrita em que se utilizam termos convencionais para indicar as instruções do programa. Esses termos são geralmente uma mistura de palavras da nossa linguagem natural com palavras e notações típicas das linguagens de programação.

A utilização de pseudocódigo permite ao programador expressar as suas ideias sem ter de se preocupar com a sintaxe da linguagem de programa-

ção. Para isso, são utilizadas primitivas (comandos genéricos) que podem ser facilmente traduzidos para uma linguagem de programação específica. As primitivas possuem a mesma função dos símbolos em um fluxograma, ou seja, descrever as ações a serem executadas e garantir a correta interpretação do algoritmo.

Entre as principais primitivas usadas, podemos citar:

ALGORITMO <nome> - primitiva usada para nomear o algoritmo representado;

INICIO (do inglês ***START***)- esta primitiva é usada para identificar o ponto inicial do algoritmo;

FIM (do inglês ***END***) - identifica o ponto final do algoritmo;

LEIA <mensagem>, <variavel> (do inglês ***INPUT***) - primitiva usada para a leitura de dados do utilizador. Equivalente à entrada de dados manual do fluxograma;

ESCREVA <mensagem>, <expressão> (do inglês ***OUTPUT***) – primitiva usada para a apresentação de dados ao utilizador. Equivalente ao símbolo de exibição do fluxograma;

<variavel> +- <expressão> - atribuição do resultado da expressão à variável indicada;

SE <condição> ENTAO (do inglês ***IF ... THEN***) <instruções a executar se condição verdadeira>

SENAO (do inglês ***ELSE***) <instruções a executar se condição falsa>

FIM SE (do inglês ***END IF***) - primitiva de controlo de fluxo equivalente à decisão dos fluxogramas;

ESCOLHA (<variavel» (do inglês ***SWITCH***)

CASO <valor 1>: <instruções a executar>

CASO <valor2>: <instruções a executar>

CASO <valorN>: <instruções a executar>

CASOCONTRARIO: <instruções a executar>

FIM ESCOLHA - primitiva de controlo de fluxo, usado para representar uma estrutura de controle de seleção múltipla;

ENQUANTO <condição> FAÇA (do inglês ***WHILE***) <instruções a executar enquanto a condição for verdadeira>

FIM ENQUANTO (do inglês ***END WHILE***) - primitiva de controlo de fluxo, sem equivalência direta em fluxogramas, que implementa um ciclo executado enquanto a condição referida seja verdadeira;

REPITA (do inglês ***REPEAT***) <instruções a executar enquanto a condição for verdadeira>

ATE <condição> (do inglês ***UNTIL***) - primitiva de controle de fluxo, sem equivalência direta em fluxogramas, que implementa um ciclo executado até que a condição referida seja verdadeira;

PARA <condição inicial> ATÉ <condição final> FAÇA [PASSO <incremento>] (do inglês ***FOR ... TO ... [STEP ...] DO)***

<instruções a executar enquanto a condição final for falsa>

FIM PARA (do inglês ***END FOR***) - primitiva de controlo de fluxo, que executa as instruções nela contidas enquanto a condição final for falsa. O incremento pode ser omitido desde que seja unitário positivo;

VAR <nome da variavel> [,]: <tipo da variavel> (neste caso é usado uma redução do termo VARIÁVEL - do inglês ***VARIABLE***) – primitiva utilizada na definição de variáveis. Os tipos de variáveis são discutidos em uma aula à frente;

CONST <nome da constante> = <valor da constante> [,] (neste caso é usado uma redução do termo CONSTANTE - do inglês ***CONSTAND*** - primitiva utilizada na definição de constantes. As constantes são discutidas à frente;

ESTRUTURA <nome da variavel>: <tipo da variavel> [,] (do inglês ***STRUCT***)

[<nome da variavel>: <tipo da variavel>]

FIM ESTRUTURA (do inglês ***END STRUCT***) - primitiva utilizada na definição de estruturas de variáveis. Os tipos de variáveis são definidos à frente;

FUNCAO <nome da função> «parâmetros da função»(do inglês ***FUNCTION***) <instruções da função>

[RETORNA <variavel>] (do inglês ***RETURN***)

FIM FUNCAO (do inglês ***END FUNCTION***) - primitiva utilizada na definição de funções. Por parâmetros entende-se uma lista dentro da função. Em certas situações, como veremos mais à frente, os parâmetros podem ser utilizados para a função exportar valores;

CHAMA <nome da função> «parâmetros da função» (do inglês ***CALL***) - primitiva utilizada para executar funções definidas com a primitiva anterior.

Um exemplo de pseudocódigo é mostrado a seguir:

```
ALGORITMO Media

VAR Nl, N2, N3, Media: real

INICIO

  LEIA Nl, N2, N3
  Media ß (N 1 + N2 + N3) / 2
  SE Media >= 7 ENTAO
    ESCREVA" Aprovado"
  SENAO
    ESCREVA "Reprovado"
  FIM SE
FIM.
```

Uma grande vantagem da utilização de pseudocódigo é o fato de permitir ao programador expressar as suas ideias sem ter de se preocupar com a sintaxe da Linguagem de Programação. Por outro lado, esse tipo de representação é o que mais se aproxima da codificação final em uma Linguagem de Programação e, por isso, é a mais utilizada na prática.

Resumo

Nesta aula aprendemos as principais formas de representação de algoritmos e suas aplicações em exemplos práticos. Vimos que na Descrição Narrativa, os algoritmos são expressos diretamente em linguagem natural. No entanto esta representação é pouco usada na prática porque o uso de linguagem natural muitas vezes dá oportunidade a más interpretações, ambiguidades e imprecisões. O fluxograma nos permite mostrar graficamente a lógica de um algoritmo, enfatizando passos individuais e o fluxo de execução. Para muitos autores, o fluxograma é a forma universal de representação, pois se utiliza de figuras geométricas padronizadas para ilustrar os passos a serem seguidos para a resolução de problemas. Como desvantagem, o fluxograma não se mostra adequado para a representação de algoritmos maiores e/ou mais complexos. O Diagrama de Chapin apresenta uma visão hierárquica e estruturada da lógica do programa. Apresenta como vantagem a representação das estruturas que tem um ponto de entrada e um ponto de saída e como desvantagem, esta representação se mostra confusa à medida que aumenta a complexidade do algoritmo. O pseudocódigo, também conhecido como português estruturado ou portugol, é bastante rico em detalhes e, por assemelhar-se bastante à forma em que os programas são escritos, tem muita aceitação, sendo por tanto, a forma de representação de algoritmos mais utilizada. A utilização de pseudocódigo permite ao programador expressar as suas ideias sem ter de se preocupar com a sintaxe da linguagem de programação.

Atividades de apredizagem

1. **Algoritmo**: Entrar com dois números inteiros e, após a leitura de cada número individual, verificar se o segundo número é positivo (maior que zero). Em caso afirmativo, divida o primeiro pelo segundo numero mostrando em seguida o resultado na saída. Em caso negativo, apresente o resultado na saída com a mensagem "Divisão não permitida". Apresente em seguida os dois números lidos na saída. Com base nesta descrição do algoritmo, represente-o nas seguintes formas:

   a) Descrição narrativa

   b) Fluxograma

   c) Diagrama de Chapin

   d) Pseudocódigo

2. Assinale a segunda coluna de acordo com a primeira:

(1)Descrição narrativa **(2)Fluxograma** **(3)Diagrama de Chapin** **(4)Pseudocódigo**	( ) Mais rica em detalhes e mais próxima das linguagens de programação ( ) Utilizada em algoritmos não-computacionais ( ) Utiliza de figuras geométricas padronizadas para ilustrar os passos a serem seguidos para a resolução de problemas. ( ) Dá oportunidade a más interpretações, ambiguidades e imprecisões. ( ) Oferece grande clareza para a representação de sequenciação, seleção e repetição num algoritmo, utilizando-se de uma simbologia própria. ( ) É indicado para algoritmos mais complexos.

3. Quais as vantagens e desvantagens da utilização de fluxograma e pseudocódigo na construção de algoritmos?

4. Dado o seguinte algoritmo representado em pseudocódigo, reescreva-o utilizando as representações:

a) Descrição Narrativa

b) Fluxograma

c) Diagrama de Chapin.

```
Algoritmo Maior
VAR
  num1, num2, maior : inteiro;
INICIO
        Leia num1,num2;
        SE (num1 > num2) ENTÃO
                maior ß num1;
        SENÃO
                maior ß num2;
        FIM SE
        ESCREVA maior;
FIM
```

5. Sobre o algoritmo em pseudocódigo apresentado abaixo, assinale (V) para Verdadeiro e (F) para falso nas afirmações a seguir:

```
ALGORITMO TESTE;
VARX: Inteiro;

INICIO
REPITA .
LEIA (X);
SE (X < O) ENTAO
ESCREVA (“O valor informado deve ser positivo”)
SENAO
ESCREVA(X*X);
FIM SE
ATE (X <= O)
FIM
```

( ) O pseudocódigo é a forma de representação menos utilizada na prática;

( ) Para esse algoritmo, caso o usuário informe um valor negativo, será emitida a mensagem “O valor informado deve ser positivo” e um novo valor será lido;

( ) Esse algoritmo não utiliza uma estrutura de decisão;

( ) Esse algoritmo não utiliza uma estrutura de repetição;

( ) Esse algoritmo imprime o quadrado de vários números informados pelo usuário até que seja informado um valor negativo ou nulo.

Aula 3 – Manipulação de dados

Objetivos

Entender os conceitos de variável e de constante e suas utilizações.

Utilizar adequadamente cada tipo de dado disponível.

Todo o trabalho realizado por um computador é baseado na manipulação das informações contidas em sua memória. Estas informações podem ser classificadas em dois tipos:

- As instruções, usadas para determinar o funcionamento da máquina e a maneira como os dados devem ser tratados. As instruções são específicas para cada arquitetura de computador, pois estão diretamente relacionadas às características do hardware particular como, por exemplo, o conjunto de instruções de cada processador;

- Os dados propriamente ditos, que correspondem às informações que deverão ser processadas pelo computador.

Por isso, é necessário que haja formas de se trabalhar com diferentes tipos de dados em um programa.

3.1 Tipo de dados

Podemos definir um tipo de dados como um conjunto de objetos que tem em comum o mesmo comportamento diante de um conjunto definido de operações.

Podemos dizer, por exemplo, que os seres humanos são um tipo. Todas as pessoas formam um conjunto de objetos que se comportam de forma similar quando comem, dormem, bebem, andam, etc. Essas ações são comuns aos seres humanos e definem o conjunto de operações que pode atuar sobre o tipo "ser humano". Outro exemplo poderiam ser os aviões. Podemos identificar sobre todos os objetos aviões operações similares como voar, pousar e decolar. Podemos representar o conjunto de pessoas como:

Pessoas = {Maria, João, Ana Paula, Júlio César,...}

Porém nem todo conjunto de objetos é um tipo. Por exemplo, {1, 'a', João Batista, Fusca} não é um tipo, pois seus elementos não tem comportamento similar segundo algum conjunto de operações.

Alguns tipos de dados são formados por números. Os números inteiros (Z) e os números reais (R), por exemplo, são um conjunto de elementos que suportam operações matemáticas como adição, subtração, multiplicação, etc. Esses tipos são particularmente importantes, visto que o computador trabalha naturalmente com números. Também podemos considerar as letras como um tipo, sobre as quais poderíamos definir operações como escrever, ler, concatenar, etc.

Infelizmente as linguagens de programação são limitadas e não podem manipular todos os tipos de dados existentes no mundo real. Muito pelo contrário, apenas um conjunto restrito de dados pode ser compreendido. A classificação apresentada a seguir não se aplica a nenhuma linguagem de programação específica, pelo contrário, ela sintetiza os padrões utilizados na maioria das linguagens.

3.1.1 Tipos inteiros

São caracterizados como tipos inteiros, os dados numéricos positivos ou negativos, excluindo-se destes qualquer número fracionário. Como exemplo deste tipo de dado, tem-se os valores: 35, 0, -56, 1024 entre outros.

3.1.2 Tipos reais

São caracterizados como tipos reais, aqueles que possuem parte decimal ou são números fracionários, e podem ser positivos, negativos ou zero. Exemplos de dados do tipo real são 3.2 (real positivo), 0.00 (zero real) e -19.76 (real negativo).

3.1.3 Tipos caracteres

São caracterizados como tipos caracteres, as sequências contendo letras, números e símbolos especiais. Uma sequência de caracteres deve ser indicada entre aspas (" "). Este tipo de dado também é conhecido como alfanumérico, string, literal ou cadeia. Como exemplo deste tipo de dado, tem-se os valores: "Programação", "Rua Alfa, 52 Apto 1", "Fone 574-9988", "04387-030", " ", "7" entre outros.

3.1.4 Tipos lógicos

São caracterizados como tipos lógicos os dados com valor **verdadeiro** e **falso**, sendo que este tipo de dado poderá representar apenas um dos dois valores. Ele é chamado por alguns de **tipo booleano**, devido à contribuição do filósofo e matemático inglês George Boole na área da lógica matemática.

3.1.5 Tipos de dados na linguagem de programação Pascal

Apresentamos nos Quadros 3.1 e 3.2 os tipos de dados numéricos suportados pela Linguagem de Programação Pascal, que será estudada nesse curso.

Quadro 3.1: Relação dos tipos de dados inteiros

Tipo	Nº de *Bytes*	Faixa de Valores
Short	1	-128 a 127
Integer	2	-32768 a 32767
Longint	4	-2.147.483.648 a 2.147.483.648
Byte	1	0 a 255
Word	2	0 a 65.535

Fonte: Adaptado de Medina *et al*, 2006

Quadro 3.2: Relação dos tipos de dados real

Tipo	Nº de *Bytes*	Dígitos Significativos	Faixa de Valores
Real	6	11-12	2,9e-39...1.7e38
Single	4	7-8	1,5e-45...3,4e38
Double	8	15-16	5,0e-324...1,7e308
Extended	10	19-20	3,4e-4932...1,1e4932
Comp	8	19-20	-9,2e18...9,2e18

Fonte: Adaptado de Medina *et al*, 2006

Os tipos de dados **caracteres** são formados por sequências contendo letras, números e (ou) símbolos especiais. Em Pascal são utilizados apóstrofes para delimitar a sequência de caracteres. Este tipo de dados é referenciado pelo identificador **string**, podendo armazenar de 1 até 255 caracteres. Semelhante ao identificador string existe também o tipo **char** que difere apenas por só poder armazenar um caractere.

Os tipos de dados lógicos no Pascal podem ser caracterizados por dois valores que são **true** (verdadeiro) e **false** (falso). Os tipos lógicos também são chamados de tipos booleanos e são referenciados pelo identificador **boolean**.

3.2 Variáveis e constantes

3.2.1 Armazenamento de dados na memória

Para armazenar os dados na memória, imagine que a memória de um computador é um grande arquivo com várias gavetas, onde cada gaveta pode armazenar apenas um único valor (seja ele numérico, caractere ou lógico). Se é um grande arquivo com várias gavetas, é necessário identificar com um nome a gaveta que se pretende utilizar. Desta forma o valor armazenado pode ser utilizado a qualquer momento.

3.2.2 Conceito e utilidade de variáveis

Têm-se como definição de variável tudo aquilo que é sujeito a variações, que é incerto, instável ou inconstante. E quando se fala de computadores, temos que ter em mente que o volume de informações a serem tratadas é grande e diversificado. Desta forma, os dados a serem processados serão bastante variáveis.

O computador, para poder trabalhar com alguma dessas informações, seja para ler seu conteúdo ou alterá-lo, precisa saber exatamente onde, na memória, o dado está localizado. Fisicamente, cada gaveta ou, cada posição de memória – esse é o termo técnico correto – possui um endereço, ou seja, um número que indica o seu endereço na memória, em outras palavras, onde cada informação está localizada. Esse número é normalmente representado usando notação hexadecimal, tendo o tamanho de quatro ou mais *bytes* – dependendo da arquitetura do computador. São exemplos de endereços físicos (Quadro 3.3a).

Quadro 3.3a: Endereços Físicos

END. FÍSICO	CONTEÚDO
7000:A210	'José Maria'
3000:12BC	987
4100:0012	54.987

Fonte: Elaborado pelo autor

O endereçamento das posições de memória por meio de números hexadecimais é perfeitamente compreendido pela máquina, mas para nós humanos torna-se uma tarefa complicada.

Para resolver esse problema, as linguagens de computador facilitam o manuseio, por parte dos programadores, das posições de memória da máquina, permitindo que, ao invés de trabalhar diretamente com os números hexa-

decimais, seja possível atribuir nomes diferentes a cada posição de memória. Além disso, tais nomes são de livre escolha do programador. Com esse recurso, os usuários ficaram livres dos endereços em hexadecimal e passam a trabalhar com endereços lógicos. Como o conteúdo de cada gaveta (endereço lógico) pode variar, ou seja, mudar de valor ao longo da execução do programa convencionou-se chamar de **variável** a referência a cada gaveta.

Assim, para os exemplos de endereços físicos mostrados anteriormente, poderíamos rotular a posição de memória **7000:A210** como **nome** e, como isso, sempre que precisássemos do valor dessa posição de memória, bastaria referenciar a variável **nome**. Assim ficaria o exemplo anterior (Quadro 3.3b).

Quadro 3.3b: Endereços Físicos

END. FÍSICO	END. LÓGICO	CONTEÚDO
7000:A210	nome	'José Maria'
3000:12BC	valor	987
4100:0012	total	54.987

Basicamente, uma variável possui três atributos: um **nome**, um **tipo de dado** associado à mesma e a **informação** por ela guardada. Toda variável possui um nome que tem a função de diferenciá-la das demais. Cada linguagem de programação estabelece suas próprias regras de formação de **nomes** de variáveis. Todavia a maioria delas adota as seguintes regras:

- um nome de variável deve necessariamente começar com uma letra;
- um nome de variável não deve conter nenhum símbolo especial, exceto a sublinha (_) e nenhum espaço em branco;
- um nome de variável não poderá ser uma palavra reservada a uma instrução de programa.

Exemplos de nomes de variáveis:

Salario – correto
1ANO – errado (não começou uma letra)
ANO1 – correto
a casa – errado (contém o caractere branco)
SAL/HORA – errado (contém o caractere "/")
SAL_HORA – correto
_DESCONTO – errado (não começou com uma letra)

Obviamente é interessante adotar nomes de variáveis relacionados às funções que serão exercidas pelas mesmas dentro de um programa. Outro atributo característico de uma variável é o **tipo de dado** que ela pode armazenar. Este atributo define a natureza das informações contidas na variável. Por último há o atributo **informação**, que nada mais é do que a informação útil contida na variável.

Uma vez definidos, os atributos **nome** e **tipo de dado** de uma variável **não** podem ser alterados e assim permanecem durante toda a sua existência, desde que o programa que a utiliza não seja modificado. Por outro lado, o atributo **informação** está constantemente sujeito a mudanças de acordo com o fluxo de execução do programa.

Em resumo, o conceito de variável foi criado para facilitar a vida dos programadores, permitindo acessar informações na memória dos computadores por meio de um nome, em vez do endereço de uma célula de memória.

3.2.3 Definição de variáveis em algoritmos

Todas as variáveis utilizadas em algoritmos devem ser definidas antes de serem utilizadas. Isto se faz necessário para permitir que o compilador reserve um espaço na memória para as mesmas. Mesmo que algumas linguagens de programação (como BASIC e FORTRAN) dispensam esta definição, uma vez que o espaço na memória é reservado à medida que novas variáveis são encontradas no decorrer do programa, nos algoritmos usaremos a definição de variáveis por assemelhar-se com as principais linguagens de programação como Pascal e C.

Nos algoritmos, todas as variáveis utilizadas serão definidas no início do mesmo, por meio de um comando de uma das seguintes formas:

VAR *<nome_da_variavel> : <tipo_da_variavel>*

ou

VAR *<lista_de_variaveis> : <tipo_das_variaveis>*

- a palavra-chave VAR deverá estar presente sempre e será utilizada uma única vez na definição de um conjunto de uma ou mais variáveis;
- numa mesma linha poderão ser definidas uma ou mais variáveis do mesmo tipo; Para tal, deve-se separar os nomes das mesmas por vírgulas;
- variáveis de tipos diferentes devem ser declaradas em linhas diferentes.

Exemplos de definição de variáveis:

```
VAR
nome: caracter[30]
idade: inteiro
salario: real
tem_filhos: logico
```

Traduzindo para a Linguagem Pascal, teríamos:

```
VAR
  nome: string[30];
  idade: integer;
  salario: real
  tem_filhos: boolean;
```

No exemplo acima foram declaradas quatro variáveis:

- a variável nome, capaz de armazenar dados caracteres de comprimento 35 (35 caracteres);
- a variável idade, capaz de armazenar um número inteiro;
- a variável salário, capaz de armazenar um número real;
- a variável tem_filhos, capaz de armazenar uma informação lógica.

3.2.4 Conceito e utilidade de constantes

Têm-se como definição de constante tudo aquilo que é fixo ou estável. Existirão vários momentos em que este conceito deverá estar em uso, quando desenvolvermos programas.

As constantes são muito utilizadas na programação principalmente em situações nas quais temos de referenciar um mesmo valor várias vezes ao longo do programa. Para esses casos, podemos definir um nome de constante e atribuir-lhe um valor. A partir daí basta referenciar o nome da constante e implicitamente estaremos nos referenciando ao seu valor. A principal vantagem dessa abordagem é a possibilidade de alterar o valor da constante apenas em sua definição e essa alteração ser automaticamente refletida em qualquer outro local do programa em que a sua referência é usada. Em casos como esse, sem o uso de constantes, será necessário alterar o valor em cada linha do programa onde ele foi usado.

3.2.5 Definições de constantes em algoritmos

Nos algoritmos, todas as constantes utilizadas serão definidas no início do mesmo, por meio de um comando da seguinte forma:

CONST <nome_da_constante> = <valor>

Exemplo de definição de constantes:

CONST PI = 3.14159

nome_da_empresa = "Enxuga Gelo SA"

Resumo

Nesta aula discutimos como o computador armazena e manipula as informações em sua memória. Os tipos de dados são importantes na construção de algoritmos e linguagens de programação. São eles: Inteiro, Real, Caractere e Lógico. Cada tipo de dado requer uma quantidade diferente de bytes para armazenar a sua informação na memória. Essa quantidade varia de acordo com o tipo de computador ou linguagem de programação específica. Uma variável é caracterizada por um nome para diferenciá-la das demais e permitir encontrar sua a sua posição (endereço) na memória e um tipo de dado que define o tipo de informação que ela é capaz de guardar. Uma vez definidos, os atributos **nome** e **tipo de dado** de uma variáve **não** podem ser alterados e assim permanecem durante toda a sua existência, desde que o programa que a utiliza não seja modificado. Por outro lado, o atributo **informação** está constantemente sujeito a mudanças de acordo com o fluxo de execução do programa. Já uma constante não sofre alteração no seu valor durante a execução do programa.

Atividades de aprendizagem

1. Sobre o uso de variáveis e constantes em um algoritmo é **incorreto** afirmar que:

   a) toda informação manipulada diretamente pelo computador é armazenada na memória principal (RAM) e as variáveis representam referências a posições dessa memória;

   b) o valor de uma variável não pode ser alterado ao longo do programa, mantendo o mesmo valor até o encerramento do programa;

   c) o tipo de dado implica na forma como os dados são representados na memória;

   d) os vetores e matrizes só podem manipular dados de um mesmo tipo de dado;

e) um dado constante não sofre alteração no seu valor durante a execução do programa.

2. Com base no que foi estudado nesta aula, escolha a alternativa que apresenta, respectivamente, os tipos de dados mais adequados para variáveis que deverão armazenar os seguintes conteúdos: idade, temperatura, nome da cidade, número da carteira de identidade, nota de um aluno.

a) Inteiro, real, caractere, caractere, real.

b) Inteiro, inteiro, caractere, caractere, inteiro.

c) Inteiro, real, inteiro, caractere, real.

d) Inteiro, real, real, caractere, inteiro.

e) Inteiro, real, caractere, real, real.

3. Classifique os conteúdos das variáveis abaixo de acordo com seu tipo, assinalando com (I) para Inteiro, (R) para Real, (C) para caractere e (L) para lógico.

( ) 0	( ) "abc"	( ) "João"
( ) 5.7	( ) 1012	( ) **FALSO**
( ) -49	( ) +342	( ) 569
( ) "Lucas"	( ) "VERDADEIRO"	( ) 0.00001
( ) **VERDADEIRO**	( ) -545	( ) " 444 "

4. Assinale com um X os nomes de variáveis válidos.

( ) abc	( ) 3abc	( ) a
( ) 123a	( ) –a	( ) acd1
( ) -_ad	( ) A&a	( ) guarda-chuva
( ) A123	( ) Aa	( ) guarda_chuva
( ) ABC DE	( ) etc.	( ) b316
( ) leia	( ) enquanto	( ) escreva

5. Comente sobre a vantagem do uso de variáveis e constantes em Algoritmos.

Aula 4 - Operadores e expressões

Objetivos

Identificar, utilizar e avaliar corretamente as expressões aritméticas e lógicas.

Identificar utilizar os principais operadores utilizados na programação de computadores.

4.1 Operadores

Operadores são elementos funcionais que atuam sobre termos (também chamados de operandos) e produzem um determinado resultado.

Os operadores são, na prática, instruções especiais pelas quais incrementamos, decrementamos, comparamos e avaliamos dados dentro de um programa de computador. Podemos classificar os operadores em três classes:

- operadores aritméticos;
- operadores relacionais;
- operadores lógicos.

Com o uso de operadores é possível construir expressões, assim como na matemática. A complexidade de uma expressão é determinada pela quantidade de operadores e termos (variáveis ou valores constantes). De acordo com o número de termos sobre os quais os operadores atuam, podemos classificá-los em:

- **Binários**: quando atuam sobre dois termos. Esta operação é chamada diadema. Como exemplo, temos os operadores aritméticos básicos: adição, subtração, multiplicação e divisão.

- **Unários**: quando atuam sobre um único termo. Esta operação é chamada monódica. Como exemplo, temos o sinal de negativo (-) na frente de um número, cuja função é inverter seu sinal.

4.1.1 Operadores aritméticos

Os operadores aritméticos são utilizados para obter resultados numéricos. Além da adição, subtração, multiplicação e divisão, grande parte das linguagens de programação também disponibilizam o operador para a operação de exponenciação. Os símbolos para os operadores aritméticos estão listados na tabela a seguir:

Operador	Tipo	Operação	Prioridade
+	Binário	Adição	4
-	Binário	Subtração	4
*	Binário	Multiplicação	3
/	Binário	Divisão	3
Operador	Tipo	Operação	Prioridade
MOD	Binário	Resto da Divisão	3
DIV	Binário	Divisão Inteira	3
**	Binário	Exponenciação	2
+	Unário	Manutenção do sinal	1
-	Unário	Inversão do sinal	1

A prioridade entre operadores define a ordem em que os mesmos devem ser avaliados dentro de uma mesma expressão, ou seja, operadores de prioridade menor serão avaliados primeiro. A avaliação da expressão – o que resulta no cálculo do seu resultado – depende da ordem na qual os operadores são processados. Via de regra, as expressões devem ser calculadas da esquerda para a direita, obedecendo a prioridade descrita na tabela acima. Se existirem expressões dentro de parênteses, estas terão prioridade sobre os demais operadores.

É importante salientar que, na Linguagem de Programação Pascal (assim como na maioria das linguagens de programação), o tipo de dado resultante depende dos tipos de dados dos termos. A seguir é apresentada uma tabela com os tipos de dados resultantes de cada operador.

OPERADOR	TIPO DE DADO A	TIPO DE DADO B	TIPO RESULTANTE
A + B	Inteiro	Inteiro	Inteiro
	Real	Real	Real
	Inteiro	Real	Real
	Real	Inteiro	Real
A – B	Inteiro	Inteiro	Inteiro
	Real	Real	Real
	Inteiro	Real	Real
	Real	Inteiro	Real

Continua

A * B	Inteiro	Inteiro	Inteiro
	Real	Real	Real
	Inteiro	Real	Real
	Real	Inteiro	Real
A / B	Inteiro	Inteiro	Real
	Real	Real	Real
	Inteiro	Real	Real
	Real	Inteiro	Real
A DIV B	Inteiro	Inteiro	Inteiro
A MOD B	Inteiro	Inteiro	Inteiro

Coclusão

4.1.2 Operadores relacionais

Os operadores relacionais são utilizados para comparar dados em um programa. Os valores a serem comparados podem estar armazenados em constantes, variáveis, valores numéricos ou literais. São operadores binários que devolvem os valores lógicos: verdadeiro e falso. Os operadores relacionais são:

SÍMBOLO	SIGNIFICADO
=	Igual a
< >	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a

Estes operadores são somente usados quando se deseja efetuar comparações. Comparações só podem ser feitas entre objetos de mesma natureza, isto é, variáveis do mesmo tipo de dado. O resultado de uma comparação é sempre um valor lógico.

Por exemplo, digamos que a variável inteira escolha contenha o valor 7. A primeira das expressões a seguir fornece um valor falso, e a segunda um valor verdadeiro:

```
escolha <= 5
escolha > 5
```

Com valores *string*, os operadores relacionais comparam os valores ASCII dos caracteres correspondentes em cada *string*. Uma *string* é dita "menor que" outra se os caracteres correspondentes tiverem os números de códigos ASCII menores. Por exemplo, todas as expressões a seguir são verdadeiras:

"algoritmo" > "ALGORITMO"
"ABC" < "EFG"
"Pascal" < "Pascal competir"

A-Z

ASCII
Sigla para *American Standard Code for Information Interchange*, que em português significa "Código Padrão Americano para o Intercâmbio de Informação" é uma codificação de caracteres de sete bits baseada no alfabeto inglês. Desenvolvida a partir de 1960, grande parte das codificações de caracteres modernas a herdaram como base.

Observe que as letras minúsculas têm códigos <PANT>**ASCII**<PANT> maiores do que os das letras maiúsculas. Observe também que o comprimento da *string* se torna o fator determinante na comparação de duas *strings*, quando os caracteres existentes na *string* menor são os mesmos que os caracteres correspondentes na *string* maior. Neste caso, a *string* maior é dita "maior que" a menor. Com valores lógicos, apenas os operadores de igualdade e a diferença estão disponíveis.

Vale ressaltar que não é eficaz comparar valores do tipo real com os operadores de igualdade em virtude de a representação em ponto flutuante ser inexata em alguns casos.

Apesar de algebricamente correta, a expressão (1.0/3.0) + (1.0/3.0) + (1 .0/3.0) = 1 é avaliada como falsa devido ao fato de 1.0/3.0 ter como resultado um valor que contém número infinito de casas decimais (3.3333333 ...). O computador é apenas capaz de utilizar um número finito de casas decimais e, portanto, é feito um arredondamento do valor de 1/3 em cada ocorrência. Para evitar os erros causados pela representação inexata de valores reais, deve-se evitar utilizar as comparações de igualdade com números reais.

4.1.3 Operadores lógicos

Os operadores lógicos – também conhecidos como operadores booleanos – servem para combinar resultados de expressões relacionais, devolvendo como resultado final os valores: verdadeiro ou falso.

Esse tipo de operador é amplamente usado na composição de expressões lógicas que são muito utilizadas nas estruturas de decisão e repetição em um programa. Os operadores lógicos são:

E (do inglês *AND*) - uma expressão desse tipo é verdadeira se todas as condições forem verdadeiras;

OU (do inglês *OR*) - uma expressão desse tipo é verdadeira se pelo menos uma das condições forem verdadeiras;

NÃO (do inglês *NOT*) - uma expressão desse tipo inverte o valor da expressão ou condição, se verdadeira inverte para falsa e vice-versa.

Os operadores lógicos e relacionais são elementos de fundamental importância na elaboração de um programa. Em todos os programas são utilizadas expressões relacionais e lógicas para a tomada de decisões e consequente desvio do fluxo do programa.

O resultado de uma operação lógica vai depender dos valores dos termos submetidos. A seguir é apresentada uma tabela com os tipos de dados resultantes de cada operador.

OPERADOR	TERMO 1	TERMO 2	RESULTADO
E	Verdadeiro	Verdadeiro	Verdadeiro
	Verdadeiro	Falso	Falso
	Falso	Verdadeiro	Falso
	Falso	Falso	Falso
OU	Verdadeiro	Verdadeiro	Verdadeiro
	Verdadeiro	Falso	Verdadeiro
	Falso	Verdadeiro	Verdadeiro
	Falso	Falso	Falso
NÃO	Verdadeiro	-	Falso
	Falso	-	Verdadeiro

O quadro a seguir mostra a prioridade do uso dos operadores lógicos:

OPERADOR	OPERAÇÃO	PRIORIDADE
NÃO	NEGAÇÃO	1
E	CONJUNÇÃO	2
OU	DISJUNÇÃO	3

Por exemplo, na expressão:

(A=B) .OU. .NÃO. (A<9)

Devemos avaliar primeiro: .NÃO. (A<9).

4.1.4 Operador de concatenação

Esse operador é usado para concatenar, ou seja, juntar dois valores ou variáveis do tipo texto (literal). Normalmente utilizamos esse tipo de operador para compor mensagens ao longo do programa.

Na Linguagem de Programação Pascal (assim como na maioria das linguagens de programação), o símbolo usado para a concatenação é o mesmo da adição. Assim a expressão "Programar"+"é fácil" teria como resultado o

literal "Programar é fácil". É comum utilizar a concatenação em expressões envolvendo variáveis. Caso a variável seja do tipo literal, esse operador pode ser usado diretamente; caso contrário será necessário algum tipo de conversão a fim de torná-lo um literal. Na verdade, a maioria das linguagens de programação traz funções para esse propósito.

4.1.5 Operador de atribuição

O operador de atribuição é usado para definir o valor de uma variável. Na prática, para executar essa operação, o computador preenche a posição de memória apontada pela variável com o valor posicionado do lado direito do operador de atribuição. Em algoritmo, é representado pelo símbolo: ←. A sintaxe de um comando de atribuição é:

NomedaVariavel* ← *expressao

A expressão: **custo ← 23** implica em atribuir o valor 23 à variável chamada **custo**. É comum o uso de expressões aritméticas ou relacionais no lado direito desse operador. Nesses casos, o valor resultante da expressão é transferido para a variável.

É importante ressaltar que o termo do lado esquerdo do operador de atribuição deve sempre ser um identificador de variável, de outra forma não haverá onde armazenar o valor posicionado no lado direito. O tipo da variável (termo do lado esquerdo) deverá ser do mesmo tipo do resultado da expressão (termo do lado direito), caso contrário, a operação de atribuição estará inválida e provavelmente causará erro durante tradução para a linguagem de programação Pascal.

Em Pascal, o símbolo de atribuição é representado por: "**:=**". Traduzindo a expressão acima teríamos:

NomedaVariavel* := *expressao

4.2 Expressões

O conceito de expressão, em termos computacionais, está intimamente ligado ao conceito de expressão (ou fórmula) usado na matemática, na qual um conjunto de variáveis e constantes numéricas relaciona-se por meio de operadores aritméticos compondo uma fórmula que, uma vez avaliada, resulta em um valor final.

O conceito de expressão aplicado à computação assume um sentido mais amplo: uma expressão é uma combinação de variáveis, constantes e operadores (aritméticos, relacionais ou lógicos) e que, uma vez avaliada, resulta em um valor.

4.2.1 Expressões aritméticas

Expressões aritméticas são aquelas cujo resultado da avaliação é do tipo numérico, seja ele inteiro ou real. Somente o uso de operadores aritméticos, variáveis numéricas e parênteses é permitido em expressões deste tipo.

4.2.2 Expressões lógicas

Expressões lógicas são aquelas cujo resultado da avaliação é um valor lógico verdadeiro ou falso. Nestas expressões são usados os operadores relacionais e os operadores lógicos, podendo ainda serem combinados com expressões aritméticas. Quando forem combinadas duas ou mais expressões que utilizem operadores relacionais e lógicos, os mesmos devem utilizar os parênteses para indicar a ordem de precedência.

4.2.3 Avaliação de expressões

Expressões que apresentam apenas um único operador podem ser avaliadas diretamente. No entanto, à medida que as mesmas vão tornando-se mais complexas com o aparecimento de mais de um operando na mesma expressão, é necessária a avaliação da mesma passo a passo, tomando um operador por vez. A sequência destes passos é definida de acordo com o formato geral da expressão, considerando-se a prioridade (precedência) de avaliação de seus operadores e a existência ou não de parênteses na mesma.

As seguintes regras são essenciais para a correta avaliação de expressões:

1. Deve-se observar a prioridade dos operadores, conforme mostrado nas tabelas de operadores: operadores de maior prioridade devem ser avaliados primeiro. Se houver empate com relação à precedência, então a avaliação se faz da esquerda para a direita.

2. Os parênteses usados em expressões têm o poder de "roubar" prioridade dos demais operadores, forçando a avaliação da subexpressão em seu interior.

3. Entre os três grupos de operadores existentes, a saber: aritmético, lógico e relacional, há certa prioridade de avaliação: os aritméticos devem ser avaliados primeiro; a seguir, são avaliadas as subexpressões com operadores relacionais e, por último os operadores lógicos são avaliados.

Por exemplo, vamos analisar a seguinte expressão e determinar o resultado final:

5>=6 .OU. 6<7 .E. (.NÃO.(a+5-6=8 .OU. a mod 2 > 5)), para a=5.

Seguindo as regras para avaliação de expressões, vamos avaliar a subexpressão que está nos parênteses mais internos. Sendo assim, vamos calcular em primeiro lugar a subexpressão:

(a+5-6=8 .OU. a mod 2 > 5)

Vamos substituir o valor da variável a por 5:

(5+5-6=8 .OU. 5 mod 2 > 5).

Avaliando as expressões aritméticas: 5+5-6 e 5 mod 2, temos:

(4=8 .OU. 1 > 5).

Avaliando as expressões relacionais: 4=8 e 1 > 5, temos:

(.Falso. .OU. .Falso.).

Avaliando a expressão lógica, temos como resultado o valor: **Falso**. Substituindo o valor encontrado na expressão inicial, temos:

5>=6 .OU. 6<7 .E. (.NÃO.(.Falso.))

Podemos perceber que a expressão ficou mais simples. Continuaremos a avaliar as expressões entre parênteses, começando com os mais internos.

Avaliando a expressão lógica: .**NÃO.(.Falso.),** temos como resultado o valor: **Verdadeiro**. Substituindo na expressão, temos:

5>=6 .OU. 6<7 .E. .Verdadeiro.

Agora que os parênteses foram eliminados, podemos avaliar as expressões relacionais:

5>=6 e 6<7. Substituindo os resultados na expressão, temos:

.Falso. .OU. .Verdadeiro. .E. .Verdadeiro.

A expressão inicial foi reduzida a uma expressão lógica envolvendo dois operadores: .OU. e .E. Segundo as regras de prioridade, resolveremos primeiro a expressão: **.Verdadeiro. .E. .Verdadeiro**. Sendo assim, temos:

.Falso. .OU. .Verdadeiro.

Enfim conseguimos reduzir a expressão inicial a uma simples expressão lógica envolvendo um operador .OU., tendo como resultado final o valor: **Verdadeiro**. Então podemos concluir que o resultado da expressão:

5>=6 .OU. 6<7 .E. (.NÃO.(a+5-6=8 .OU. a mod 2 > 5)), para a=5, tem como resultado final o valor: **Verdadeiro.**

Resumo

Nesta aula foram apresentados e discutidos os principais operadores Aritméticos, Relacionais e Lógicos usados na Programação de computadores. Também foi apresentado e discutido a definição, a construção e avaliação de expressões que utilizam esses operadores. Na avaliação de expressões lava-se em consideração a junção dos três operadores em uma mesma expressão. À medida que as expressões vão tornando-se mais complexas com o aparecimento de mais de um operando na mesma expressão, é necessária a avaliação da mesma passo a passo, tomando um operador por vez. A sequência destes passos é definida de acordo com o formato geral da expressão, considerando-se a prioridade (precedência) de avaliação de seus operadores e a existência ou não de parênteses na mesma.

Atividades de aprendizagem

1. Assinale a alternativa correta:

a) O operador > (maior que) pode ser classificado como operador relacional do tipo unário.

b) O resultado de uma expressão aritmética será um valor lógico.

c) O resultado de uma expressão lógica sempre será um valor lógico.

d) Um operador relacional usado para comparar dois números inteiros sempre retornará um valor inteiro.

e) O operador .OU. é classificado com operador lógico do tipo binário e tem prioridade sobre os demais operadores lógicos.

2. Das alternativas apresentadas abaixo indique aquela que não devolve um valor lógico. Considere os seguintes valores para as variáveis: Mediaß7, Notaß7.25, Valor ← verdadeiro e Opção ← "a".

a) ((Nota >= Media) E (Opção="b"))

b) (((Nota + 1) > Media) OU (NAO Valor))

c) (((Média*2)-1) MOD 2)

d) (((Media DIV 7) = 1) OU (Nota >= Media))

e) (((Nota - Media) > 3) = Valor)

3. Qual o valor final das variáveis A e B, após a execução das instruções abaixo ?

A ← 3
B ← A*5
A← B DIV 2
B ← A – 10

4. Se X possui o valor 15 e foram executadas as seguintes instruções:

X ← X + 3
X ← X - 6
X ← X / 2
X ← 3 * X

Qual será o valor armazenado em X?

5. Assinalar os comandos de atribuição considerados inválidos:

var
NOME, COR, TESTE, DIA: **caracter**
SOMA, NUM: **inteiro**
Salario: **real**
X: **logico**

a) ( ) NOME ← "5"

b) ( ) SOMA ← NUM + 2 * X

c) ( ) TESTE ← SOMA

d) ( ) NUM ← SOMA

e) ( ) COR ← "PRETO"

f) ( ) X ← X + 1

g) ( ) NUM ← "*ABC*"

h) ( ) DIA ← "SEGUNDA"

i) ( ) SOMA + 2 ← NUM

j) ( ) X ← (NOME = COR)

k) ( ) salário ← 5.000

l) ( ) salário ← 150

m) ( ) salário ← "insuficiente"

6. Quais os valores armazenados em SOMA, NOME e TUDO, supondo-se que NUM, X, COR, DIA,TESTE e TESTE2 valem, respectivamente, 5, 2, "AZUL", "TERÇA", FALSO e VERDADEIRO?

a) NOME ← DIA

b) SOMA ← (NUM**2/X) + (X + 1)

c) TUDO ← .NÃO. ((TESTE .OU. TESTE2) .E. (X <> NUM))

7. Resolva as expressões lógicas, determinando se a expressão é verdadeira ou falsa:

a) 2>3 =

b) (6<8) .ou. (3>7) =

c) não (2<3) .e. (2 mod 1> 1 .ou. 0/1 >=0) =

d) (34>9 .e. 5+u = 34) .ou. (5=15/3 .e. 8>12) = ((u = 29) .e. 8>12), {onde u = 29}

8. Considere a seguinte expressão ((((AMOD 5) > 5) OU (B/C >= 1)) E ((NAO ((A<50) E (B < > C)) OU (C=5))) e determine o valor para essa expressão, tendo os valores 23, 5 e 5 como valores das variáveis A, B e C, respectivamente.

Respostas

1. LETRA "C"

2. LETRA "C"

3. A = 7, B = -3

4. X = 18

5. b, c, f, g, i

6. NOME = "TERÇA"; SOMA= 15,5; TUDO = FALSO

7. a) falso, b) verdadeiro, c) falso, d) falso

8. Verdadeiro

Aula 5 - Instruções primitivas

Objetivos

Identificar uma instrução e utilizar blocos de instruções no desenvolvimento de algoritmos.

Conhecer e utilizar os comandos de entrada e saída de dados.

Identificar e aplicar os conceitos de entrada, processamento e saída na resolução de algoritmos.

Aplicar corretamente o teste de mesa nos algoritmos a fim de verificar se a lógica aplicada está correta.

Como o próprio nome diz, **instruções primitivas** são os comandos básicos que efetuam tarefas essenciais para a operação dos computadores, como entrada e saída de dados (comunicação com o usuário e com dispositivos periféricos), e movimentação dos mesmos na memória. Estes tipos de instrução estão presentes na absoluta maioria das linguagens de programação. Para entendermos a definição das instruções primitivas, precisamos definir alguns termos:

- **Dispositivo de entrada** é o meio pelo qual as informações (mais especificamente os dados) são transferidas pelo usuário ou pelos níveis secundários de memória ao computador. Os exemplos mais comuns são o teclado, o mouse, leitora ótica, leitora de código de barras, as fitas e discos magnéticos.
- **Dispositivo de saída** é o meio pelo qual as informações (geralmente os resultados da execução de um programa) são transferidas pelo computador ao usuário ou aos níveis secundários de memória. Os exemplos mais comuns são o monitor de vídeo, impressora, fitas e discos magnéticos.
- **Sintaxe** é a forma como os comandos devem ser escritos, a fim de que possam ser entendidos pelo tradutor de programas. A violação das regras sintáticas é considerada um erro sujeito à pena do não reconhecimento por parte do tradutor.
- **Semântica** é o significado, ou seja, o conjunto de ações que serão exercidas pelo computador durante a execução do referido comando.

As instruções de uma linguagem de programação sempre serão executadas em sequência. Portanto, podemos considerar o processo de construção de um algoritmo como o encadeamento de instruções individuais que, em seu conjunto, formarão o algoritmo.

5.1 Blocos de instruções

A utilização de blocos de instruções deixa claro onde a sequência de instruções inicia e termina. Um bloco apresenta o início e o término do conjunto de instruções que serão executadas em sequência. Todo o algoritmo tem pelo menos um bloco de instruções, que indica o início e o final do próprio algoritmo. O bloco de instruções é construído como se segue:

```
< declaração de variáveis >
Inicio
        < primeira instrução do bloco >
        < segunda instrução do bloco >
        ...
        < última instrução do bloco >
Fim
```

Se juntarmos as instruções e a declaração de variáveis e constantes que já apresentamos, formaremos um código válido: Exemplo:

```
CONST maximo = 100;
VAR
  quantidade, filhos, netos : inteiro;
  aberto : booleano;
  altura : real;
  resposta : caracter;
INICIO
  altura ← 1.80;
  filhos ← 3;
  netos ← filhos * 2 – 3;
  quantidade ← maximo -10;
  aberto ← .Falso.;
  aberto ← aberto .E. .Verdadeiro.;
  aberto ← quantidade < maximo;
  resposta ← 'S';
FIM
```

Segue abaixo algumas funções matemáticas para utilizarmos na resolução de algoritmos:

ABS (x)	Retorna o valor absoluto de uma expressão
ARCTAN (x)	Retorna o arco de tangente do argumento utilizado
COS (r)	Retorna o valor do co-seno
EXP(r)	Retorna o valor exponencial
FRAC (r)	Retorna a parte fracionária
LN (r)	Retorna o logaritmo natural
PI	Retorna o valor de PI
SIN (r)	Retorna o valor do seno
SQR (r)	Retorna o parâmetro elevado ao quadrado.
SQRT (r)	Retorna a raiz quadrada

A partir deste momento, já temos ferramentas suficientes para construirmos algoritmos simples. De forma genérica, a construção de um algoritmo se resume às seguintes etapas:

a) entendimento do problema;

b) elaboração da solução algorítmica; e

c) codificação da solução no Português Estruturado.

Problema 1: Sabendo que a área de um círculo é dado pela fórmula: Área = πr^2, faça um algoritmo para calcular a área de um círculo de raio = 2.

Etapa 1: dos tempos de escola, sabemos que na fórmula do calculo da área de um círculo, o π(PI) representa uma constante real cujo valor é : 3,1416... Para simplificarmos nosso cálculo, vamos utilizar apenas as duas primeiras casas decimais da constante PI. Sabendo que r representa o raio do círculo, podemos prosseguir com a resolução do algoritmo.

Etapa 2: o próximo passo é então montar a solução algorítmica que represente a fórmula matemática do cálculo da área do circulo. Para isso temos que declarar as variáveis necessárias para representar a solução:

```
Área ← PI * sqr(raio);
```

Solução:

```
Algoritmo Area_circulo;
CONST PI=3,14;
VAR
  Area, raio : real;
INICIO
  raio ← 2;
  Area ← PI * sqr(raio);
FIM
```

Problema 2: Faça um algoritmo para encontrar as raízes de uma equação do segundo grau para a seguinte equação: 2x2 + 4x – 3.

De forma geral, chama-se equação do segundo grau com uma variável toda equação que pode ser escrita da forma: ax2 + bx + c, em que x é a variável e "a", "b" e "c" são os coeficientes da equação do segundo grau.

"a" representa o coeficiente de x2
"b" representa o coeficiente de x.
"c" representa o termo independente.

Para encontrar as raízes reais de uma equação de segundo grau, podemos utilizar a fórmula:

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

$$x'' = -b-\sqrt{\Delta}$$

Solução:

```
Algoritmo Raízes;
VAR
   Delta, x1, x2, a, b, c : real;
INICIO
   a ← 2;
   b ← 4;
   c ← 3;
   delta ← sqr(b) – 4*a*c;
   x1 ← (-b + (sqrt (delta) / 2*a);
   x2 ← (-b – (sqrt (delta) / 2*a);
FIM
```

5.2 Entrada, processamento e saída

Um programa pode ser visto de forma sistêmica. Por esta abordagem, esperamos que um sistema/programa seja composto de uma entrada, processamento e saída. Sendo assim, todo programa estará trabalhando com estes três conceitos. Se os dados forem entrados de forma errada, serão consequentemente processados de forma errada e resultarão em respostas erradas. Desta forma, dizer a alguém que foi erro do computador é ser um tanto "medíocre". E isto é o que mais ouvimos quando nosso saldo está errado e vamos ao banco fazer uma reclamação, ou quando recebemos uma cobrança indevida. Se houve algum erro, é porque foi causado por falha humana. Realmente é impossível um computador errar por vontade própria, pois vontade é uma coisa que os computadores não têm.

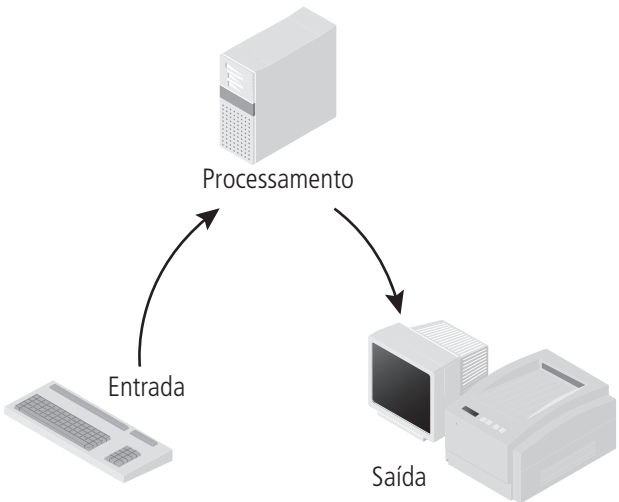


Figura 5.1: Estrutura de um sistema/programa
Fonte: Elaborada pelo autor

Na receita de bolo (Aula 1), os ingredientes podem ser considerados a entrada, o preparo, o processamento e o bolo, a saída. Existem muitos meios de entrada e saída em um computador. Como dispositivos de entrada, podemos citar o teclado, o mouse, o scanner etc. Como saída, temos o vídeo, a impressora etc. O periférico de entrada mais comumente utilizada é o teclado. Para a saída temos o vídeo. Convencionaremos que tais interfaces serão utilizadas sempre que tratarmos de entrada e saída, a não ser que seja explicitamente definido outro periférico. Os algoritmos elaborados na seção 5.1 são interessantes, mas falta-lhes adaptabilidade.

O primeiro calcula a área de um círculo para um raio específico, enquanto o segundo é capaz de encontrar as raízes reais de uma equação de segundo grau. Porém, ambos os algoritmos sempre que executados resolvem apenas

um único problema: o cálculo da área de um círculo de raio 2 e a resolução das raízes da equação 2x2+ 4x - 3. Se tivermos a necessidade de calcular a área de outro círculo de raio qualquer, ou de descobrir as raízes de outra equação de segundo grau, como x2+ 2x + 1, teremos que reescrever o algoritmo.

Se todos os programas fossem assim, para que um usuário pudesse utilizar um programa, necessitaria ter acesso ao código-fonte, adaptar o algoritmo à sua necessidade, compilá-lo e, então, executá-lo. Sempre que precisasse resolver o mesmo problema para valores diferentes, teria que refazer todo esse processo. Entretanto, na computação não se espera que o usuário tenha essas habilidades. O conhecimento de alterar programas e compilá-los é inerente ao desenvolvedor/programador. O usuário apenas utiliza programas disponibilizados pelo desenvolvedor para atender às suas necessidades.

Para que o usuário pudesse utilizar o programa para atender às suas necessidades específicas, seria necessário, então, que o desenvolvedor readaptasse seu código a cada nova situação. Esse processo é extremamente trabalhoso e inviável. Além disso, em muitos casos o desenvolvedor não tem contato com seus usuários, pois seu programa pode ser utilizado inclusive por pessoas que ele nem conhece.

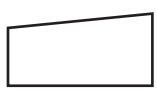
Para que o programa possa atender às diferentes necessidades do usuário, basta que este possa informar ao programa quais os valores que este deve processar e, então, aguardar resultado. Conforme a abordagem sistêmica citada no inicio desta seção, teremos o esquema **Entrada -> Processamento -> Saída**.

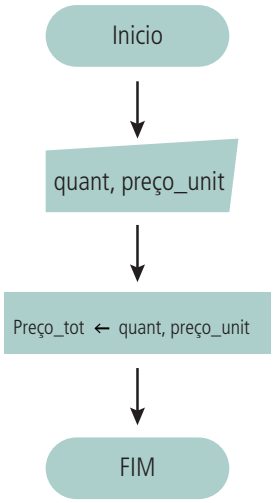
5.2.1 Entrada

A entrada é o meio pelo qual o usuário pode informar dados que serão utilizados pelo programa em seu processamento. Desta forma, não haverá necessidade de adaptação do código para atender à maioria das demandas particulares do usuário. A entrada é feita pelos comandos:

```
Leia (<variavel>);
Leia (<lista_de_variaveis>)
```

No diagrama de blocos o comando de entrada de dados é representado por:

 = Operação de entrada de dados



A leitura de um valor implica necessariamente o uso de uma variável, pois tal valor precisa ser armazenado em uma posição de memória para futura manipulação.

A seguir temos um exemplo de um algoritmo utilizando o comando de entrada de dados:

```
Algoritmo exemplo_comando_de_entrada_de_dados
Var preco_unit, preco_tot : real
quant : inteiro
Inicio
Leia (preco_unit, quant);
preco_tot ß preco_unit * quant;
Fim
```

5.2.2 Saída

Para que o usuário possa ter acesso aos resultados do processamento do programa, toda linguagem de programação fornece mecanismos de apresentação (saída) dos dados. A saída é feita pelos comandos:

- **ESCREVA** <*variavel*>
  Ex: **ESCREVA** X
- **ESCREVA** <*lista_de_variaveis*>
  Ex: **ESCREVA** nome, endereco, cidade
- **ESCREVA** <*literal*>
  Ex: **ESCREVA** "Algoritmo é o máximo!"
- **ESCREVA** <*literal*>, <*variavel*>, ... ,<*literal*>, <variavel>
  Ex: **ESCREVA** "Meu nome é:", nome, "e meu endereço é:", endereco

No diagrama de blocos o comando de saída de dados é representado por:

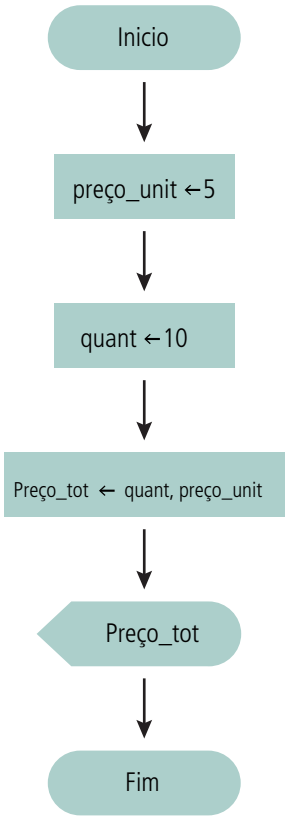
 = Operação de saída de dados em vídeo

 = Operação de saída de dados em impressora

O valor a ser escrito na tela do monitor pode ser um texto (neste caso, entre aspas) ou o conteúdo de uma variável/constante. Os argumentos do comando são enviados para o dispositivo de saída. No caso de uma lista de variáveis, o conteúdo de cada uma delas é pesquisado na memória e enviado para o dispositivo de saída. No caso de argumentos do tipo literal ou string, estes são enviados diretamente ao referido dispositivo. Há ainda a possibilidade de se misturar nomes de variáveis com literais na lista de um mesmo comando. O efeito obtido é bastante útil e interessante: a lista é lida da esquerda para a direita e cada elemento da mesma é tratado separadamente; se um nome de variável for encontrado, então a informação da mesma é colocada no dispositivo de saída; no caso de um literal, o mesmo é escrito diretamente no dispositivo de saída.

A seguir temos um exemplo de um algoritmo utilizando o comando de saída de dados:

```
Algoritmo exemplo_comando_de_saida_de_dados
Var preco_unit, preco_tot : real
quant : inteiro
Inicio
preco_unit ← 5.0
quant := 10
preco_tot := preco_unit * quant
Escreva preco_tot
Fim.
```



Agora que aprendemos os comandos de entrada e saída e seus objetivos, vamos agora refazer o algoritmo descrito na Seção 5.1. O algoritmo inicial era:

- Sabendo que a área de um círculo é dado pela fórmula: Área = πr2, faça um algoritmo para calcular a área de um círculo de raio = 2.

- Agora, vamos construir o algoritmo:

- Sabendo que a área de um círculo é dado pela fórmula: Área = πr2, faça um algoritmo para calcular a área de um círculo de raio de raio qualquer.

```
Algoritmo Area_Circulo;
CONST PI=3,14;
VAR
  raio,area : real;
INICIO
  Leia(raio);
  area ← PI * sqr(raio);
  Escreva('A área do circulo é:',area);
FIM;
```

Fazendo a adaptação para o algoritmo do cálculo das raízes de uma equação do segundo grau, teríamos ao invés das raízes para a equação específica: 2x2+ 4x – 3, as raízes para uma equação do segundo grau qualquer: ax2+ bx + c, onde os coeficientes a, b e c seriam lidos.

```
Algoritmo Eq2grau;
VAR
  a,b,c: inteiro;
  delta,x1,x2: real;
INICIO
  LEIA(a,b,c);
  delta ← sqr(b) – 4*a*c;
  x1 ← -b + sqrt(delta)/(2*a);
  x2 ← -b - sqrt(delta)/(2*a);
  ESCREVA('As raízes são:',x1,x2);
FIM
```

5.2.3 Teste de mesa

Para o programador iniciante (e às vezes para os não tão iniciantes assim), é difícil verificar se o algoritmo construído realiza realmente a tarefa para a qual foi projetado. Aliás, a correção de um algoritmo é uma das características essenciais a qualquer algoritmo.

Podemos dizer que um algoritmo só é correto se produz o resultado esperado para qualquer entrada possível. As entradas possíveis podem ser restritas na proposição do algoritmo. É importante destacar que muitos algoritmos funcionam de forma adequada para a maioria das entradas, porém apresentam resultados inválidos ou incorretos para determinadas entradas. Neste caso, consideramos o algo ritmo incorreto.

O teste de mesa é um meio pelo qual podemos acompanhar a execução de um algoritmo, passo a passo, ou instrução a instrução. Desta forma, podemos encontrar erros e confirmar se a lógica do algoritmo está correta. Para acompanhar o desenrolar de um algoritmo é importante verificar o estado dos dados a cada instrução, verificando o Conteúdo de todas as variáveis definidas no algoritmo. Para facilitar o acompanhamento dos algoritmos, colocaremos o número das linhas que contenham instruções. Vejamos o exemplo do algoritmo para o cálculo de uma equação do 2º grau, para os valores de a=1, b=-4 e c=3.

```
        Algoritmo Eq2grau;
        VAR
          a,b,c: inteiro;
          delta,x1,x2: real;
        INICIO
1       LEIA(a,b,c);
2        delta ← sqr(b) – 4*a*c;
3        x1 ← -b + sqrt(delta)/(2*a);
4        x2 ← -b - sqrt(delta)/(2*a);
5        ESCREVA('As raízes são:',x1,x2);
        FIM
```

Quadro 5.1: Teste de mesa para algoritmo do cálculo de uma equação do 2º grau

Linha	a	b	c	delta	x1	x2	Comentário
1	[1]	[-4]	[3]	?	?	?	Leitura das variáveis a,b,c
2	1	-4	3	4	?	?	Cálculo de delta
3	1	-4	3	4	3	?	Cálculo de x1
4	1	-4	3	4	3	1	Cálculo de x2
5	1	-4	3	4	{3}	{1}	Saída com as raízes x1 e x2

Outro exemplo:

Faça um algoritmo que armazene dois números em duas variáveis e que troque os valores destas variáveis.

Solução:

A abordagem mais direta para a troca dos valores de uma variável seria a atribuição cruzada das variáveis. Vejamos o resultado desta abordagem:

```
        Algoritmo troca_valores_v1;
        VAR
          a,b: inteiro;
        INICIO
1         LEIA (a,b);
2         a ← b;
3         b ← a;
4         ESCREVA(a,b);
        FIM
```

Apesar de parecer simples e correto à primeira vista, essa implementação tem um erro que pode ser identificado pelo teste de mesa. Veja a tabela abaixo:

Quadro 5.2: Teste de mesa para o algoritmo da troca de valores

Linha	a	b	Comentário
1	[1]	[2]	Leitura das variáveis a e b
2	2	2	Atribuição do valor da variável b para a variável a
3	2	2	Atribuição do valor da variável a para a variável b
4	{2}	{2}	Saída com os valores das variáveis a e b.

Perceba que ao contrário do esperado, não houve troca de valores. Em vez disso, o valor 2 (da variável b) foi colocado nas duas variáveis e o valor 1 (da variável a) foi perdido. Isto ocorre porque cada variável pode manter apenas um valor a cada momento, e a atribuição do valor de b em a (linha 2) fez com que o valor inicial da variável a (1) fosse perdido para sempre.

Bem, como dois corpos não podem ocupar o mesmo espaço no mesmo momento, precisamos ter um local intermediário para efetivar a troca. Da mesma forma que quando duas pessoas vão trocar de lugar, uma precisa ir para um terceiro lugar, para que a primeira assuma sua posição e, depois disso, a primeira pessoa passa a assumir o local agora vago, temos que ter um espaço intermediário. Esse espaço intermediário será criado por meio de uma variável auxiliar.

```
        Algoritmo troca_valores_v2;
        VAR
          a,b,aux: inteiro;
        INICIO
1         LEIA (a,b);
2         aux ←a;
3         a ← b;
4         b ← aux;
5         ESCREVA(a,b);
        FIM
```

Quadro 5.3: Teste de mesa para o algoritmo da troca de valores com variável auxiliar

Linha	a	b	aux	Comentário
1	[1]	[2]	?	Leitura das variáveis a e b
2	1	2	1	Atribuição do valor da variável a para a variável aux
3	2	2	1	Atribuição do valor da variável b para a variável a
4	2	1	1	Atribuição do valor da variável aux para a variável b
5	{2}	{1}	1	Saída com os valores das variáveis a e b.

Perceba que a criação da variável aux permitiu que o valor da variável a fosse salvo antes que a variável b fosse atribuída a variável a. Desta forma, esse valor guardado em aux pode ser utilizado para consumar a troca das variáveis.

Resumo

Nesta aula foram apresentados e discutidos a importância das instruções primitivas na construção de um algoritmo. Vimos que a utilização de blocos de instruções representam o início e o término do conjunto de instruções que serão executadas em sequência a fim de atingir um determinado objetivo. Aprendemos a importância da utilização dos comandos de entrada e saída de dados em um algoritmo de forma a torná-lo mais adaptável. Os conceitos de entrada, processamento e saída são fundamentais para se entender um problema e devem ser aplicados na resolução de algoritmos. Os testes de mesa apresentam uma boa alternativa para correção e otimização da lógica aplicada nos algoritmos.

Atividades de aprendizagem

Para um melhor aprendizado, procure transcrever os algoritmos em pseudo-código utilizando o *software* Visualg. Desta forma, você poderá testar se sua lógica está correta.

1. Desenvolva algoritmos em forma de diagramas de bloco e português estruturado para os seguintes problemas:

a) Ler uma temperatura em graus Celsius e apresentá-la convertida em graus Fahrenheit. A fórmula de conversão é: F = (9*C+160)/5, sendo F a temperatura em Fahrenheit e C a temperatura em Celsius.

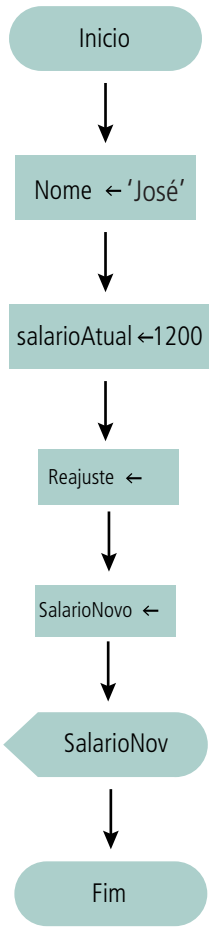
b) Efetuar o cálculo da quantidade de litros de combustível gastos em uma viagem, utilizando-se um automóvel que faz 12Km por litro. Para obter o cálculo, o usuário deverá fornecer o tempo gasto e a velocidade média durante a viagem. Desta forma, será possível obter a distância percorrida com a fórmula: DISTÂNCIA = TEMPO*VELOCIDADE. Tendo o valor da distância, basta calcular a quantidade de litros de combustível utilizada na viagem com a fórmula: LITROS_USADOS = DISTANCIA/12. O algoritmo deverá apresentar os valores da velocidade média, tempo gasto na viagem, distância percorrida e a quantidade de litros utilizada na viagem.

c) Efetuar o cálculo e a apresentação do valor de uma prestação em atraso, utilizando a fórmula: PRESTAÇÃO ← VALOR + (VALOR*(TAXA/100)*TEMPO), sendo que a TAXA representa o percentual de juros aplicados e TEMPO representa a quantidade de dias em atraso.

2. Faça um teste de mesa para o algoritmo abaixo e descreva a finalidade deste algoritmo.

```
Algoritmo numero;
VAR
valor_inicial,valor_temp,milhar,dezena,centena,unidade,valor_
invertido:inteiro;
INICIO
  ESCREVA('Entre com um numero de 4 dígitos');
  LEIA(valor_inicial);
  valor_temp ← valor_inicial;
  milhar ← valor_temp div 1000;
  valor_temp := valor_temp mod 1000;
  centena ← valor_temp div 100;
  valor_temp ← valor_temp mod 100;
  dezena ← valor_temp div 10;
  valor_temp ← valor_temp mod 10;
  unidade ← valor_temp;
  valor_invertido ← unidade * 1000;
  valor_invertido ← valor_invertido + (dezena * 100);
  valor_invertido ← valor_invertido + (centena * 10);
  valor_invertido ← valor_invertido + milhar;
  ESCREVA('O valor final é:',valor_invertido);
FIM
```

3. Sabendo-se que José tem direito a 15% de reajuste de salário, complete o diagrama abaixo:



Aula 6 - Estruturas de controle: a tomada de decisões

Objetivos

Conhecer as características das estruturas de decisão simples e aplicá-las em exemplos práticos.

Conhecer as características das estruturas de decisão compostas e aplicá-las em exemplos práticos.

Aplicar a combinação destas estruturas em exemplos práticos.

Até agora, todos os algoritmos que foram vistos, têm um fluxo de execução único, ou seja, as instruções que serão executadas serão sempre as mesmas, independentemente dos valores de entrada e produzirão resultados diferentes dependendo desses valores. Entretanto, na vida real, muitas vezes executamos determinados procedimentos dependendo de uma série de situações ou condições. Por exemplo, quando vamos ao mercado, escolhemos o que compraremos com base em alguns fatores, como tempo, dinheiro, meio de transporte, entre outros. Caso tenhamos mais dinheiro, podemos comprar produtos mais caros, caso contrário, escolhemos marcas mais baratas ou abdicamos de comprar alguns produtos. Nesse exemplo, o dinheiro determina a ação e a forma de compra, que será diferente dependendo dessa condição.

Na construção de algoritmos, também temos as mesmas necessidades de determinar ações diferentes dependendo da avaliação de certas condições. O uso de condições ou comandos de decisão muda o fluxo das instruções de um algoritmo, permitindo que diferentes instruções sejam executadas de acordo com a entrada do programa. Imagine um exemplo: Um programa que apresente a média escolar de um aluno. O programa precisa, além de calcular a média de um aluno, apresentar se ele está aprovado ou reprovado segundo a análise de sua média. Observe que será necessário verificar a média do aluno para então tomar uma decisão no sentido de apresentar a sua real situação: aprovado ou reprovado.

As estruturas de decisão podem ser classificadas de quatro formas diferentes, sendo essa classificação baseada na organização lógica existente em cada situação. Essa classificação diferencia uma estrutura de decisão como: decisão simples, decisão composta, decisão encadeada e decisão de múltipla escolha.

6.1 Estrutura de decisão simples (comando: se...fim_se)

Na estrutura de decisão simples, uma instrução ou um conjunto de instruções é executado somente se o teste condicional especificado retornar o valor verdadeiro. Caso o resultado do teste seja falso, nenhuma das instruções delimitadas pela estrutura de seleção será executada, e a execução das instruções será desviada para a instrução imediatamente seguinte à estrutura de seleção.

A seguir temos a representação desta estrutura através de pseudocódigo e diagramas de bloco:

```
se (<condição>) então
  < instruções para a condição verdadeira >
fim_se
  < instruções para a condição falsa ou após ser verdadeira >
```

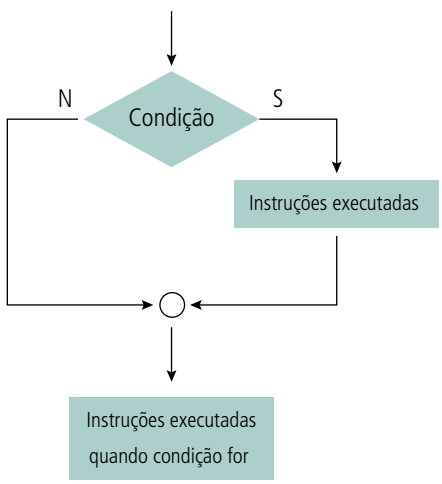


Figura 6.1: Representação em diagrama de blocos para a estrutura de decisão simples
Fonte: Elaborada pelo autor

A seguir é mostrado um exemplo de um algoritmo - na forma de fluxograma e pseudocódigo - que lê dois valores numéricos, depois efetua sua adição e apresenta o seu resultado caso o valor somado seja maior que 10.

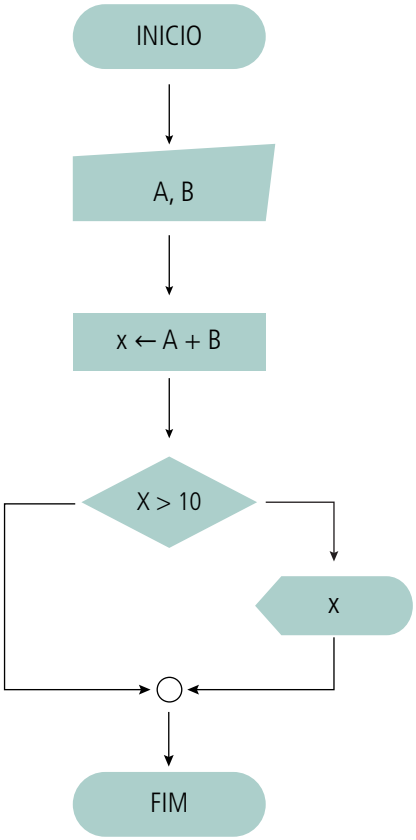


Figura 6.2: Exemplo da utilização da estrutura se...então...fim_se através de diagrama de blocos.
Fonte: Elaborada pelo autor

Pseudocódigo:

```
        Algoritmo soma_numeros
        Var
        X : inteiro
        A : inteiro
        B : inteiro
        Inicio
1        Leia(A)
2        Leia(B)
3        X ← A + B
4        se (X > 10) entao
5       Escreva(X)
         fim_se
6   Escreva('Fim do Algoritmo')
        Fim
```

Reecrevendo o algoritmo no **Visualg**:

```
Algoritmo "soma_numeros"
var
  X : inteiro
  A : inteiro
  B : inteiro
inicio
  Leia(A)
  Leia(B)
  X <- A + B
  se (X > 10) entao
   Escreva(X)
  Fimse
  Escreva("Fim do Algoritmo")
fimalgoritmo
```

Observe que após a definição dos tipos de variáveis, é solicitada a leitura dos valores para as variáveis A e B, depois esses valores são implicados na variável X, a qual possui o resultado da adição dos dois valores. Neste ponto, é questionado no programa uma condição que permitirá imprimir o resultado da soma caso esta seja maior que 10, e não sendo, o programa apenas imprime a mensagem "fim do algoritmo" e é encerrado sem apresentar a referida soma, uma vez que a condição é falsa.

A seguir temos o Teste de Mesa para A = 5, B = 4. Observe que a linha 5 contendo a instrução: "Escreva(X)" não é executada, pois a soma das variáveis A e B atribuída a X não satisfaz a condição da instrução contida na linha 4, fazendo com que a instrução contida na linha 5 não seja executada.

Instrução	Linha	A	B	X
1	1	[5]	?	?
2	2	5	[4]	?
3	3	5	4	9
4	4	5	4	9
5	6	5	4	9

A seguir temos o Teste de Mesa para A = 7, B = 8. Observe agora que todas as linhas do algoritmo são executadas, pois a soma das variáveis A e B atribuída a X satisfaz a condição da instrução contida na linha 4 fazendo com que a instrução contida na linha 5 seja executada.

Instrução	Linha	A	B	X
1	1	[5]	?	?
2	2	5	[4]	?
3	3	5	4	9
4	4	5	4	9
5	5	5	4	{9}
6	6	5	4	9

6.2 Estrutura de decisão composta (comando: se...senão...fim_se)

Na Estrutura de decisão composta uma instrução ou um conjunto de instruções é executado se o teste condicional especificado retornar o valor verdadeiro e, caso o resultado do teste seja falso, outra instrução ou um conjunto de instruções é executado. Assim a seleção composta permite desviar o fluxo para dois caminhos distintos. A seguir temos a representação desta estrutura através de pseudocódigo e diagramas de bloco:

```
se (<condição>) então
  < instruções para a condição verdadeira >
senão
  < instruções para a condição falsa >
fim_se
< continuação das instruções do algoritmo >
```

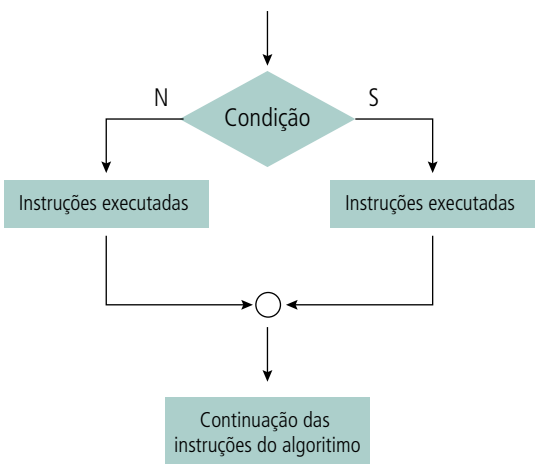


Figura 6.3: Representação em diagrama de blocos para a estrutura de decisão composta
Fonte: Elaborada pelo autor

Para um exemplo da utilização desta estrutura considere o seguinte problema: Ler um número e determinar se o mesmo é par ou ímpar.

Pseudocódigo:

```
        Algoritmo par_impar
        Var
        X : inteiro
        Inicio
1       Leia(X)
2       se (X mod 2) = 0 então
3     Escreva(X, 'é par')
        senão
4         Escreva(X, 'é ímpar')
        fim_se
5  Escreva('Fim do Algoritmo')
   Fim
```

Reecrevendo o algoritmo no **Visualg**:

```
Algoritmo "par_impar"
var
X : inteiro
inicio
 Escreva("entre com um número:")
 Leia(X)
 se (X mod 2)= 0 entao
  Escreval(X, "é par")
 Senao
  Escreval(X, "é impar")
 fimse
 Escreval("Fim do Algoritmo")
fimalgoritmo
```

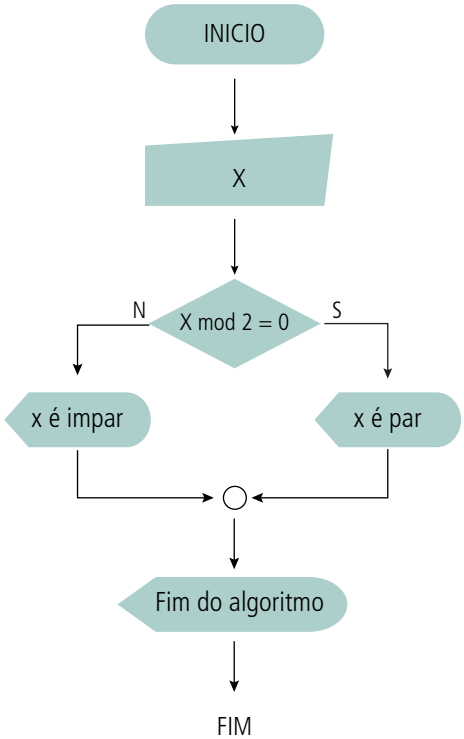


Figura 6.4: Representação do diagrama de blocos para o algoritmo par_impar
Fonte: Elaborada pelo autor

Observe que após a definição do tipo da variável, é solicitada a leitura do valor para a variável X, depois é verificado o resto da divisão inteira do valor de X por 2, ou seja, verificamos se X é divisível ou não por 2. Neste ponto, dependendo do resultado da avaliação da expressão "X mod 2=0", é mostrado o resultado do algoritmo "X é par " para o caso verdadeiro ou "X é ímpar" para o caso falso.

A seguir temos o Teste de Mesa para X = 5. Observe que a linha 3 contendo a instrução: "X é par" não é executada, pois o teste lógico da expressão: "X mod 2 = 0" é falso. Sendo assim, a instrução da linha 4 é executada em seguida da instrução da linha 5, onde o algoritmo é encerrado.

Instrução	Linha	X
1	1	[5]
2	2	5
3	4	{5}
4	5	{Fim do algoritmo}

6.3 Estruturas de decisão encadeadas

Uma instrução de seleção pode ser inserida dentro de outra, formando uma estrutura de seleção chamada de **estrutura de seleção encadeada**. O encadeamento dessas instruções também é conhecido como **aninhamento de instruções de seleção**, não existindo limite de quantas estruturas de seleção podem estar dentro de outra. Vai depender da complexidade do algoritmo. O encadeamento de instruções permite tornar uma solução algorítmica ainda mais eficiente, uma vez que diminui testes condicionais desnecessários.

A seguir temos a representação desta estrutura através de pseudocódigo e diagramas de bloco:

```
se (<condição>) então
  < instruções para a condição verdadeira >
Senão
        Se (<condição>) então
          < instruções para a condição verdadeira >
        Senão
          Se (<condição>) então
                < instruções para a condição verdadeira >

...
Fim_se
< continuação das instruções do algoritmo >
```

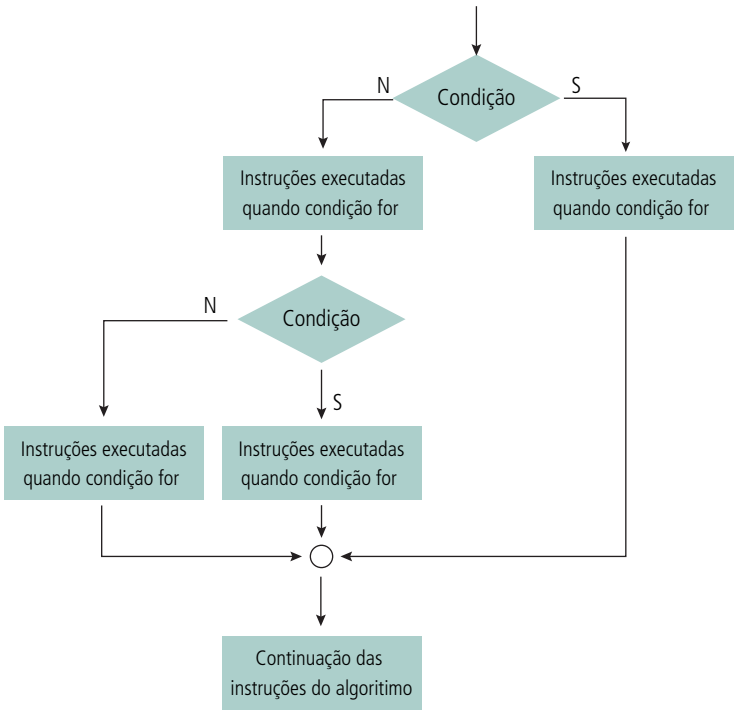


Figura 6.5: Representação em diagrama de blocos para a estrutura de decisão encadeada
Fonte: Elaborada pelo autor

A seguir mostraremos um exemplo de um algoritmo que utiliza a estrutura de seleção encadeada. Problema: Elaborar um algoritmo que efetue o cálculo do reajuste de salário de um funcionário. Considere que o funcionário deverá receber um reajuste de 15% caso seu salário seja menor que R$ 500,00. Se o salário for maior ou igual a R$ 500 e até 1000, seu reajuste será de 10%. Caso o funcionário ganhe acima de R$ 1000,00 o reajuste deverá ser de 5%.

Resolvendo,
Perceba que o problema em questão estabelece três condições para calcular o reajuste do salário de um funcionário, sendo:

- Salário < 500 : reajuste de 15%
- Salário >= 500, mas <= 1000 : reajuste de 10%
- Salário acima de 1000 : reajuste de 5%.

Diagrama de blocos

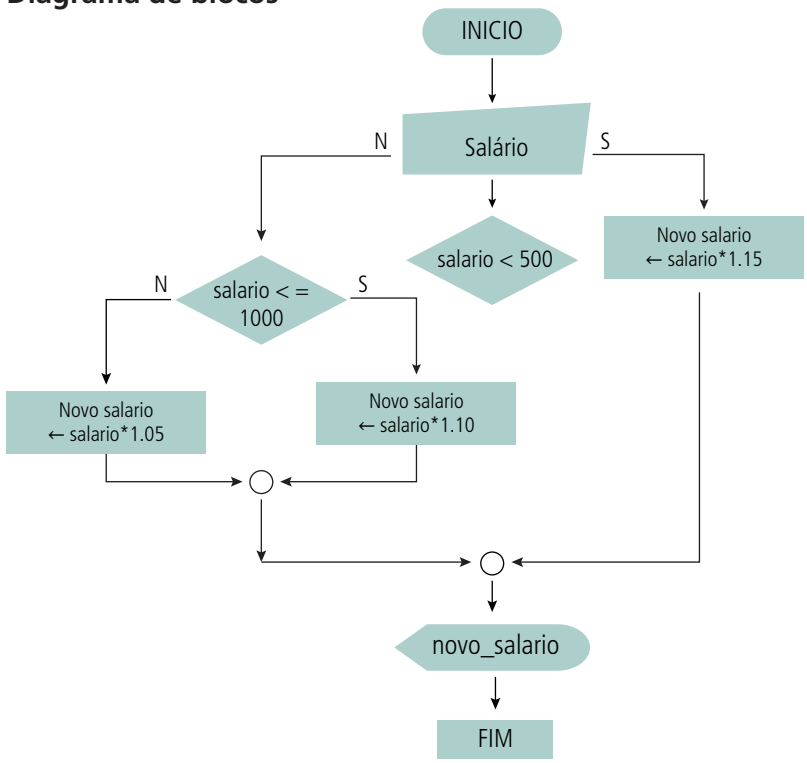


Figura 6.6: exemplo da utilização de uma estrutura condicional encadeada através de diagrama de blocos
Fonte: Elaborada pelo autor

Pseudocódigo

```
Algoritmo Reajusta_salario;
Var
  novo_salario : real;
  salario : real;
Inicio
  Leia(salario);
  Se (salario < 500) então
   novo_salario ← salario * 1.15
  Senão
   Se (salario <= 1000) então
    novo_salario ← salario * 1.10
   Senão
    novo_salario ← salario * 1.05
   Fim_se
  Fim_se
Fim
```

Reecrevendo o algoritmo no **Visualg**:

```
algoritmo "Reajusta_salario"
var
  novo_salario : real
  salario : real
inicio
  Escreval("Entre com o salário:")
  Leia(salario)
  Se (salario < 500) entao
   novo_salario <- salario * 1.15
  Senao
   Se (salario <= 1000) entao
    novo_salario <- salario * 1.10
   Senao
    novo_salario <- salario * 1.05
   Fimse
  Fimse
  Escreval("O novo salario é:", novo_salario)
fimalgoritmo
```

6.4 Estruturas de decisão de múltipla escolha (comando: caso)

Outra estrutura de decisão, muito utilizada nas avaliações de valores para uma variável ou valor individual, é a **decisão de múltipla escolha**. Diferente das estruturas discutidas anteriormente, nesta o teste condicional não retorna um valor lógico, mas sim um valor inteiro, real ou caractere. Outra diferença está no fato de que essa estrutura permite desviar o fluxo de execução para vários caminhos. Em outras palavras, para a expressão definida no teste condicional, é verificada a sua igualdade com as opções definidas resultando na execução de um bloco, ou de uma única instrução específica, para cada opção.

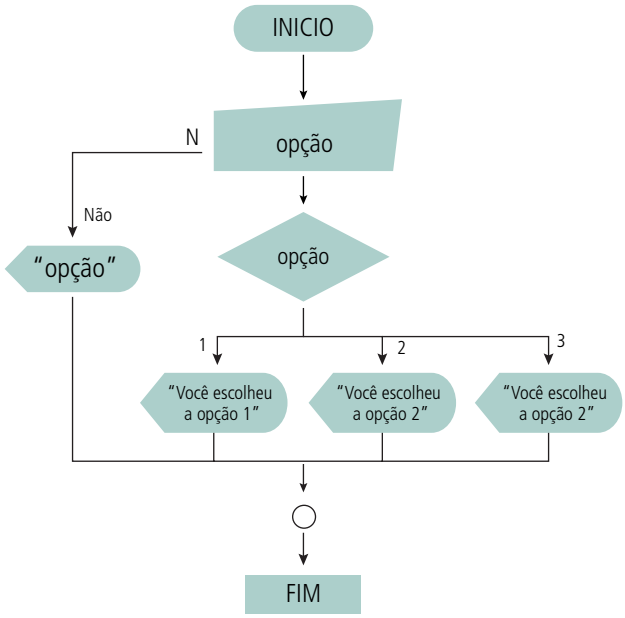


Figura 6.7: Representação em diagrama de blocos para a estrutura de decisão de múltipla escolha
Fonte: Elaborada pelo autor

Representando a estrutura através de pseudocódigo, temos:

CASO <valor (inteiro/caractere)> seja:
<primeiro valor (inteiro/caractere)> : <bloco de instruções>

<segundo valor (inteiro/caractere) > : <bloco de instruções>

<terceiro valor (inteiro/caractere) > : <bloco de instruções>

...

<caso contrário> :<bloco de instruções>

FIM_CASO

A seguir mostraremos um exemplo de um algoritmo que utiliza a estrutura de decisão de múltipla escolha. Problema: Elaborar um algoritmo que leia um número de um mês qualquer e escreva o mês correspondente por extenso. Resolvendo,

Sabemos que os meses do ano são 12 (doze) e que cada mês é representado por um número. Assim sendo, o primeiro mês do ano (janeiro) é representado pelo número 1 (um), o segundo mês (fevereiro) é representado pelo número 2 (dois) e assim por diante até chegarmos ao último mês do ano (dezembro) que é representado pelo numero 12 (doze).

Pseudocódigo:

```
Algoritmo escreve_mes
Var
 NumerodoMes : inteiro;
Inicio
 Leia (numerodomes);
 Caso (numerodomes) seja :
  1 : escreva("Janeiro");
  2 : escreva("Fevereiro");
  3 : escreva("Março");
  4 : escreva("Abril");
  5 : escreva("Maio");
  6 : escreva("Junho");
  7 : escreva("Julho");
  8 : escreva("Agosto");
  9 : escreva("Setembro");
  10 : escreva("Outubro");
  11 : escreva("Novembro");
  12 : escreva("Dezembro");
  Caso Contrário:
   escreva("Mês inválido");
 Fim_Caso;
Fim.
```

Reecrevendo o algoritmo no **Visualg**:

```
algoritmo "escreve_mes"
var
 numerodomes : inteiro
inicio
 Leia (numerodomes)
 Escolha (numerodomes)
  caso 1
    escreval("Janeiro")
  caso 2
    escreval("Fevereiro")
  caso 3
    escreval("Março")
  caso 4
    escreval("Abril")
  caso 5
    escreval ("Maio")
  caso 6
    escreval("Junho")
  caso 7
    escreval("Julho")
  caso 8
    escreval("Agosto")
  caso 9
    escreval("Setembro")
  caso 10
        escreval("Outubro")
  caso 11
    escreval("Novembro")
  caso 12
    escreval("Dezembro")
  outrocaso
    escreval("Mês inválido")
 fimescolha
 Escreval(nomedomes)
fimalgoritmo
```

Resumo

Nesta aula foram discutidas as principais estruturas de decisão usadas na construção de algoritmos. As instruções em um algoritmo são dispostas em uma estrutura sequencial, no entanto, existem casos em que o fluxo de instruções não deve ser executado sequencialmente, ou seja, é necessário que este fluxo seja desviado mediante determinadas condições ou circunstâncias do próprio algoritmo. É neste caso que entram as estruturas de decisão que são responsáveis por desviar o fluxo de instruções em um algoritmo a partir do resultado de um teste condicional. Partindo do caso mais simples para o mais complexo, foram apresentadas as estruturas condicionais simples, as estruturas condicionais compostas, as estruturas condicionais encadeadas e as estruturas condicionais de múltipla escolha. Para cada caso particular de um algoritmo pode-se utilizar uma estrutura de decisão mais adequada, ou seja, aquela que deixa o algoritmo mais "legível".

Atividades de aprendizagem

1. Faça um algoritmo que leia 3 notas de um aluno, calcule a média aritmética e escreva uma mensagem de acordo com a seguinte tabela:

Média	Mensagem
Maior ou igual a 0 e menor que 4	Reprovado
Maior ou igual a 4 e menor que 7	Exame Final
Maior ou igual a 7 e menor ou igual a 10	Aprovado

RESPOSTA:

Utilizando estrutura condicional simples:

```
ALGORITMO MEDIA_ALUNO
Var
  nota1,nota2,nota3,media:real;
Inicio
  Escreva("digite a 1ª nota do aluno:");
  Leia(nota1);
  Escreva("digite a 2ª nota do aluno:");
  Leia(nota2);
  Escreva("digite a 3ª nota do aluno:");
  Leia(nota3);
  media ← (nota1+nota2+nota3)/3;
  escreva ("A média obtida foi",media);
  se (media >=0 E media < 4) entao
```

```
   escreva("Reprovado")
  se (media >=4 E media < 7) entao
    escreva("Exame Final")
  se (media >=7 E media <=10) entao
    escreva("Aprovado")
Fim.
```

Utilizando estrutura condicional composta:

```
ALGORITMO MEDIA_ALUNO
Var
  nota1,nota2,nota3,media:real;
Inicio
  Escreva("digite a 1ª nota do aluno:");
  Leia(nota1);
  Escreva("digite a 2ª nota do aluno:");
  Leia(nota2);
  Escreva("digite a 3ª nota do aluno:");
  Leia(nota3);
  media ß (nota1+nota2+nota3)/3;
  escreva ("A média obtida foi",media);
  se (media >=0 E media < 4) então
   escreva("Reprovado")
  senão
   se(media >=4 E media < 7) então
    escreva("Exame final")
   senão
    se(media >=7 E media <= 10) então
      escreva("Aprovado")
   fimse
  fimse
Fim.
```

2. Faça um algoritmo que simule uma calculadora com as quatro operações básicas: (soma(+), subtração(-), multiplicação(*) e divisão(/)). O algoritmo deve solicitar ao usuário a entrada de dois operandos e da operação a ser executada. Dependendo da opção escolhida, deve ser executada a operação solicitada e escrito seu resultado. (Dica: Utilize uma variável caractere para armazenar a operação e utilize o comando de decisão de múltipla escolha para escolher a operação a ser executada).

RESPOSTA:

```
ALGORITMO CALCULADORA;
Var
  operando1,operando2: inteiro;
  operacao: caractere;
  resultado: real;
Inicio
  Escreva("Entre com dois números");
  Escreva("Entre com o operando 1:")
  Leia(operando1);
  Escreva("Entre com o operando 2:")
  Leia(operando2);
  Escreva("Entre com a operação desejada:(+,-,*,/)");
  Leia(operacao);
  CASO operacao seja:
   "+" : resultado ← operando1+operando2;
   "-" : resultado ← operando1-operando2;
   "*" : resultado ← operando1*operando2;
   "/" : resultado ← operando1/operando2;
   Caso Contrario : Escreva ("operação inválida");
  FIMCASO
  Escreva(resultado);
Fim.
```

3. Faça um algoritmo que leia três lados de um triangulo, e se os lados formarem um triângulo (condição: cada lado tem que ser menor que a soma dos outros dois), escreva o tipo do triangulo formado: isósceles (dois lados iguais), eqüilátero (três lados iguais) ou escaleno (três lados diferentes).

```
Algoritmo Triangulos
var
  a, b, c: real
Inicio
escreva("Digite o tamanho dos lados seguidos")
Leia(a,b,c)
se (a<(b+c)) e (b<(a+c)) e (c<(a+b)) entao
  se a < > b entao
   se b < > c entao
     escreva("Este é um triângulo escaleno.")
   senao
     escreva("Este é um triângulo isósceles.")
   fimse
  senao
   se a < > c entao
     escreva("Este é um triângulo isósceles.")
   senao
     escreva("Este é um triângulo equilátero.")
   fimse
  fimse
senao
 escreva("Esses lados não configuram um triângulo.")
fimse
Fim
```

Aula 7 - Estruturas de repetição

Objetivos

Conhecer as características das estruturas de repetição e em que situações é mais apropriado usá-las.

Entender passo-a-passo a sintaxe e como funciona cada estrutura de repetição (Enquanto, Repita e Para) e estudar como um problema pode ser resolvido utilizando-se mais de um algoritmo com estruturas de repetição diferentes.

Construir algoritmos utilizando estruturas de repetição combinadas.

7.1 Introdução

No mundo real, é comum a repetição de procedimentos para realizar tarefas do dia-a-dia. Esses procedimentos são repetidos várias vezes, mas se encerram quando o objetivo e atingido. Por exemplo, quando uma pessoa aperta um parafuso, ela gira a chave de fenda uma vez, duas vezes, etc. até que o parafuso esteja apertado o suficiente. Durante esse processo, é verificado a cada volta, se o parafuso está bem firme.

Da mesma forma, podemos estruturar várias atividades diárias como repetitivas. Durante a chamada feita por um professor, por exemplo, ele chama os nomes enquanto não terminar a lista. Outras repetições podem ser qualificadas com antecedência. O aluno de castigo precisa escrever 100 vezes no quadro negro: "Nunca mais vou fazer bagunça", executa a mesma instrução 100 vezes.

Todas as repetições têm característica comum: o fato de haver uma verificação de condição que pode ser representada por um valor lógico, para determinar se a repetição prossegue ou não. Essa é a base para a implementação dos comandos de repetição em algoritmos. Em vez de fazermos trabalho braçal, escrevendo a mesma instrução várias vezes, poderemos utilizar uma estrutura que indique que tal instrução será executada quantas vezes forem necessárias.

7.2 Estrutura de repetição para um número indefinido de repetições e teste no início: estrutura enquanto

Antes de vermos a sintaxe em nosso pseudocódigo, vejamos um exemplo do mundo real: o problema do elevador. Um elevador residencial tem um comportamento que pode ser descrito de forma algorítmica. Vejamos o seu funcionamento:

Na subida: sobe cada andar, verificando se está em um andar selecionado dentro do elevador. Isso é feito até chegar ao andar mais alto selecionado dentro ou fora do elevador. Enquanto não chegar ao andar mais alto selecionado interna ou externamente faça:

```
Inicio
  Suba um andar.
  Se o andar foi o selecionado internamente então
  Inicio
    Pare,
    Abra as portas,
    Feche as portas,
  Fim
Fim
```

Na descida: desce cada andar, verificando se está em um andar selecionado dentro ou fora do elevador. Isso é feito até chegar ao andar mais baixo selecionado dentro ou fora do elevador. Enquanto não chegar ao andar mais baixo selecionado interna ou externamente faça:

```
Inicio
  Desça um andar.
  Se o andar foi o selecionado internamente então
  Inicio
    Pare,
    Abra as portas,
    Feche as portas,
  Fim
Fim
```

O comando enquanto caracteriza-se por uma verificação de encerramento de atividades antes de se iniciar (ou reiniciar) a execução de seu bloco de instruções. Dessa forma, no algoritmo do elevador, antes de subir/descer um andar é verificado se o andar atual é o mais alto/baixo selecionado. Caso não seja, um conjunto de atividades é executado (sobe/desce um andar, verifica se é um andar selecionado e abre (ou não) as portas.

Sintaxe:

```
Enquanto < valor booleano > faça
   < bloco de instruções >
Fim_Enquanto
< continuação do algoritmo>
```

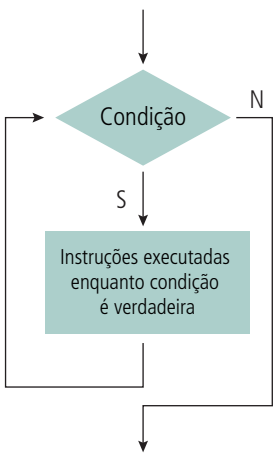


Figura 7.1: Diagrama de blocos da estrutura enquanto...faça...fim_enquanto
Fonte: Elaborada pelo autor

Voltemos ao exemplo do aluno de castigo. Fazer um algoritmo que escreves-se para ele, cem vezes, "Não vou fazer mais bagunça", antes deste capítulo, seria uma tarefa bastante cansativa. O algoritmo seria:

```
Algoritmo Lição_Aluno_versão1
Inicio
  Escreva("Nunca mais vou fazer bagunça!");
  Escreva("Nunca mais vou fazer bagunça!");
  Escreva("Nunca mais vou fazer bagunça!");
  Escreva("Nunca mais vou fazer bagunça!");
  ...
  {O comando precisa ser escrito 100 vezes...}
Fim
```

Para que possamos utilizar o comando de repetição, precisaremos verificar, de alguma forma, se o comando já foi executado 100 vezes.

```
=
enquanto <não foi executado 100 vezes o próximo bloco> faça
escreva("Nunca mais vou fazer bagunça ");
```

Bem, o problema reside em implementar essa verificação. Uma estratégia muito comum para esse tipo de situação é acompanhar a execução das repetições contando cada vez que o bloco é executado. Cada execução do bloco de instruções é chamada iteração. O próprio comando de repetição em conjunto com seu bloco de instruções é conhecido como loop ou laço.

Para que tenhamos a informação de quantas iterações já foram realizadas no laço, necessitaremos de uma variável que fará o papel de contador. Essa variável conterá o número de iterações já realizadas, sendo atualizada a cada nova iteração. Ex:

Faça um algoritmo que escreva 100 vezes o texto: "Nunca mais vou fazer bagunça", utilizando uma estrutura de repetição.

```
Algoritmo Licao_Aluno_Versao2
var
  contador: inteiro;
Inicio
  contador ← 0;
  Enquanto (contador < 100) faça
  Início
   escreva("Não vou fazer mais bagunça!");
   contador ← contador + 1;    { A cada iteração, conta-se mais 1 }
  Fim
Fim
```

Outro exemplo:
Faça um algoritmo que leia 100 números e retorne a média desses valores.

```
Algoritmo Soma_Media_100
var
contador: inteiro; valor, soma, media: real;
 Inicio
                                   {Nenhuma iteração foi feita até aqui}
                                   {Ainda não foi somado nenhum valor}
                                   {O bloco será repetido 100 vezes}
1  contador ← 0;
2  soma ← 0;
3  Enquanto (contador < 100) faça
   Inicio
4    escreva("Entre com um valor: ");
5    leia(valor) ;
6    soma ← soma + valor;
7    contador ← contador + 1; { A cada iteração, conta-se mais 1 }
   Fim
8  media ← soma/contador;
9  escreva("Soma: ", soma);
10  escreva("Média: ", media) ;
 Fim.
```

É interessante verificar o processo de acumulação de valores feito na variável soma (linha 6). Seu valor é atualizado a cada iteração, somando-se seu valor atual com o valor lido. Para que isso funcione, é importante que o valor inicial da variável seja definido antes da entrada do laço, para que um valor desconhecido não seja atribuído primeira iteração do laço.

Vejamos o teste de mesa para a melhor compreensão do processo. Para viabilizar a realização do teste de mesa, consideremos o laço até 3 e não até 100, como está no Algoritmo

Entrada: 5,4,9 (ou seja, os valores que serão entrados pelo usuário serão 5,4,9 nesta sequência)

Quadro 7.1: Teste de Mesa para o algoritmo SomaMédia100					
Instrução	Linha	Contador	Valor	Soma	Média
1	1	0	?	?	?
2	2	0	?	0	?
3	3	0	?	0	?
4	4	0	?	0	?
5	5	0	[5]	0	?
6	6	0	5	5	?
7	7	1	5	5	?
8	3	1	5	5	?
9	4	1	5	5	?
10	5	1	[4]	5	?
11	6	1	4	9	?
12	7	2	4	9	?
13	3	2	4	9	?
14	4	2	4	9	?
15	5	2	[9]	9	?
16	6	2	9	18	?
17	7	3	9	18	?
18	3	3	9	18	?
19	8	3	9	18	6
20	9	3	9	{18}	6
21	10	3	9	18	{6}

Note que a condição de entrada/saída do laço está na linha 3. Esse teste é executado instruções 3, 8, 13 e 18. Perceba também que o teste é repetido sempre após o bloco de instruções (linha 7) pertencente ao laço. Quando a condição é verdadeira, a ser executada é a do início do bloco, na linha 4 (instruções 4, 9 e 14). Porém, na instrução 18, a condição é falsa (no nosso teste, estamos considerando **enquanto** contador < 3) e a próxima instrução a ser executada (instrução 19) está na linha 8, após instruções pertencentes ao comando **enquanto.**

7.3 Estrutura de repetição para um número indefinido de repetições e teste no final: estrutura repita

Além do comando **enquanto**, existem outras estruturas para implementar laços repetitivos. O comando **repita** funciona de forma similar ao comando **enquanto**, exceto pelo fato de que a condição de controle só é testada após a execução do bloco de comandos e não antes, como é o caso do comando **enquanto**. Outra diferença é que os comandos serão repetidos pelo menos uma vez, já que a condição de parada se encontra no final. Sintaxe:

```
Repita
   < bloco de instruções >
Até < valor booleano >;
< continuação do algoritmo >
```

Assim, podemos utilizar o comando **repita** sempre que tivermos certeza de que o bloco de instruções será executado ao menos uma vez, sem a necessidade do teste na entrada do bloco. Vejamos o mesmo exemplo utilizado na seção anterior. Faça um algoritmo que leia 100 números e retorne a média desses valores.

```
Algoritmo Soma_Media_100_versao2;
var
 contador: inteiro; valor, soma, media: real;
 Inicio
1  contador ← 0;
2  soma ← 0;
3  Repita // Enquanto (contador < 100) faça
4    escreva("Entre com um valor: ");
5    leia(valor) ;
6    soma ← soma + valor;
7    contador ← contador + 1; { A cada iteração, conta-se mais 1 }
   Ate (contador >= 100)
8  media ← soma/contador;
9  escreva("Soma: ", soma);
10  escreva("Média: ", media) ;
 Fim.
```

Pelo exemplo, podemos perceber que a condição do laço **até** (contador >= 100) é diferente da similar no comando **enquanto** implementada na seção anterior. Isso ocorre porque a saída de um laço **repita** ocorre quando a condição booleana se torna verdadeira. Nos laços **enquanto**, a saída do laço só ocorre quando a condição se torna falsa. Outro Exemplo:

Faça um algoritmo que calcule a soma dos números ímpares menores que 100.

Um número é impar quando sua divisão por 2 não é exata, ou seja, o resto resultante da divisão inteira do número por 2 tem valor 1. Lembrando que em pseudocódigo, o resto é representado pelo operador: **mod**.

Como o algoritmo solicita a soma dos valores ímpares dentro de uma faixa de valores específica (0 a 100), teremos que fazer o acúmulo do resultado apenas quando a condição ímpar for atendida. Essa condição será testada por todos os números dentro da faixa, por meio de um laço.

```
Algoritmo Soma_Impares;
Var
 num, soma : inteiro;
Início
 soma ← 0;
 num ← 1;
 Repita
   Se (num mod 2 = 1) então
    soma ← soma + num;
   Fim_se
   num ← num + 1;
 Até (num > 100)
 Escreva("A soma é:",soma);
Fim.
```

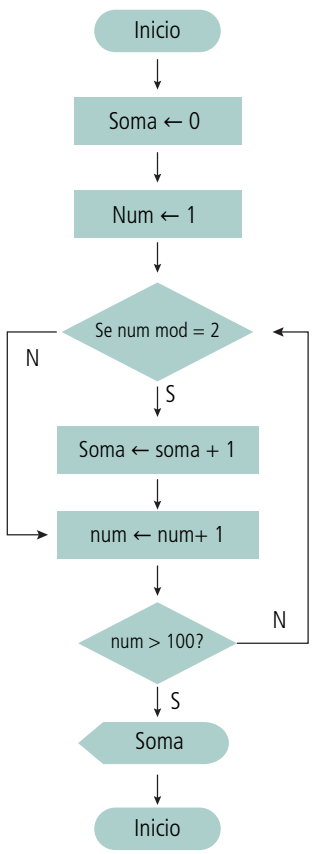


Figura 7.2: Diagrama de blocos do algoritmo soma_impares
Fonte: Elaborada pelo autor

7.4 Estrutura de repetição para um número definido de repetições: estrutura para

Se analisarmos os exemplos de utilização de laços, percebemos que a maioria deles tem comportamento similar. Uma situação inicial, definida antes do início do laço como uma preparação para a sua entrada, um teste de controle para a entrada/saída do bloco e uma instrução dentro do laço que, em algum momento, fará com que a condição controle seja atingida e o laço se encerre no momento oportuno. O comando para procura resumir essas três características comuns à maioria das implementações de laços em uma só instrução, facilitando assim a construção típica de laços.

Sintaxe:

```
Para <variavel de controle> de <inicio> até <fim> [passo <incremento>]
        < instruções >
<Fim_Para>
```

Portanto, temos a separação do comando em três cláusulas:

1. Preparação: **<variavel de controle> de <inicio>**
2. Condição: **<variavel de controle> até <fim>**
3. Incremento: **passo <incremento>**

A informação de passo está entre colchetes porque é opcional. O passo indica como será a variação da variável de controle. Por exemplo, quando for indicado passo 2, a variável de controle será aumentada em 2 unidades a cada interação até atingir o valor final. Quando a informação do passo for suprimida, isso significa que o incremento ou o decremento da variável de controle será de 1 (uma) unidade.

Exemplos:

```
Para I de 1 até 10 faça
Escreva( I)
```

O comando escreva(I) será executado 10 vezes, ou seja, para i variando de 1 a 10. Assim os valores de i serão: 1,2,3,4,5,6,7,8,9 e 10.

```
Para J de 1 até 9 passo 2 faça
Escreva(J)
```

O comando escreva(J) será executado 5 vezes, ou seja, para J variando de 1 a 9, de 2 em 2. Assim os valores de i serão: 1,3,5,7,9.

```
Para J de 15 até 1 passo -2 faça
Escreva(J)
```

O comando escreva(J) será executado 8 vezes, ou seja, para J variando de 15 a 1, de 2 em 2. Assim os valores de i serão: 15,13,11,9,7,5,3,1.

Vale citar que, quando temos mais de uma instrução de preparação (primeira cláusula), estas são separadas por vírgula e executadas na ordem em que se encontram no comando. Quando temos mais de uma condição de controle (segunda cláusula), os testes também são realizados em ordem, e a condição toda é Verdadeira apenas se todas as condições forem, ou seja, é equivalente a uma condição lógica ligada pelo conectivo **E**. Não há como representar composição de condições **OU** no comando para. Nesse caso, deve-se obrigatoriamente fazer uso dos comandos **enquanto** ou **repita**. O comando **para** é executado da seguinte forma:

1. Execute a instrução de preparação na primeira interação do laço
2. Execute o teste de controle. Caso seja verdadeiro, passe para o item 3. Caso contrário passe para o item 6.
3. Execute o bloco de instruções
4. Execute o passo para o alcance da condição de controle
5. Passe para o item 2.
6. Saia do laço e prossiga o algoritmo.

O comando para é equivalente à seguinte estrutura:

```
<instrução de preparação>
Enquanto < teste de controle>
Inicio
  <instruções>
  <passo>
Fim.
```

Para exemplificar, vamos refazer o algoritmo da soma dos números impares menores que 100 (feito anteriormente com as estruturas **enquanto** e **repita**) agora com a estrutura **para**.

```
Algoritmo Soma_Impares_versao3;
Var
  num,soma: inteiro;
Inicio
  Soma ← 0;
  Para (num de 1 até 100 passo 2) faça
    Se (num mod 2 = 1) então
      soma ← soma+num;
    Fim_se
  Fim_para
  Escreva("A soma dos números impares é:"soma);
Fim.
```

Será executado o conjunto de instruções entre a instrução **para** e a instrução **fim_para**, sendo a variável num (variável de controle) inicializada com o valor 1 e incrementada de mais 2 por meio da instrução **passo** até o valor 100.

Reescrevendo o algoritmo o algoritmo para ler 100 números e retornar a média desses valores.

```
Algoritmo Soma_Media_100_versao3;
var
 contador: inteiro;
 valor, soma, media: real;
 Início
1  contador ← 0;
2  soma ← 0;
3  Para (contador de 1 a 100) faça
4     escreva("Entre com um valor: ");
5     leia(valor) ;
6     soma ← soma + valor;
    Fim_para
8  media ← soma/100;
9  escreva("Soma: ", soma);
10  escreva("Média: ", media) ;
 Fim.
```

Será executado o conjunto de instruções entre a instrução para e a instrução fim_para, sendo a variável contador (variável de controle) inicializada com o valor 1 e incrementada de mais 1 por meio da instrução passo até o valor 100.

Este tipo de estrutura de repetição poderá ser utilizado todas as vezes que houver a necessidade de repetir trechos finitos, em que se conhecem os valores inicial e final.

7.5 Estruturas de controle encadeadas

Na aula anterior, foi discutido o fato de ocorrer o encadeamento das estruturas de decisão. Este fato pode também ocorrer com as estruturas de repetição. E neste ponto poderá ocorrer o encadeamento de um tipo de estrutura de repetição com outro tipo de estrutura de repetição. A existência destas ocorrências vai depender do problema a ser solucionado.

Devemos aqui salientar um pequeno detalhe. Nunca procure decorar estas regras, pois isto é impossível. Você precisa conhecer os comandos de entrada, processamento e saída, as estruturas de decisão e de repetição. Desta forma, conhecendo bem, você saberá utilizá-las no momento que for conveniente, pois na resolução de um problema, ele "pedirá" a estrutura mais conveniente. E, você, conhecendo-as bem, saberá automaticamente o momento certo de utilizá-las. A seguir serão apresentados os tipos de aninhamento que poderão ser combinados e será mostrado um exemplo de aplicação de cada uma destas estruturas para o seguinte problema:

Problema: Apresentar os resultados da tabuada de multiplicação dos números de 1 a 10, a qual deve ser apresentada no seguinte formato:

```
1 x 1   = 1
1 x 2   = 2
1 x 3   = 3
1 x 4  = 4
1 x 5  = 5
...
1 x 10  = 10

2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
```

```
2 x 5 = 10
...
2 x 10 = 20

3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
...
3 x 10 = 30
...
10 x 1 = 3
10 x 2 = 6
10 x 3 = 9
10 x 4 = 12
10 x 5 = 15
...
10 x 10 = 100
```

A tabuada de um número é o produto deste número pela sequência de números de 1 a 10. Sendo assim, a tabuada do numero 2 é formada pelas multiplicações do numero 2 pelos números: 1,2,3,4,5,6,7,8,9 e 10. Desta forma, o algoritmo deverá repetir para os 10 números a mesma regra, ou seja, efetuar multiplicações sucessivas de cada número pela sequência de números de 1 a 10. A variável número irá representar os números da primeira coluna da multiplicação, ou seja, o números que serão calculados a tabuada e a variável sequência irá representar os números da segunda coluna da multiplicação, ou seja, a sequência que será multiplicada por cada número. Algoritmo:

1º Passo: Inicializar as variáveis **fatorial** e **contador** com 1;
2º Passo: Repetir a execução do 3º e 4º passos por 10 vezes
3º Passo: **escreva(numero,"x",sequencia,"=",numero*sequencia);***
4º Passo: Incrementar 1 à variável **sequencia**
5º Passo: Incrementar 1 à variável **numero** efetuando o controle até 10.
6º Passo: Voltar ao 2º Passo, caso a variável numero seja menor ou igual a 10.

7.5.1 Encadeamento de estrutura enquanto com enquanto

Exemplo de encadeamento de estrutura **enquanto** com estrutura **enquanto**. Segue abaixo a representação em pseudocódigo:

Pseudocódigo

```
Enquanto <condição> faça
 Enquanto <condição> faça
   < instruções>
 Fim_enquanto
Fim_enquanto
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **enquanto** com **enquanto**.

```
Algoritmo Tabuada_1;
Var
  numero   : inteiro;
  sequencia : inteiro;
Inicio
  numero ← 1;
  sequencia ← 1;
  Enquanto (numero <= 10) faça
   Enquanto (sequencia <=10)faça
    escreva(numero,"x",sequencia,"=",numero*sequencia);
    sequencia ← sequencia+1;
   Fim_Enquanto
   numero ← numero +1;
  Fim_Enquanto;
Fim
```

7.5.2 Encadeamento de estrutura enquanto com repita

Exemplo de encadeamento de estrutura **enquanto** com estrutura **repita**. Segue abaixo a representação em pseudocódigo: Pseudocódigo

```
Enquanto <condição> faça
 Repita
   < instruções>
 Até que (<condição>)
Fim_enquanto
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **enquanto** com **repita**.

```
Algoritmo Tabuada_2;
Var
  numero   : inteiro;
  sequencia : inteiro;
Inicio

  numero ← 1;

  Enquanto (numero <= 10) faça
   sequencia ← 1;
   Repita
    escreva(numero,"x",sequencia,"=",numero*sequencia);
    sequencia ← sequencia+1;
   Até que (sequencia > 10)
   numero ← numero +1;
  Fim_Enquanto;
Fim
```

7.5.3 Encadeamento de estrutura enquanto com para

Exemplo de encadeamento de estrutura **enquanto** com estrutura **para**. Segue abaixo a representação em pseudocódigo:

```
Pseudocódigo
Enquanto <condição> faça
 Para <var> de <inicio> até <fim> passo <incr> faca
   < instruções>
 Fim_para
Fim_enquanto
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **enquanto** com **para**.

```
Algoritmo Tabuada_3;
Var
  numero    : inteiro;
  sequencia : inteiro;
Inicio
  numero ← 1;
  Enquanto (numero <= 10) faça
   Para (sequencia de 1 até 10) faça
    escreva(numero,"x",sequencia,"=",numero*sequencia);
   Fim_para
   numero ← numero +1;
  Fim_Enquanto;
Fim
```

7.5.4 Encadeamento de estrutura repita com repita

Exemplo de encadeamento de estrutura **repita** com estrutura **repita**. Segue abaixo a representação em pseudocódigo:

Pseudocódigo:

```
 Repita
  Repita
    < instruções>
  Até que (<condição2>)
 Até que (<condição1>)
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **repita** com **repita**.

```
Algoritmo Tabuada_4

Var
  numero   : inteiro
  sequencia : inteiro
Inicio
  numero<-1
  Repita
   sequencia<-1
   Repita
    escreva(numero,"x",sequencia,"=",numero*sequencia)
    sequencia <- sequencia+1

   Ate que(sequencia > 10)
   numero <- numero+1
  Ate que (numero > 10)
Fim
```

7.5.5 Encadeamento de estrutura repita com enquanto

Exemplo de encadeamento de estrutura **repita** com estrutura **enquanto**. Segue abaixo a representação em pseudocódigo:

Pseudocódigo:

```
 Repita
  Enquanto (<condição1>)faça
    < instruções>
  Fim_Enquanto
 Até que (<condição2>)
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **repita** com **enquanto**.

```
Algoritmo Tabuada_5
Var
  numero   : inteiro
  sequencia : inteiro
Inicio
  numero ← 1
  Repita
   sequencia ← 1
   Enquanto (sequencia <= 10) faca
    escreva(numero,"x",sequencia,"=",numero*sequencia)
    sequencia ← sequencia+1
   Fim_Enquanto
   numero ← numero+1
  Ate que (numero > 10)
Fim
```

7.5.6 Encadeamento de estrutura repita com para

Exemplo de encadeamento de estrutura **repita** com estrutura **para**. Segue abaixo a representação em pseudocódigo:

Pseudocódigo:

```
 Repita
 Para <var> de <inicio> até <fim> passo <incr> faca
   < instruções>
   Fim_para
 Até que (<condição1>)
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **repita** com **para**.

```
Algoritmo Tabuada_6
Var
  numero   : inteiro
  sequencia : inteiro
Inicio
  numero ← 1
  Repita
   Para sequencia de 1 ate 10 faca
     escreva(numero,"x",sequencia,"=",numero*sequencia)
   Fim_Para
   numero ← numero+1
  Ate que (numero > 10)
Fim
```

7.5.7 Encadeamento de estrutura para com para

Exemplo de encadeamento de estrutura **para** com estrutura **para**. Segue abaixo a representação em pseudocódigo:

Pseudocódigo:

```
Para <var1> de <inicio> até <fim> [passo <incr>] faca
  Para <var2> de <inicio> até <fim> [passo <incr>] faca
   < instruções>
    Fim_para
  Fim_para
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **para** com **para**.

```
Algoritmo Tabuada_7
Var
  numero   : inteiro
  sequencia : inteiro
Inicio
  Para numero de 1 ate 10 faca
   Para sequencia de 1 ate 10 faca
     escreva(numero,"x",sequencia,"=",numero*sequencia)
   Fim_Para
  Fim_Para
Fim
```

7.5.8 Encadeamento da estrutura para com enquanto

Exemplo de encadeamento da estrutura **para** com estrutura **enquanto**. Segue abaixo a representação em pseudocódigo:

Pseudocódigo:

```
Para <var> de <inicio> até <fim> [passo <incr>] faca
  Enquanto (<condição>) faça
    <instruções>
  Fim_Enquanto
  Fim_para
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **para** com **enquanto.**

```
Algoritmo Tabuada_8
Var
  numero   : inteiro
  sequencia : inteiro
Inicio
  Para numero de 1 ate 10 faca
   sequencia ← 1
   Enquanto (sequencia <= 10) faca
     escreva(numero,"x",sequencia,"=",numero*sequencia)
     sequencia ← sequencia+1
   Fim_Enquanto
  Fim_Para
Fim
```

7.5.9 Encadeamento da estrutura para com repita

Exemplo de encadeamento da estrutura **para** com estrutura **repita**. Segue abaixo a representação em pseudocódigo:

Pseudocódigo:

```
Para <var> de <inicio> até <fim> [passo <incr>] faca
  Repita
    <instruções>
  Até que (<condição>)
  Fim_para
```

Resolução do problema da tabuada utilizando o encadeamento da estrutura **para** com **para.**

```
Algoritmo Tabuada_9
Var
  numero   : inteiro
  sequencia : inteiro
Inicio
  Para numero de 1 ate 10 faca
   sequencia ← 1
   Repita
     escreva(numero,"x",sequencia,"=",numero*sequencia)
     sequencia ← sequencia+1
   Ate que (sequencia > 10)
  Fim_Para
Fim
```

Resumo

Nesta aula foram discutidas as principais estruturas de repetição usadas na construção de algoritmos. Todas as repetições têm característica comum: o fato de haver uma verificação de condição que pode ser representada por um valor lógico, para determinar se a repetição prossegue ou não. Essa é a base para a implementação dos comandos de repetição em algoritmos. Em vez de fazermos trabalho braçal, escrevendo a mesma instrução várias vezes, poderemos utilizar uma estrutura que indique que tal instrução será executada quantas vezes for necessária.

O comando **enquanto** caracteriza-se por uma verificação de encerramento de atividades antes de se iniciar (ou reiniciar) a execução de seu bloco de instruções. O comando **repita** funciona de forma similar ao comando **enquanto,** exceto pelo fato de que a condição de controle só é testada após a execução do bloco de comandos e não antes, como é o caso do comando **enquanto.** Outra diferença é que os comandos serão repetidos pelo menos uma vez, já que a condição de parada se encontra no final. O comando **para** procura resumir as três características comuns à maioria das implementações de laços – são elas: preparação, condição e incremento - em uma só instrução, facilitando assim a construção típica de laços.

O encadeamento das estruturas de repetição, assim como nas estruturas de decisão, pode ser necessário dentro de um algoritmo. A existência destas

ocorrências vai depender do problema a ser solucionado e podem ocorrer todas as combinações possíveis das estruturas de repetição vistas nesta aula. Nesta aula foram mostradas todas as combinações, sendo estas aplicadas em um problema conhecido de elaboração de uma tabuada.

Atividades de aprendizagem

1. Faça um algoritmo que apresente todos os números divisíveis por 4 que sejam menores que 200.

```
algoritmo lista_div_por_4

var
  numero : inteiro
Inicio
numero ← 1
Enquanto (numero < 200) faca
  se (numero mod 4 = 0) entao
    escreva(numero)
  fim_se
  numero ← numero+1;
Fim_Enquanto
Fim
```

No Visualg:

```
algoritmo "lista_divisiveis_por_4"
var
  numero : inteiro
inicio

numero <-1
Enquanto (numero < 200) faca
  se (numero mod 4 =0) entao
    escreval(numero)
  fimse
  numero<-numero+1;
FimEnquanto
fimalgoritmo
```

2. Faça um algoritmo que receba 2 notas de 10 alunos e calcule e mostre:

- A média aritmética das duas notas de cada aluno
- A mensagem que está na tabela a seguir:

Média Aritmética	Mensagem
Menor que 3	Reprovado
Entre 3(inclusive) e 7	Exame Final
De 7 pra cima	Aprovado

```
algoritmo Media_Aritmética
var
   nota1,nota2,media : real
   qtalunos : inteiro
inicio
  qtalunos ← 1
  Repita
   Escreva("Entre com a 1a. nota do ",qtalunos,"o. Aluno:")
   leia(nota1)
   Escreva("Entre com a 2a. nota do ",qtalunos,"o. Aluno:")
   leia(nota2)
   media ← (nota1+nota2)/2
   se (media < 3) entao
   escreva("Reprovado")
   senao
    se (media >=3) e (media < 7) entao
     escreva("Exame final")
    senao
     se media > 7 entao
       escreva("Aprovado")
     fim_se
    fim_se
   fim_se
   qtalunos ← qtalunos+1
  ate que (qtalunos > 10)
Fim
```

No Visualg:

```
algoritmo "Media_Aritmética"
var
   nota1,nota2,media : real
   qtalunos : inteiro
inicio
// Seção de Comandos
  qtalunos <-1
  Repita
   Escreval("Entre com a 1a. nota do ",qtalunos,"o.
Aluno:")
   leia(nota1)
   Escreval("Entre com a 2a. nota do ",qtalunos,"o.
Aluno:")
   leia(nota2)
   media <- (nota1+nota2)/2
   se (media < 3) entao
   escreval("Reprovado")
   senao
    se (media >=3) e (media < 7) entao
     escreval("Exame fina")
    senao
     se media > 7 entao
       escreval("Aprovado")
     fimse
    fimse
   fimse
   qtalunos <- qtalunos+1
  ate (qtalunos > 10)
fimalgoritmo
```

3. Faça um algoritmo que efetue a leitura de valores positivos inteiros até que um valor negativo seja informado. Ao final deve ser apresentado o maior valor informado pelo usuário.

Comentário: Interpretando este problema, percebemos que não podemos resolvê-lo utilizando uma estrutura de repetição **PARA**, pois não sabemos a princípio quantos valores serão lidos. Portanto podemos aplicar as estruturas de repetição **ENQUANTO** ou **REPITA**. Vamos então construir o algoritmo utilizando a estrutura **ENQUANTO**.

```
Algoritmo MaiorValor
var
  valor, maiorvalor : inteiro
Inicio
  leia(valor)
  maiorvalor ← valor
  Enquanto (valor > 0) faca
    se (valor > maiorvalor) entao
      maiorvalor ← valor
    fim_se
    leia(valor)
  Fim_Enquanto
  Se (maiorvalor < 0) entao
    escreva("Não foi lido nenhum numero inteiro
positivo!")
  senao
    escreva("O maior valor é:",maiorvalor)
  fim_se
Fim.
```

No Visualg:

```
algoritmo "MaiorValor"
var
  valor, maiorvalor : inteiro
inicio

  leia(valor)
  maiorvalor <- valor
  Enquanto (valor > 0) faca
    se (valor > maiorvalor) entao
      maiorvalor<-valor
    fimse
    leia(valor)
  FimEnquanto
  Se (maiorvalor < 0) entao
    escreval("Não foi lido nenhum numero inteiro
positivo!")
  senao
    escreval("O maior valor é:",maiorvalor)
  fimse
fimalgoritmo
```

4. Faça um algoritmo que apresente a diferença da soma dos números pares pela soma dos números ímpares, que sejam menores que 100.

```
algoritmo Dif_Par_impar_100
var
  somaPar, somaImpar, diferenca,numero : inteiro
inicio
  Para numero de 1 ate 100 faca
    se (numero mod 2 = 1) entao
      somaImpar ← somaImpar+numero
    senao
      somaPar ← somaPar + numero
    fim_se
  Fimpara
  diferenca ← somaPar-somaImpar
  escreva("A diferença da soma dos pares pelos
Impares é:",diferenca)
fim
```

No Visualg:

```
algoritmo "Dif_Par_impar_100"
var
  somaPar, somaImpar, diferenca, numero : inteiro
inicio
  Para numero de 1 ate 100 faca
    se (numero mod 2 = 1) entao
      somaImpar <- somaImpar+numero
    senao
      somaPar <- somaPar + numero
    fimse
  Fimpara
  diferenca <- somaPar-somaImpar
  escreval("A diferença da soma dos pares pelos
Impares é:",diferenca)
fimalgoritmo
```

5. Faça um algoritmo que apresente no final o somatório dos números pares compreendidos entre 1 e 500.

6. Faça um algoritmo que apresente a soma e a média aritmética dos números divisíveis por 3 compreendidos entre 50 e 80.

7. Analise o seguinte algoritmo e complete-o de forma que o mesmo produza a seguinte soma para um determinado valor de N.

soma = 1 + 1/2 + 1/3 + 1/4 + 1/5 + ... + 1/N
Exemplo: para N= 6, temos:
Soma = 1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6
Soma = 1 + 0,5 + 0, 33 + 0,25 + 0,2 + 0,16 = 2,44 (aproximadamente)

```
Algoritmo somaTermos
Var
 n,__ : inteiro
 soma : real
Inicio
 ______
 Soma ← 0;
 Para i de 1 até __ faça
   soma ← soma + ____;
 Fim_para
 ________
Fim.
```

8. Faça um algoritmo para gerar o seguinte resultado, para os N primeiros termos:

Resultado = 1 – 2 + 3 – 4 + 5 – 6 + 7 – ...
Exemplo: Para N = 5, quer dizer que queremos calcular a expressão para os cinco primeiros números. Neste caso, temos:
Resultado = 1 – 2 + 3 – 4 + 5
Resultado= 3

9. Analise o seguinte algoritmo e escreva sua saída:

```
Algoritmo serie
Var
  cont, num1, num2, res : inteiro;
Inicio
  num1 ← 0
  num2 ← 1
  Escreva(num1)
  Escreva(num2)
  Para cont de 1 ate 8 faça
   res ← num1 + num2
   Escreva(res)
   num1 ← num2
   num2 ← res
  Fim_para
Fim
```

10. Faça um algoritmo que receba um numero inteiro maior que 1, verifique se o numero fornecido é primo ou não e mostre uma mensagem dizendo "o numero é primo" ou "o numero não é primo". Dica: Um número é primo quando ele é divisível apenas por 1 e por ele mesmo. Ex: O numero 7 é primo pois ele só é divisível (resto da divisão é igual a 0) por 1 e por 7. Já o número 6 não é primo pois o mesmo é divisível por 1, 2, 3 e 6.

11. O seguinte algoritmo faz o calculo da area de um triângulo. Modifique este algoritmo utilizando a estrutura de repetição **REPEAT** para que não seja permitido ler a variável base e a variável altura com o valor 0 (zero).

```
Algoritmo AreaTriangulo
Var
 base, altura : inteiro
 area : real
Inicio
 Leia (base)
 Leia (altura)
 area ← (base * altura)/2
 Escreva(area)
Fim
```

12. Faça um algoritmo que receba várias idades, calcule e mostre a média das idades lidas. Finalize o algoritmo caso o usuário digite o valor 0 (zero) para idade.

13. Faça um algoritmo que receba o nome, a idade e o sexo de 10 pessoas e mostre o nome da pessoa apenas se a pessoa for do sexo masculino e tiver mais de 21 anos.

14. Faça um algoritmo para ler uma quantidade N de números inteiros e mostre ao final quantos são pares e quantos são ímpares.

15. No campeonato brasileiro de basquete, se inscreveram 10 times. Sabendo-se que na lista oficial de cada time consta o peso e a idade dos 5 jogadores, crie um algoritmo que apresente as seguintes informações:

- O peso médio e a idade média de cada um dos times
- O peso médio e a idade media de todos os participantes

16. Uma empresa de fornecimento de energia elétrica faz a leitura mensal dos medidores de consumo. Para cada consumidor, são digitados os seguintes dados:

- Numero do consumidor
- Quantidade de KWh consumidos durante o mês.
- Tipo (código) do consumidor

1. – residencial, preço em reais por KWh = 0,3
2. – comercial, preço em reais por KWh = 0,5
3.– industrial, preço em reais por KWh = 0,7

Os dados devem ser lidos até que seja encontrado um consumidor com numero 0 (zero). O algoritmo deverá calcular e mostrar:

- O custo total para o consumidor
- O total de consumo para os três tipos de consumidor
- A media de consumo dos tipos 1 e 2

17. Crie um algoritmo que leia um conjunto de pedidos de compra e calcule o valor total da compra. Cada pedido é composto pelos seguintes campos:

- Numero do pedido
- Data do pedido (dia, mês, ano)
- Preço unitário
- Quantidade

O algoritmo deverá processar novos pedidos até que o usuário digite 0 (zero) como numero do pedido.

18. Reescreva o seguinte algoritmo utilizando estrutura de repetição **ENQUANTO**, de modo que produza a mesma saída.

```
algoritmo fatoriais
var
 fat : real
 i,num : inteiro
inicio
Para num de 1 ate 10 faca
  fat ← 1
  para i de 1 ate num faca
    fat ← fat*i
  fim_para
  escreva("fatorial de", num,": ",fat)
fim_para
Fim
```

19. Em uma eleição existem 4 candidatos. Os votos são informados por meio de código. Os códigos utilizados são:

1,2,3,4	Votos para os respectivos candidatos
5	Voto nulo
6	Voto em branco

Faça um algoritmo que calcule e mostre:

- O total de votos para cada candidato
- O total de votos nulos
- O total de votos brancos
- O percentual de votos nulos sobre o total de votos
- O percentual de votos brancos sobre o total de votos

Aula 8 - Estruturas homogêneas: vetores e matrizes

Objetivos

Conceituar as estruturas homogêneas e compreender suas utilizações em diversas situações práticas.

Aprender a declarar, preencher, atribuir valores e mostrar os elementos de um vetor ou matriz.

Utilizar vetores e matrizes na construção de algoritmos de acordo com os problemas apresentados

8.1 Introdução

Nas aulas anteriores, percebemos que muitos problemas podem ser resolvidos através dos algoritmos utilizando as estruturas vistas até agora. Porém, ainda não podemos afirmar que podemos resolver todo tipo de problema, pois foram trabalhadas apenas variáveis simples que armazenam somente um valor por vez.

Nesta aula, aprenderemos a trabalhar com o agrupamento de várias informações dentro de uma mesma variável. Vale salientar que esse agrupamento ocorrerá obedecendo sempre ao mesmo tipo de dado, e por esta razão é chamado de estrutura de dados homogênea.

A utilização deste tipo de estrutura de dados recebe diversos nomes, como: variáveis indexadas, variáveis compostas, variáveis subscritas, arranjos, vetores, matrizes, tabelas em memória ou *arrays*. Utilizaremos os termos **Vetores** e **Matrizes** para representar estas estruturas de dados homogêneas.

8.2 Vetores

8.2.1 Definição

Vetores são conhecidos como variáveis compostas homogêneas unidimensionais. Isto quer dizer que se trata de um conjunto de variáveis de mesmo tipo, que possuem o mesmo identificador (nome) e são armazenadas se-

quencialmente na memória. Como as variáveis tem o mesmo nome, o que as distingue é um índice que referencia sua localização dentro da estrutura, ou seja, podemos identificar o primeiro, o segundo, até o último elemento do conjunto que pode ser identificado por sua posição.

Fazendo uma analogia, podemos identificar a presença de um vetor à lista de alunos de uma turma em uma escola. Nessas escolas, os alunos recebem números conforme a ordem alfabética de seus nomes na turma. Cada aluno sabe seu número e, em certas oportunidades, o uso deste número é útil para cada um deles. Assim, se um professor colocar em edital as notas de uma turma e relacionar apenas os números dos alunos e suas respectivas notas, o aluno João da Silva, numero 22, pode descobrir sua nota olhando diretamente na linha onde se encontra o numero 22, sem precisar verificar as outras notas que não lhes interessam. Dizemos que este acesso é indexado ou que os vetores possuem índices. Estes termo foi criado como uma analogia aos livros , em que podemos acessar um capítulo específico diretamente na página que queremos, utilizando o índice do livro, que diz em que página cada capítulo ou seção se encontra. A utilização do índice evita buscas longas e desgastantes por todo o livro para encontrar o capitulo desejado.

8.2.2 Declaração

< identificador > : **Vetor** [< numero de elementos >] de <tipo>
Onde:
< identificador > é o nome da variável do tipo vetor;
< numero de elementos > é a quantidade de variáveis que vão compor o vetor;
< tipo > é o tipo básico dos dados que serão armazenados no vetor;

8.2.3 Exemplo de vetor

Notas : vetor[20] de real;

Notas

Notas[1]								...		
	1	2	3	4	5	6	7	...	19	20

A variável Notas pode armazenar até 20 notas dos alunos de uma classe.

O acesso a esses elementos é feito pela sua posição dentro do vetor. A posição é indicada pelo número entre colchetes. A primeira posição é a 1 e a última é o tamanho do vetor. Cada elemento do vetor, por meio de seu índice,

pode ser acessado como uma variável primitiva individual. Portanto se o vetor é de inteiros, cada elemento funciona como uma variável do tipo inteiro.

8.2.4 Atribuindo valores ao vetor

As atribuições em um vetor exigem que seja informada em qual de suas posições o valor ficará armazenado.

Notas[3] ← 6,5; Notas[1]← 8

Neste exemplo, os valores 6,5 e 8 serão armazenados respectivamente nas posições de índice 3 e 1 do vetor.

8.2.5 Preenchendo um vetor

Preencher um vetor significa atribuir valores a todas as suas posições. Assim, deve-se implementar um mecanismo que controle o valor do índice e a estrutura de repetição **PARA** é o mecanismo mais adequado. Ex:

```
...
Para i de 1 ate 5 faça
  Escreva (“digite o”, i, “° numero”);
  Leia(x)
Fim_para
```

Nesse exemplo, a estrutura de repetição **PARA** foi utilizada para garantir que a variável i assuma todos os valores possíveis para o índice do vetor. Assim, para cada execução da repetição, será utilizada uma posição diferente do vetor. A Figura 8.1 mostra a simulação passo-a-passo do preenchimento do vetor.

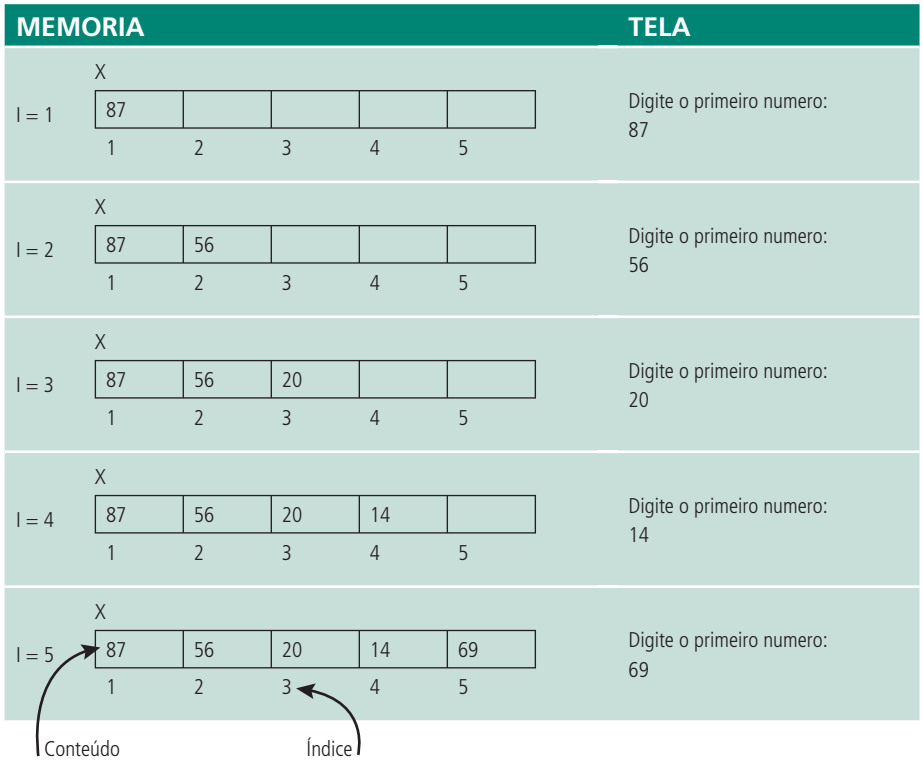


Figura 8.1: Simulação passo-a-passo do preenchimento de um vetor
Fonte: Ascencio, 2007

Tenha cuidado para não confundir o índice com o elemento. O índice é o endereço de alocação de uma unidade do vetor, enquanto elemento é o conteúdo armazenado em um determinado endereço.

8.2.6 Mostrando os elementos do vetor

Mostrar os valores contidos em um vetor também implica a utilização do índice.

```
Para i de 1 até 5 faça
        Escreva("Este é o ",i, "º elemento do vetor")
Fim_para
```

Nesse exemplo, assim como no exemplo anterior, a estrutura de repetição PARA foi utilizada para garantir que a variável i assuma todos os valores possíveis para o índice do vetor. Assim, para cada execução da repetição, será utilizada uma posição diferente e dessa forma, todos os valores do vetor serão mostrados.

Os índices utilizados para acessar elementos de um vetor devem pertencer ao intervalo do vetor, ou seja, deverão estar entre 1 e o tamanho do vetor. Assim, a operação a seguir é inválida, pois <BOLD>Notas<BOLD> foi declarada como um vetor de 20 elementos reais (1 até 20)

Notas [21] ←10 // Errado!

Vetores não podem ser acessados como um todo. Ao contrário, seus elementos precisam ser manipulados individualmente por meio do índice. Portanto, as operações seguintes são inválidas:

Escreva (Notas); // Errado!
Leia (Notas); // Errado!

A única exceção que faremos a esta regra será em relação aos vetores de caracteres. Neste caso, podemos tratar os valores individualmente, pelo índice, ou como variável primitiva. Nesses casos, a manipulação dos caracteres individualmente seria um trabalho grande e desnecessário. Portanto as operações seguintes são válidas:

```
NomeAluno : Vetor [50] de caractere;
Leia (NomeAluno)          // Correto, variável tratada como primitiva
Escreva(NomeAluno)        // Correto, variável tratada como primitiva
NomeAluno ← "Maria"       // Correto, variável tratada como primitiva
NomeAluno[7] ←"D"         // Correto, variável tratada como vetor
Escreva(NomeAluno[4])     // Correto, variável tratada como vetor
...
```

8.3 Matrizes

Existem certas estruturas que não podem ser representadas pelos vetores vistos até agora. Tais estruturas têm pelo menos uma dimensão a mais que os vetores. Nesta seção veremos estas estruturas multidimensionais, também conhecidas por matrizes.

8.3.1 Definição

Uma matriz é uma variável composta homogênea multidimensional. Ela é formada por uma sequência de variáveis, todas do mesmo tipo, com o identificador (mesmo nome), e armazenadas sequencialmente na memória. Uma

vez que as variáveis têm o mesmo nome, o que as distingue são índices que referenciam sua localização dentro da estrutura. Uma variável do tipo matriz precisa de um índice para cada uma de suas dimensões.

Este tipo de estrutura também tem sua principal utilização vinculada à criação de tabelas ou estruturas que precisem de dois parâmetros para identificá-las. Vejamos alguns exemplos:

- Tabuleiros de jogos em geral (xadrez, damas, etc.)
- Matrizes matemáticas
- Todos os pontos definidos em um monitor de vídeo
- O bilhete de um jogo de loteria
- Uma página de palavras-cruzadas

Geralmente utilizamos matrizes em situações que precisam de linhas e colunas para a identificação de elementos.

8.3.2 Declaração de uma matriz

Um algoritmo pode declarar uma matriz, conforme descrito a seguir:

<identificador> : **Matriz**[dimensão1, dimensão2,..., dimensãoN] de <tipo>

Onde:

<identificador> é o nome da variável do tipo Matriz;
<dimensão1> é a quantidade de elementos da 1ª dimensão (geralmente chamada de linha)
< dimensão2> é a quantidade de elementos da 2ª dimensão (geralmente chamada de coluna)
< dimensão2> é a quantidade de elementos da Nª dimensão
< tipo > é o tipo de dados da Matriz.

Exemplo: **VAR**
M : MATRIZ [1 .. 5 , 1 .. 10] **DE INTEIRO**

Também é possível definir matrizes com várias dimensões, por exemplo:

Ex.: **VAR**
N : MATRIZ [1 .. 4] **DE INTEIRO**
O : MATRIZ [1 .. 50 , 1 .. 4] **DE INTEIRO**
P : MATRIZ [1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**
Q : MATRIZ [1 .. 3 , 1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**
R : MATRIZ [1 .. 2 , 1 .. 3 , 1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**
S : MATRIZ[1 .. 2 , 1 .. 2 , 1 .. 3 , 1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**

A utilidade de matrizes desta forma é muito grande. No exemplo acima, cada matriz pode ser utilizada para armazenar uma quantidade maior de informações:

- a matriz **N** pode ser utilizada para armazenar 4 notas de um aluno
- a matriz **O** pode ser utilizada para armazenar 4 notas de 50 alunos.
- a matriz **P** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas.
- a matriz **Q** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas, de 3 turmas.
- a matriz **R** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas, de 3 turmas, de 2 colégios.
- a matriz **S** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas, de 3 turmas, de 2 colégios, de 2 cidades

8.3.3 Exemplo de matriz

O exemplo a seguir define uma matriz bidimensional, onde o tamanho da 1ª dimensão (Linha) é 3 e o da 2ª dimensão (coluna) é 5.

X: Matriz[1..3,1..5] de inteiro

	1	2	3	4	5
1	X[1,1]				
2					
3				X[4,3]	

8.3.4 Atribuindo valores a uma matriz

Cada elemento de uma matriz pode armazenar um valor. Para fazer este armazenamento, é necessário executar uma atribuição, informando o numero das dimensões.

X[2,4] ← 26; X[3,1] ← 13

8.3.5 Preenchendo uma matriz

Para preencher uma matriz, é necessário identificar todas as suas posições . Isto exige a utilização de um índice para cada dimensão da matriz.

No exemplo a seguir, uma matriz bidimensional com três linhas e cinco colunas são mostradas. Observe que a variável i varia dentro de um intervalo de 1 a 3, ou seja, exatamente as linhas. Para cada valor de i, a variável j varia de 1 a 5, ou seja, as cinco colunas que cada linha possui.

```
Para i de 1 até 3 faça
  Para j de 1 até 5 faça
     Escreva (“digite o numero da linha ”, i, “e coluna ”, j)
     Leia (X[i,j]);
  Fim_para
Fim_para
```

MEMORIA		TELA
I	J	
1	1	Digite o numero da linha 1 e coluna 1: 5
1	2	Digite o numero da linha 1 e coluna 2: 15
1	3	Digite o numero da linha 1 e coluna 3: 6
1	4	Digite o numero da linha 1 e coluna 4: 10
1	5	Digite o numero da linha 1 e coluna 5: 32
2	1	Digite o numero da linha 2 e coluna 1: 5
2	2	Digite o numero da linha 2 e coluna 2: 18
2	3	Digite o numero da linha 2 e coluna 3: 3
2	4	Digite o numero da linha 2 e coluna 4: 14
2	5	Digite o numero da linha 2 e coluna 5: 24
3	1	Digite o numero da linha 3 e coluna 1: 2
3	2	Digite o numero da linha 3 e coluna 2: 9
3	3	Digite o numero da linha 3 e coluna 3: 16
3	4	Digite o numero da linha 3 e coluna 4: 25
3	5	Digite o numero da linha 3 e coluna 5: 35

Assim podemos imaginar os elementos dispostos em uma estrutura bidimensional, como uma tabela:

5	15	6	10	32
5	18	3	14	24
2	9	16	25	35

8.3.6 Mostrando os elementos de uma matriz

Para mostrar os elementos de uma matriz, é necessário identificar suas posições. Isto exige a utilização de um índice para cada dimensão da matriz.

No exemplo a seguir, uma matriz bidimensional com três linhas e cinco colunas é mostrada. Observe que a variável i varia dentro de um intervalo de 1 a 3, ou seja, exatamente as linhas. Para cada valor de i, a variável j varia de 1 a 5, ou seja, as cinco colunas que cada linha possui.

```
Para i de 1 até 3 faça
  Para j de 1 até 5 faça
     Escreva (“Digite o numero da linha ”, i, “e coluna ”, j)
     Escreva (X[i,j]);
  Fim_para
Fim_para
```

Resumo

Nesta aula aprendemos como declarar, preencher, atribuir valores e mostrar os elementos de um vetor ou matriz. Também entendemos em quais situações devemos utilizar vetores e matrizes na construção de algoritmos de acordo com os problemas apresentados.

Atividade de aprendizagem

1. Desenvolver um algoritmo que efetue a leitura de 10 (dez) elementos de um vetor A. Construir um vetor B do mesmo tipo, observando a seguinte lei de formação: se p valor do índice for par, o valor deverá ser multiplicado por 5. Caso seja ímpar, deverá ser somado com 5. Ao final, mostrar o conteúdo do vetor B.

Resolução:

Este exemplo de resolução estará mostrando como fazer o tratamento da condição do índice.

a) Iniciar o contador de índice, variável i como 1 em um contador até 10;

b) Ler os 10 valores, um a um;

c) Verificar se o índice é par. Em caso afirmativo, multiplica por 5, se não soma 5. Criar a matriz B.

d) Apresentar os conteúdos das duas matrizes.

Deverá ser perguntado se o valor do índice i em um determinado momento é par (ele será par quando dividido por 2 obtiver resto igual a zero). Sendo a condição verdadeira, será implicada no vetor B[i] a multiplicação do elemento do vetor A[i] por 5. Caso o valor do índice i seja ímpar, será implicada no vetor B[i] a soma do elemento do vetor B[i] por 5.

```
Algoritmo Indice_Par_Impar;
Var
  A,B : vetor [1..10] de Real
  i,R : inteiro
Inicio
  Para i de 1 até 10 faça
   Leia (A[i])
  Fim_para
  Para i de 1 até 10 faça
   R ← (i mod 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 faça
   Escreva(B[i])
  Fim_para
Fim
```

No Visualg:

```
Algoritmo "Indice_Par_Impar"
Var
  A,B : vetor [1..10] de Real
  i,R : inteiro
Inicio
  Para i de 1 ate 10 faca
   Leia (A[i])
  Fimpara
  Para i de 1 ate 10 faca
   R <- (i mod 2)
   Se (r=0) entao
     B[i] <-A[i] * 5
   Senao
     B[i] <-A[i] + 5
   Fimse
  Fimpara
  Para i de 1 ate 10 faca
   Escreva(B[i])
  Fimpara
FimAlgoritmo
```

2. Crie um algoritmo que armazene o nome e duas notas de 10 alunos e mostre ao final uma listagem contendo o nome do aluno, a média obtida nas duas notas e uma mensagem "APROVADO" caso a media seja maior ou igual a seis e "REPROVADO", caso a media seja menor que 6.

Resolução:

Devemos criar 4 (quatro) vetores: Um para armazenar o nome dos alunos, outros dois para armazenar as notas e outro para armazenar a media. Uma vez que os vetores dos nomes dos alunos e das notas foram lidos, o vetor das medias é calculado com base nos valores das notas lidas de cada aluno. Tudo isto deve ser feito dentro de uma estrutura de repetição. Devemos criar outra estrutura de repetição para apresentar o resultado do algoritmo, ou seja, a relação dos nomes dos alunos, suas respectivas medias e a mensagem.

```
Algoritmo Calcula_media_alunos
Var
 Nomes : vetor [1..10] de caractere
 nota1, nota2, media : vetor [1..10] de real
 i : inteiro
Inicio
  Para i de 1 até 10 faça
   Leia(nomes[i])
   Leia(nota1[i]
   Leia(nota2[i])
   Media[i] ← (nota1[i]+nota2[i])/2
  Fim_para
  Para i de 1 até 10 faça
   Se media[i] >= 6 então
    Escreva(nomes[i],media[i], "Aprovado")
   Senão
    Escreva(nomes[i],media[i], "Reprovado")
   Fim_se
  Fim_para
Fim.
```

No Visualg:

```
Algoritmo "Calcula_media_alunos"
Var
 nomes : vetor [1..10] de caractere
 nota1, nota2, media : vetor [1..10] de real
 i : inteiro
Inicio
  Para i de 1 ate 10 faca
   Escreval("Entre com o nome do ",i,"º aluno:")
   Leia(nomes[i])
   Escreval("Entre com a primeira nota do ",i,"º aluno:")
   Leia(nota1[i])
   Escreval("Entre com a segunda nota do ",i,"º aluno:")
   Leia(nota2[i])
   Media[i] <-(nota1[i]+nota2[i])/2
  Fimpara
  Para i de 1 ate 10 faca
   Se media[i] >= 6 entao
    Escreval(nomes[i]," ",media[i], " Aprovado")
   Senao
    Escreval(nomes[i]," ",media[i], " Reprovado")
   Fimse
  Fimpara
Fimalgoritmo
```

3. Crie um Algoritmo que armazene cinco nomes de pessoas em um vetor e depois possa ser digitado um nome para procura e se for encontrado, mostrar a posição desse nome no vetor, caso contrário, mostrar a mensagem: "Nome não encontrado!"

Resolução:

Devemos criar um vetor de caractere para armazenar os nomes das pessoas, uma variável nome para receber o nome da pessoa para pesquisa, uma variável i para manipular os índices do vetor e uma variável lógica para indicar se o nome foi encontrado ou não no vetor. Essa variável lógica tem um papel muito importante na construção deste algoritmo pois sem ela o algoritmo ficaria muito complicado de se entender.

Utilizaremos uma estrutura de repetição do tipo **REPITA** pois durante a pesquisa no vetor, podemos encontrar o nome em uma posição bem antes da posição final do vetor. Em outras palavras, depois que encontramos o nome no vetor, não somos mais obrigados a percorrê-lo até o final. Com isso economizamos tempo de processamento em um programa de computador e otimizamos nosso algoritmo.

```
Algoritmo procura_pelo_nome
Var
 i : inteiro
 nomes : vetor [1..5] de caractere;
 nome : caractere;
Inicio
 Para i de 1 a 5 faça
  Leia(nomes[i])
 Fim_para
 Escreva("Entre com o nome para pesquisa:");
 Leia(nome)
 i ←1
 Encontrei ← falso
 Repita
  Se nome = nomes[i] então
   Encontrei ← Vedadeiro
  Fim_se
  i ← i+1
 Até que (i > 5) ou (encontrei = verdadeiro)
 Se (encontrei = verdadeiro) então
  Escreva("O nome foi encontrado na posição:",i)
 Senao
  Escreva("O nome não foi encontrado")
 Fim_se
Fim
```

No Visualg:

```
Algoritmo "procura_pelo_nome"
Var
 i : inteiro
 nomes : vetor [1..5] de caractere
 nome : caractere
 encontrei : Logico
Inicio
 Para i de 1 ate 5 faca
  Escreval("Entre com o ",i,"° nome:")
  Leia(nomes[i])
 Fimpara
 Escreva("Entre com o nome para pesquisa:")
 Leia(nome)
 i<-1
 encontrei <- falso
 Repita
  Se nome = nomes[i] entao
   encontrei <- verdadeiro
  Fimse
  i <-i+1
 Ate (i > 5) ou (encontrei = verdadeiro)
 Se (encontrei = verdadeiro) entao
  Escreval("O nome foi encontrado na posição:",i)
 Senao
  Escreval("O nome não foi encontrado")
 Fimse
Fimalgoritmo
```

4. Crie um algoritmo que leia todos os elementos de uma matriz inteira de ordem 3 e escreva os elementos da diagonal principal, a soma dos elementos da diagonal secundária.

Resolução:

Vamos observar uma matriz de ordem 3 (3 linhas e 3 colunas). Quando falamos assim, estamos nos referindo a uma matriz quadrada.

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

Os elementos que se encontram nas posições [1,1], [2,2], [3,3] formam a **diagonal principal**. Observe que o numero da linha é sempre igual ao numero da coluna.

Os elementos que se encontram nas posições [1,3], [2,2], [3,1] formam a **diagonal secundária**. Observe que o numero da linha somado ao numero da coluna é sempre igual a 4, ou seja, o numero da ordem da matriz somado de 1 (L+C=ordem+1).

Outras regras interessantes são:

- Elementos acima da diagonal principal: numero da linha sempre menor que o numero da coluna(L < C)

- Elementos acima da diagonal secundária: numero da linha somado com o número da coluna é sempre menor ou igual a ordem da matriz (L+C <= ordem)

- Elementos abaixo da diagonal principal: numero da linha é sempre maior que o numero da coluna (L >C)

- Elementos abaixo da diagonal secundária: numero da linha somado com o numero da coluna é sempre maior ou igual a ordem da matriz somada com 2 (L+C >= ordem+2)

Para que possamos mostrar ou ler os elementos de uma matriz, precisaremos estruturas de repetição encadeadas e a estrutura **PARA** é a mais adequada para este tipo de operação.

```
Algoritmo Matrizes_diagonais;
Var
 A : matriz[1..3,1..3] de inteiro
 lin, col : inteiro
 somaDS : inteiro
Inicio
 SomaDS ←0
 Para lin de 1 até 3 faça
  Para col de 1 até 3 faça
   Leia(A[lin,col])
   Se (lin+col) = 4 então
    somaDS ← somaDS+A[lin,col]
   Fim_se
  Fim_para
 Para lin de 1 até 3 faça
  Para col de q até 3 faça
   Se lin=col então
    Escreva (A[lin,col])
   Fim_se
  Fim_para;
 Fim_para
 Escreva("A soma dos elementos da
diagonal secundária e:", somaDS)
Fim.
```

```
No Visualg:

Algoritmo "Matrizes_diagonais";
Var
 A : vetor [1..3,1..3] de inteiro
 lin, col : inteiro
 somaDS : inteiro
Inicio
 somaDS <-0
 Para lin de 1 ate 3 faca
  Para col de 1 ate 3 faca
   Leia(A[lin,col])
   Se (lin+col) = 4 entao
    somaDS <- somaDS+A[lin,col]
   Fimse
  Fimpara
 Fimpara
 Para lin de 1 ate 3 faca
  Para col de 1 ate 3 faca
   Se lin=col entao
    Escreva (A[lin,col])
   Fimse
  Fimpara
 Fimpara
 Escreval()
 Escreva("A soma dos elementos da diagonal secundaria
e: ", somaDS)
Fimalgoritmo
```

5. Faça um algoritmo que preencha uma vetor com 5 alunos e uma matriz 5 x 3 com as notas desses 5 alunos em 3 provas. O algoritmo deverá mostrar o nome dos alunos, e suas respectivas medias , sendo que a média será calculada com base nas duas maiores notas de cada aluno, ou seja, a menor nota de cada aluno deverá ser descartada.

Resolução:

Para resolver este problema, devemos criar um vetor de caractere para armazenar os nomes dos alunos, uma matriz de 5x3 para armazenar as 3 provas dos 5 alunos, um vetor de real para armazenar as medias dos alunos, uma variável de nome **menor** para descartar a menor nota e uma variável **somanota** para somar as duas maiores notas de cada aluno.

Vamos utilizar estruturas de repetição encadeadas pois estamos trabalhando com matrizes. No primeiro momento, encadeamos duas estruturas de repetição do tipo **PARA** para lermos os nomes dos alunos e suas respectivas notas. No segundo momento encadeamos novamente duas estruturas de repetição do tipo PARA para fazermos o cálculo das medias e mostra o resultado final.

```
Algoritmo Media_provas
Var
  alunos : vetor [1..5] de caractere
  notas : matriz [1..5,1..3] de real
  media : vetor [1..5] de real
  menor, somanotas : real;
  i,j : inteiro
Inicio
  Para i de 1 ate 5 faça
   Leia (alunos[i]
   Para j de 1 até 3 faça
     Leia (notas[i,j])
   Fim_para
  Fim_para
  Para i de 1 até 5 faça
   menor ← notas [i,1];
   somanotas ← 0
   Para j de 1 ate 3 faça
    Leia (notas[i,j]
    Se notas[i,j] < menor então
     menor ← notas[i,j];
    Fim_se
    somanota ← somanota+notas[i,j]
   Fim_para
   media[i]← (somanota – menor)/2
   Escreva(nomes[i],” “,”Media das duas maiores
provas:”, media[i])
  Fim_para;
Fim
```

No Visualg:

```
Algoritmo “Media_provas”
Var
  alunos : vetor [1..5] de caractere
  notas : vetor [1..5,1..3] de real
  media : vetor [1..5] de real
  menor, somanotas : real
  i,j : inteiro
Inicio
  Para i de 1 ate 5 faca
   Escreva(“Entre com o nome do “,i,”º Aluno:”)
   Leia (alunos[i])
   Para j de 1 ate 3 faca
     escreva(“Entre com a “,j,”ª nota do aluno:”)
     Leia (notas[i,j])
   Fimpara
  Fimpara
  Para i de 1 ate 5 faca
   menor <-notas[i,1]
   somanotas <-0
   Para j de 1 ate 3 faca
    Se notas[i,j] < menor entao
     menor <-notas[i,j]
    Fimse
    somanotas <-somanotas+notas[i,j]
   Fimpara
   Media[i]<-(somanotas - menor)/2
   Escreval(alunos[i],” “,”media : “,media[i])
  Fimpara
FimAlgoritmo
```

6. Faça um algoritmo que receba:

- Um vetor com o nome de 4 cidades diferentes
- Uma matriz 4 x 4 com a distancia entre as cidades, sendo que a diagonal principal deve ser colocada automaticamente a distancia 0 (zero), ou seja, não deve ser permitida a digitação.
- O consumo de combustível de um veículo, ou seja, quantos quilômetros este veículo percorre com um litro de combustível.

O algoritmo deverá calcular e mostrar todos os percursos (nome da cidade de origem e nome da cidade destino), juntamente com a quantidade de combustível necessária para o veiculo percorrê-los.

Algoritmo Mostra_percurso

```
Var
  cidades : vetor [1..4] de caractere
  distancia : matriz [1..4,1..4] de real
  consumo, quantidade : real
  i, j : inteiro
Inicio
  Para i de 1 ate 4 faça
   Leia (cidades[i])
   Para j de 1 até 4 faça
    Se i = j então
      distancia[i,j] ← 0;
    Senão
     Leia (distancia[i,j])
    Fim_Se
   Fim_para
  Fim_para
  Leia(consumo)
  Para i de 1 até 4 faça
   Para j de 1 até 4 faça
    Se i <> j então
     quantidade ← distancia[i,j]/consumo
     Escreva("A distância em Km de", cidade[i],
         " para ", cidade[j], "é de: ", distancia[i,j],
        " e o consumo de combustível e de", quantidade)
    Fim_se
   Fim_para
  Fim_para
Fim
```

No Visualg:

```
Algoritmo "Mostra_percurso"
Var
  cidades : vetor [1..4] de caractere
  distancia :vetor [1..4,1..4] de real
  consumo, quantidade : real
  i,j : inteiro
Inicio
  Para i de 1 ate 4 faca
   Escreva("Entre com a ",i,"ª cidade: ")
   Leia (cidades[i])
  Fimpara
  Para i de 1 ate 4 faca
    Para j de 1 ate 4 faca
    Se i = j entao
      distancia[i,j]<-0
    Senao
     Escreval("Entre com a distância entre ",cidades[i]," e
          ", cidades[j],":")
     Leia (distancia[i,j])
    FimSe
   Fimpara
  Fimpara
  Escreval("Entre com o consumo de combustível em
Km/l:")
  Leia(consumo)
  Para i de 1 ate 4 faca
   Para j de 1 ate 4 faca
    Se i <> j entao
     quantidade<-(distancia[i,j]/consumo)
     Escreva("A distancia em Km de", cidades[i]," para
         ", cidades[j]," é de: ", distancia[i,j])
     Escreval(" e o consumo de combustível e de",
quantidade)
    Fimse
   Fimpara
  Fimpara
FimAlgoritmo
```

7. Faça um algoritmo que leia dois vetores A e B de 10 elementos inteiros cada um e mostre um terceiro vetor C formado pela soma dos elementos dos vetores A e B.

8. Faça um algoritmo para corrigir provas de múltipla escolha. Cada prova tem oito questões e cada questão vale um ponto. O primeiro conjunto de dados a ser lido é o gabarito da prova. Os outros dados são a quantidade de alunos e as respostas que deram as questões. Existem 10 (dez) alunos matriculados. O algoritmo deverá calcular e mostrar o numero e a nota de cada aluno e a mensagem "Aprovado" caso obtenha nota maior ou igual a 6 (seis), e "Reprovado" em caso contrário.

9. Analise o seguinte algoritmo e escreva o vetor resultante correspondente ao seu resultado. Vetor inicial:

3	5	4	2	1	6	8	7	11	9

```
Algoritmo XXX;
Var
  vet : vetor [1..10; de inteiro
  i, j, aux : inteiro
Inicio
  Para i de 1 ate 10 faça
   Para j de 1 ate 9 faça
    Se vet[j] < vet [j+1] então
     aux ← vet[j]
     vet[j] ← vet[j+1]
     vet[j+1] ← aux
    Fim_Se
   Fim_para
  Fim_para
  Para i de 1 ate 10 faça
   Escreva(vet[i])
  Fim_para
Fim.
```

10. Faça um algoritmo que preencha um vetor com dez números inteiros, calcule e mostre os números superiores a 50 (cinquenta) e suas respectivas posições. O algoritmo deverá mostrar uma mensagem caso não exista nenhum numero que satisfaça esta condição.

11. Faça um algoritmo que preencha um vetor com os modelos de cinco carros (Ex: Fusca, Gol, Palio, etc). Preencha outro vetor com o consumo desses carros, isto é, quantos quilômetros cada um deles faz com um litro de combustível. O algoritmo deverá mostrar:

- O modelo de carro mais econômico
- Quantos litros de combustível cada um dos carros cadastrados consomem para percorrer uma distancia de 500 quilômetros.

12. Faça um algoritmo que armazene o nome e o salário de 20 pessoas e calcule e armazene o novo salário sabendo-se que o reajuste foi de 8%. Mostre ao final o nome das pessoas e seus respectivos salários reajustados.

13. Faça um algoritmo que leia vários números inteiros e positivos. A leitura se encerra quando for lido um numero negativo ou quando o vetor ficar completo. Sabendo-se que o vetor possui no máximo 10 (dez) elementos, o algoritmo deverá gerar e mostrar um vetor onde cada elemento é o quadrado do elemento correspondente do vetor original.

14. Faça um algoritmo para realizar as reservas de passagens aéreas de uma companhia. Além da leitura.

Aula 9 - Sub-rotinas

Objetivos

Conceituar as sub-rotinas e compreender as vantagens de sua utilização na construção de algoritmos.

Entender o mecanismo de funcionamento de uma sub-rotina dentro de um algoritmo.

Diferenciar os tipos de sub-rotinas existentes e aplicá-las dentro de um algoritmo através de chamadas.

Utilizar parâmetros nas sub-rotinas como meio de comunicação entre o programa principal e outras sub-rotinas.

Diferenciar parâmetros reais e formais.

Diferenciar passagem de parâmetros por valor e por referencia e entender o que isso afeta dentro de um algoritmo.

A partir desta aula será estudada a aplicação de sub-rotinas em algoritmos, também conhecidas por módulos, subprogramas ou sub-rotinas. Na realidade não importa como são chamadas, o que importa é a forma como funcionam e como devem ser aplicadas em um programa e é isto que o aluno aprenderá a partir de agora.

A complexidade dos algoritmos está intimamente ligada à da aplicação a que se destinam. Em geral, problemas complicados exigem algoritmos extensos para sua solução. Sempre é possível dividir problemas grandes e complicados em problemas menores e de solução mais simples. Assim, pode-se solucionar cada um destes pequenos problemas separadamente, criando algoritmos para tal (sub-rotinas). Posteriormente, pela justaposição destas sub-rotinas elabora-se "automaticamente" um algoritmo mais complexo e que soluciona o problema original. Esta metodologia de trabalho é conhecida como **Método de Refinamentos Sucessivos**, cujo estudo é assunto de cursos avançados sobre técnicas de programação.

Uma sub-rotina é um nome dado a um trecho de um algoritmo mais complexo e que, em geral, encerra em si próprio um pedaço da solução de um problema maior – o algoritmo a que ela está subordinada. Este conceito é essencial numa ciência bastante recente: a Engenharia de Software.

Em resumo, as sub-rotinas são importantes na:

- subdivisão de algoritmos complexos, facilitando o seu entendimento;
- estruturação de algoritmos, facilitando principalmente a detecção de erros e a documentação de sistemas; e
- modularização de sistemas, que facilita a manutenção de softwares e a reutilização de sub-rotinas já implementadas.

A ideia da reutilização de *software* tem sido adotada por muitos grupos de desenvolvimento de sistemas de computador, devido à economia de tempo e trabalho que proporcionam. Seu princípio é o seguinte: um conjunto de algoritmos destinado a solucionar uma série de tarefas bastante corriqueiras é desenvolvido e vai sendo aumentado com o passar do tempo, com o acréscimo de novos algoritmos. A este conjunto dá-se o nome de **biblioteca**. No desenvolvimento de novos sistemas, procura-se ao máximo basear sua concepção em sub-rotinas já existentes na biblioteca, de modo que a quantidade de *software* realmente novo que deve ser desenvolvido é minimizada.

Muitas vezes as sub-rotinas podem ser úteis para encerrar em si uma certa sequência de comandos que é repetida várias vezes num algoritmo. Nestes casos, as sub-rotinas proporcionam uma diminuição do tamanho de algoritmos maiores. Antigamente, esta propriedade era tida como a principal utilidade das sub-rotinas.

9.1 Mecanismo de funcionamento

Um algoritmo completo é dividido num **algoritmo principal** e diversas **sub-rotinas** (tantos quantos forem necessários e/ou convenientes). O **algoritmo principal** é aquele por onde a execução do algoritmo sempre se inicia. Este pode eventualmente invocar as demais sub-rotinas. O corpo do algoritmo principal é sempre o último trecho do pseudocódigo de um algoritmo. As definições das sub-rotinas estão sempre colocadas no trecho após a definição das variáveis globais e antes do corpo do algoritmo principal:

```
ALGORITMO <nome do algoritmo>
Var <definição das variáveis globais>

<definições das sub-rotinas>

Inicio
  <corpo do algoritmo principal>
Fim.
```

Durante a execução do algoritmo principal, quando se encontra um comando de invocação de uma sub-rotina, a execução do mesmo é interrompida. A seguir, passa-se à execução dos comandos do corpo da sub-rotina. Ao seu término, retoma-se a execução do algoritmo que o chamou (no caso, o algoritmo principal) no ponto onde foi interrompida (comando de chamada da sub-rotina) e prossegue-se pela instrução imediatamente seguinte.

Note, também, que é possível que uma sub-rotina chame outra através do mesmo mecanismo.

9.2 Definição de sub-rotinas

A definição de uma sub-rotina consta de:

- Um **cabeçalho**, onde estão definidos o **nome** e o **tipo** da sub-rotina, bem como os seus **parâmetros** e **variáveis locais;**

- Um **corpo**, onde se encontram as instruções (comandos) da sub-rotina.

O **nome** de uma sub-rotina é o nome simbólico pelo qual ela é chamada por outro algoritmo.

O **corpo** da sub-rotina contém as instruções que são executadas cada vez que ela é invocada.

Variáveis locais são aquelas definidas dentro da própria sub-rotina e só podem ser utilizadas pela mesma.

Parâmetros são canais por onde os dados são transferidos pelo algoritmo chamador a uma sub-rotina, e vice-versa. Para que possa iniciar a execução

das instruções em seu corpo, uma sub-rotina às vezes precisa receber dados do algoritmo que o chamou e, ao terminar sua tarefa, a sub-rotina deve fornecer ao algoritmo chamador os resultados da mesma. Esta comunicação bidirecional pode ser feita de dois modos que serão estudados mais à frente: por meio de **variáveis globais** ou por meio da **passagem de parâmetros.**

O **tipo** de uma sub-rotina é definido em função do número de valores que a sub-rotina retorna ao algoritmo que a chamou. Segundo esta classificação, as sub-rotinas podem ser de dois tipos:

- **funções**, que retornam um, e somente um, valor ao algoritmo chamador;
- **procedimentos**, que retornam zero (nenhum) ou mais valores ao algoritmo chamador.

Na realidade, a tarefa desempenhada por uma sub-rotina do tipo **função** pode perfeitamente ser feita por outra do tipo **procedimento** (o primeiro é um caso particular deste). Esta diferenciação é feita por razões históricas, ou, então, pelo grande número de sub-rotinas que se encaixam na categoria de **funções**.

9.3 Funções

O conceito de **função** é originário da ideia de função matemática (por exemplo, raiz quadrada, seno, cosseno, tangente, logaritmo, entre outras), onde um valor é calculado a partir de outro(s) fornecido(s) à função. A sintaxe da definição de uma função é dada a seguir:

```
FUNÇÃO <nome> ( <parâmetros> ) <tipo_de_dado>
VAR
 <variáveis locais>
INÍCIO
  <instruções>
FIM
```

Temos que:

- ***<nome>*** é o nome simbólico pelo qual a função é invocada por outros algoritmos;

- ***<parâmetros>*** são os parâmetros da função;

- ***<tipo de dado>*** é o tipo de dado da informação retornado pela função ao algoritmo chamador;

- ***<variáveis locais>*** consiste na definição das variáveis locais à função. Sua forma é análoga à da definição de variáveis num algoritmo;

- ***<instruções>*** é o conjunto de instruções do corpo da função.

Dentro de um algoritmo, o comando de invocação de uma sub-rotina do tipo função sempre aparece dentro de uma expressão do mesmo tipo que o do valor retornado pela função. A invocação de uma função é feita pelo simples aparecimento do nome da mesma, seguido pelos respectivos parâmetros entre parênteses, dentro de uma expressão. A função é executada e, ao seu término, o trecho do comando que a invocou é substituído pelo valor retornado pela mesma dentro da expressão em que se encontra, e a avaliação desta prossegue normalmente.

Dentro de uma função, e somente neste caso, o comando **Retorne *<expressão>*** é usado para retornar o valor calculado pela mesma. Ao encontrar este comando, a expressão entre parênteses é avaliada, a execução da função é terminada neste ponto e o valor da expressão é retornado ao algoritmo chamador. Vale lembrar que uma expressão pode ser uma simples constante, uma variável ou uma combinação das duas por meio de operadores. Esta expressão deve ser do mesmo tipo que o valor retornado pela função.

O algoritmo a seguir é um exemplo do emprego de função para calcular o valor de um número elevado ao quadrado.

```
Algoritmo Exemplo_de_funcao
Var
 X, Y : real

Função Quad(w : real) : real
Var
 Z : real
Inicio
 Z ← w * w
 Retorne Z
Fim
```

```
Inicio
 Escreva (“Digite um número”)
 Leia (X)
 Y ← Quad(X)
 Escreva (X, “ elevado ao quadrado é = “, Y)
Fim.
```

Do exemplo anterior é importante notar que:

- A função **Quad** toma **w** como parâmetro do tipo real, retorna um valor do tipo

- O comando de invocação da função **Quad** aparece no meio de uma expressão, no comando de atribuição dentro do algoritmo principal.

9.4 Procedimentos

Um procedimento é uma sub-rotina que retorna zero (nenhum) ou mais valores a um algoritmo chamador ou a outra sub-rotina. Estes valores são sempre retornados por meio dos parâmetros ou de variáveis globais, mas nunca explicitamente, como no caso de funções. Portanto, a chamada de um procedimento nunca surge no meio de expressões, como no caso de funções. Pelo contrário, a chamada de procedimentos só é feita em comandos isolados dentro de um algoritmo, como as instruções de entrada (Leia) e saída (Escreva) de dados.

A sintaxe da definição de um procedimento é:

```
PROCEDIMENTO <nome> ( <parâmetros> )

Var
  <variáveis locais>
Inicio
  <instruções>
Fim.
```

Temos que:

- ***<nome>*** é o nome simbólico pelo qual o procedimento é invocado por outros algoritmos;

- ***<parâmetros>*** são os parâmetros do procedimento;

- ***<variáveis locais>*** são as definições das variáveis locais ao procedimento. Sua forma é análoga à da definição de variáveis num algoritmo;

- ***<instruções>*** é o conjunto de instruções do corpo do procedimento, que é executado toda vez que o mesmo é invocado.

O exemplo a seguir é um exemplo simples, onde um procedimento é usado para escrever o valor das componentes de um vetor.

```
Algoritmo Exemplo_procedimento
Var
 vet : vetor [1..10] de real

Procedimento ESC_VETOR()
Var
 i : inteiro
Inicio
 Para i de 1 até 10 faça
   Escreva (vet[i])
 Fim_para

Procedimento LER_VETOR()
Var
 i : inteiro
Inicio
 Para i de 1 até 10 faça
   Leia (vet[i])
 Fim_para
Fim

Inicio
  LER_VETOR()
  ESC_VETOR()
Fim.
```

No exemplo é conveniente observar:

- A forma de definição dos procedimentos LER_VETOR( ) e ESC_VETOR( ), que não possuem nenhum parâmetro, e usam a variável local **i** para "varrer" os componentes do vetor, lendo e escrevendo um valor por vez; e

- A forma de invocação dos procedimentos, por meio do seu nome, seguido de seus eventuais parâmetros (no caso, nenhum), num comando isolado dentro do algoritmo principal.

9.5 Variáveis globais e locais

Variáveis globais são aquelas declaradas no início de um algoritmo. Estas variáveis são **visíveis** (isto é, podem ser usadas) no algoritmo principal e por todas as demais sub-rotinas.

Variáveis locais são aquelas definidas dentro de uma sub-rotina e, portanto, somente **visíveis** (utilizáveis) dentro do mesmo. Outras sub-rotinas, ou mesmo o algoritmo principal, não podem utilizá-las.

No exemplo a seguir são aplicados estes conceitos.

```
Algoritmo Exemplo_variáveis_locais_e_globais
Var
  X : real
  I : inteiro

Função FUNC() : real
Var
  X : vetor [1..5] de inteiro
  Y : caracter[10]
Inicio
...
Fim

Procedimento PROC
Var
 Y : lógico
Inicio
...
```

```
 X := 4 * X
 I := I + 1
...
Fim

Início
...
 X := 3.5
...
Fim.
```

É importante notar no exemplo anterior que:

- As variáveis X e **I** são globais e visíveis a todas as sub-rotinas, à exceção da função **FUNC**, que redefine a variável **X** localmente;

- As variáveis **X** e **Y** locais ao procedimento FUNC não são visíveis ao algoritmo principal ou ao procedimento **PROC**. A redefinição local do nome simbólico **X** como um vetor[5] de inteiro sobrepõe (somente dentro da função **FUNC**) a definição global de **X** como uma variável do tipo real;

- A variável **Y** dentro do procedimento **PROC**, que é diferente daquela definida dentro da função **FUNC**, é invisível fora deste procedimento;

- A instrução **X := 3.5** no algoritmo principal, bem como as instruções **X := 4 * X** e **I :=I + 1** dentro do procedimento **PROC**, atuam sobre as variáveis globais X e I.

9.6 Parâmetros

Parâmetros são canais pelos quais se estabelece uma comunicação bidirecional entre uma sub-rotina e o algoritmo chamador (o algoritmo principal ou outra sub-rotina). Dados são passados pelo algoritmo chamador à sub-rotina, ou retornados por este ao primeiro por meio de parâmetros.

Parâmetros formais são os nomes simbólicos introduzidos no cabeçalho de sub-rotinas, usados na definição dos parâmetros do mesmo. Dentro de uma sub-rotina trabalha-se com estes nomes da mesma forma como se trabalha com variáveis locais ou globais.

```
Função Media(X, Y : real) : real
Inicio
Retorne (X + Y) / 2
Fim
```

No exemplo anterior, **X** e **Y** são parâmetros formais da função **Media**.

Parâmetros reais são aqueles que substituem os parâmetros formais quando da chamada de uma sub-rotina. Por exemplo, o trecho seguinte de um algoritmo invoca a função Media com os parâmetros reais 8 e 7 substituindo os parâmetros formais X e Y.

Z ← Media(8,7)

Assim, os parâmetros formais são úteis somente na definição (formalização) da sub-rotina, ao passo que os parâmetros reais substituem-nos a cada invocação do sub-rotina. Note que os parâmetros reais podem ser diferentes a cada invocação de uma sub-rotina.

9.7 Mecanismos de passagem de parâmetros

Como foi visto anteriormente, os parâmetros reais substituem os formais no ato da invocação de uma sub-rotina. Esta substituição é denominada passagem de parâmetros e pode se dar segundo dois mecanismos distintos: passagem por **valor** (ou por cópia) ou passagem por **referência**.

9.7.1 Passagem de parâmetros por valor

Na passagem de parâmetros por valor (ou por cópia) o parâmetro real é calculado e uma cópia de seu valor é fornecida ao parâmetro formal, no ato da invocação da sub-rotina. A execução da sub-rotina prossegue normalmente e todas as modificações feitas no parâmetro formal não afetam o parâmetro real, pois trabalha-se apenas com uma cópia do mesmo.

```
Algoritmo Exemplo_parametro_por_valor
Var
 X : inteiro
```

```
Procedimento PROC(Y : inteiro)
Início
  Y ← Y + 1
  Escreva("Durante Y = ", Y)
Fim


Início
  X ← 1
  Escreva("Antes X = ", X)
  PROC(X)
  Escreva("Depois X = ", X)
Fim.
```

O algoritmo anterior fornece o seguinte resultado:

Antes X = 1
Durante Y = 2
Depois X = 1

Isto certifica que o procedimento não alterou o valor do parâmetro real X durante sua execução. Este tipo de ação é possível porque, neste mecanismo de passagem de parâmetros, é feita uma reserva de espaço em memória para os parâmetros formais, para que neles seja armazenada uma cópia dos parâmetros reais.

9.7.2 Passagem de parâmetros por referência

Neste mecanismo de passagem de parâmetros não é feita uma reserva de espaço em memória para os parâmetros formais. Quando uma sub-rotina com parâmetros passados por referência é chamado, o espaço de memória ocupado pelos parâmetros reais é compartilhado pelos parâmetros formais correspondentes. Assim, as eventuais modificações feitas nos parâmetros formais também afetam os parâmetros reais correspondentes.

Uma mesma sub-rotina pode utilizar diferentes mecanismos de passagem de parâmetros, para parâmetros distintos. Para diferenciar uns dos outros, convencionou-se colocar o prefixo **VAR** antes da definição dos parâmetros formais passados por referência. Se por exemplo um procedimento tem o seguinte cabeçalho:

Procedimento PROC(X, Y : inteiro; Var Z : real; J: real)

Então:

- **X** e **Y** são parâmetros formais do tipo inteiro e são passados por valor;
- **Z** é um parâmetro formal do tipo real passado por referência;
- **J** é um parâmetro formal do tipo real passado por valor.

O exemplo do item anterior, alterado para que o parâmetro **Y** do procedimento seja passado por referência, torna-se:

```
Algoritmo Exemplo_parametro_por_referencia
Var
  X : inteiro

Procedimento PROC(Y : inteiro)
Início
  Y ← Y + 1
  Escreva("Durante Y = ", Y)
Fim

Início
  X ← 1
  Escreva("Antes X = ", X)
  PROC(X)
  Escreva("Depois X = ", X)
Fim.
```

O resultado do algoritmo modificado é:

Antes X = 1
Durante Y = 2
Depois X = 2

Como podemos observar, depois de chamar o procedimento com o parâmetro por referência o valor original da variável foi alterado.

Resumo

Nesta aula foram discutidas as definições, características e utilização de sub-rotinas na construção de algoritmos. O mecanismo de funcionamento de uma sub-rotina é iniciado quando se encontra um comando de invocação desta, durante a execução do algoritmo principal. A partir daí a execução do mesmo é interrompida e a seguir, passa-se à execução dos comandos do corpo da sub-rotina. Ao seu término, retoma-se a execução do algoritmo que o chamou (no caso, o algoritmo principal) no ponto onde foi interrompida (comando de chamada da sub-rotina) e prossegue-se pela instrução imediatamente seguinte. Também é possível que uma sub-rotina chame outra através do mesmo mecanismo.

Essa invocação pode se dar também através dos parâmetros que são canais por onde os dados são transferidos pelo algoritmo chamador a uma sub-rotina, e vice-versa. Esta comunicação bidirecional pode ser feita de dois modos que são por meio de variáveis globais ou por meio da passagem de parâmetros.

As funções retornam um, e somente um, valor ao algoritmo chamador enquanto os procedimentos retornam zero (nenhum) ou mais valores ao algoritmo chamador. Na realidade, a tarefa desempenhada por uma sub-rotina do tipo função pode perfeitamente ser feita por outra do tipo procedimento (o primeiro é um caso particular deste).

Os Parâmetros formais são os nomes simbólicos introduzidos no cabeçalho de sub-rotinas, usados na definição dos parâmetros do mesmo. Dentro de uma sub-rotina trabalha-se com estes nomes da mesma forma como se trabalha com variáveis locais ou globais. Já os **Parâmetros reais** são aqueles que substituem os parâmetros formais quando da chamada de uma sub-rotina.

Na passagem de parâmetros por valor (ou por cópia) o parâmetro real é calculado e uma cópia de seu valor é fornecida ao parâmetro formal, no ato da invocação do sub-rotina. A execução do sub-rotina prossegue normalmente e todas as modificações feitas no parâmetro formal não afetam o parâmetro real Já no mecanismo de passagem de parâmetros por referencia, o espaço de memória ocupado pelos parâmetros reais é compartilhado pelos parâmetros formais correspondentes. Assim, as eventuais modificações feitas nos parâmetros formais também afetam os parâmetros reais correspondentes.

Atividades de aprendizagem

1. Qual a diferença entre função e procedimento?

2. O que são parâmetros formais e parâmetros reais?

3. Explique como ocorre o mecanismo de passagem de parâmetro por referencia.

4. Faça um algoritmo que leia um valor em segundos e mostre esse valor no formato de hora, minuto e segundo através da chamada de um procedimento.

5. Crie um algoritmo que faca a chamada a uma função que receba como parâmetro uma matriz A 6x6 e retorne o valor da soma de sua diagonal secundária.

6. Crie um algoritmo que faça a chamada a uma função que receba como parâmetros a altura e o sexo de uma pessoa e retorne o seu peso ideal. Para homens, deverá calcular o peso ideal usando a fórmula: peso ideal = 72.7* altura – 58. Para mulheres: peso ideal = 62.1*altura - 44.7

7. Crie um algoritmo que faça a chamada a uma função que receba dois parâmetros X e Z e calcule e retorne o valor de X elevado a Z (X^z) através de multiplicações sucessivas.

8. Crie um algoritmo que faça a chamada de uma função que receba um numero em decimal e retorne o numero convertido em binário.

9. Crie um algoritmo que faça a chamada de um procedimento que receba uma matriz quadrada e mostre os elementos acima da diagonal principal.

10. Crie um algoritmo que leia um vetor de nomes e faca a chamada de uma função que receba um nome de uma pessoa e retorne a posição deste nome no vetor caso o nome seja encontrado no vetor e o valor -, caso o nome não seja encontrado no vetor.

Referências

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi. **Fundamentos da Programação de Computadores.** São Paulo. Pearson Pretience Hall, 2007.

MEDINA, Marco; FERTIG, Cristina. **Algoritmos e Programação: Teoria e Prática**. São Paulo. Novatec Editora, 2006.

MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. **Algoritmos: Lógica Para Desenvolvimento de Programação de Computadores.** São Paulo. Érica, 2000.

LOPES, Anita; GARCIA, Guto. **Introdução à Programação.** Rio de Janeiro. Elsevier, 2002.

SAMPAIO, Silvio Costa. **Lógica para Programação.** Palmas. Editora Educon, 2008.

FARRER, Harry et al. **Algoritmos Estruturados.** Rio de Janeiro. Editora Guanabara Koogan S.A, 1989.

GOTTFRIED, Byron S. **Programação em Pascal.** Lisboa. McGraw Hill, 1994.

MECLER, Ian & MAIA, Luiz Paulo. **Programação e Lógica com Turbo Pascal.** Rio de Janeiro, Campus, 1989.

ORTH, Afonso Inácio. **Algoritmos.** Porto Alegre. Editora Pallotti, 1985. 130p.

SALIBA, Walter Luís Caram. **Técnicas de Programação: Uma Abordagem Estruturada.** São Paulo. Makron, McGraw-Hill, 1992.

rede e-Tec Brasil

ISBN 978-85-67082-05-9