Licenciatura

Computação

Algoritmos e Programação I

Ivo Mario Mathias

UNIVERSIDADE ABERTA DO BRASIL

UAB

EDUCAÇÃO A DISTÂNCIA

EaD em rede

ABRUEM
Associação Brasileira dos Reitores das Universidades Estaduais e Municipais

EDUCAÇÃO A DISTÂNCIA

LICENCIATURA EM

Computação

ALGORITMOS E PROGRAMAÇÃO I

Ivo Mario Mathias

PONTA GROSSA / PARANÁ
2017

CRÉDITOS

Universidade Estadual de Ponta Grossa

Carlos Luciano Sant'ana Vargas
Reitor

Gisele Alves de Sá Quimelli
Vice - Reitor

Pró-Reitoria de Assuntos Administrativos
Amaury dos Martyres - Pró-Reitor

Pró-Reitoria de Graduação
Miguel Archanjo de Freitas Junior - Pró-Reitor

Núcleo de Tecnologia e Educação Aberta e a Distância
Eliane de Fátima Rauski- Coordenadora Geral
Marli de Fátima Rodrigues - Coordenadora Pedagógica

Sistema Universidade Aberta do Brasil
Eliane de Fátima Rauski - Coordenadora Geral
Marli de Fátima Rodrigues - Coordenadora Adjunta
Marcelo Ferrasa - Coordenador de Curso

Colaboradores em EAD
Dênia Falcão de Bittencourt
Cláudia Cristina Muller

Projeto Gráfico
Eloise Guenther

Colaboradores de Publicação
Denise Galdino - Revisão
Eloise Guenther - Diagramação

Ficha catalográfica elaborada pelo Setor Tratamento da Informação BICEN/UEPG

Mathias, Ivo Mario
M431a Algoritmos e programação I/ Ivo Mario Mathias. Ponta Grossa: UEPG/ NUTEAD, 2017.
175p.; il.

ISBN: 978.85.8024.298.0

Curso de Licenciatura em Computação. Universidade Estadual de Ponta Grossa.

1. Linguagem de programação. 2. Algoritmos - representação. 3. Instruções primitivas. 4. Tipos de dados. 5. Variáveis. 6. Expressões 7.Pascalzim. 8. Fluxo de execução. I. T.

CDD: 005.113

UNIVERSIDADE ESTADUAL DE PONTA GROSSA
Núcleo de Tecnologia e Educação Aberta e a Distância - NUTEAD
Av. Gal. Carlos Cavalcanti, 4748 - CEP 84030-900 - Ponta Grossa - PR
Tel.: (42) 3220-3163
www.nutead.org
2017

APRESENTAÇÃO INSTITUCIONAL

A Universidade Estadual de Ponta Grossa é uma instituição de ensino superior estadual, democrática, pública e gratuita, que tem por missão responder aos desafios contemporâneos, articulando o global com o local, a qualidade científica e tecnológica com a qualidade social e cumprindo, assim, o seu compromisso com a produção e difusão do conhecimento, com a educação dos cidadãos e com o progresso da coletividade.

No contexto do ensino superior brasileiro, a UEPG se destaca tanto nas atividades de ensino, como na pesquisa e na extensão. Seus cursos de graduação presenciais primam pela qualidade, como comprovam os resultados do ENADE, exame nacional que avalia o desempenho dos acadêmicos e a situa entre as melhores instituições do país.

A trajetória de sucesso, iniciada há mais de 40 anos, permitiu que a UEPG se aventurasse também na educação a distância, modalidade implantada na instituição no ano de 2000 e que, crescendo rapidamente, vem conquistando uma posição de destaque no cenário nacional.

Atualmente, a UEPG é parceira do MEC/CAPES/FNDE na execução dos programas de Pró-Licenciatura e do Sistema Universidade Aberta do Brasil e atua em 40 polos de apoio presencial, ofertando, diversos cursos de graduação, extensão e pós-graduação a distância nos estados do Paraná, Santa Cantarina e São Paulo.

Desse modo, a UEPG se coloca numa posição de vanguarda, assumindo uma proposta educacional democratizante e qualitativamente diferenciada e se afirmando definitivamente no domínio e disseminação das tecnologias da informação e da comunicação.

Os nossos cursos e programas a distância apresentam a mesma carga horária e o mesmo currículo dos cursos presenciais, mas se utilizam de metodologias, mídias e materiais próprios da EaD que, além de serem mais flexíveis e facilitarem o aprendizado, permitem constante interação entre alunos, tutores, professores e coordenação.

Esperamos que você aproveite todos os recursos que oferecemos para promover a sua aprendizagem e que tenha muito sucesso no curso que está realizando.

A Coordenação

SUMÁRIO

PALAVRAS DO PROFESSOR

Você está iniciando a disciplina de Algoritmos e Programação I do curso de Licenciatura em Computação. O objetivo principal da disciplina é proporcionar um estudo sobre algoritmos e programação de computadores, possibilitando o desenvolvimento do raciocínio lógico aplicado na solução de problemas em nível computacional.

Além de introduzir os conceitos básicos de desenvolvimento de algoritmos, a abordagem a ser utilizada na disciplina busca dotá-lo da capacidade de construção de algoritmos em pseudolinguagem, que modelem as possíveis soluções e a concretização desses algoritmos na linguagem de programação Pascal, bem como familiarizar-se com a nomenclatura e notações da linguagem.

Resumidamente, a disciplina ensina você como visualizar soluções computacionais para problemas em geral e ser capaz de aplicar, de modo prático, os conceitos da lógica de programação.

Como a disciplina possui uma abordagem teórica e prática a leitura deste livro é de suma importância, pois dificilmente pode-se entender e implementar uma solução prática sem que a teórica seja entendida. Sendo assim, estude por ele, procure assimilar as definições, acompanhar os exemplos e fazer todos os exercícios propostos de modo sequencial e que nenhuma atividade ou exercício não seja realizado ou entendido.

Assuma os estudos com seriedade e responsabilidade, pois a sua dedicação será recompensada com conhecimento.

Bons Estudos!

OBJETIVOS E EMENTA

OBJETIVOS

Objetivo geral:

O objetivo geral da disciplina de Algoritmos e Programação I é proporcionar o estudo de algoritmos, possibilitando que o aluno desenvolva o raciocínio lógico aplicado à solução de problemas em nível computacional.

Objetivos específicos:

- Introduzir os conceitos básicos e definições sobre Algoritmo e Programação.
- Saber classificar as linguagens de programação de computadores.
- Entender o que são compiladores e interpretadores.
- Compreender como o software de computador é classificado.
- Compreender os conceitos básicos de desenvolvimento de algoritmos.
- Aprender a escrever algoritmos.
- Entender o fluxo de execução de um programa de computador.
- Saber quais são as instruções primitivas de um algoritmo.
- Conhecer os tipos de dados que são declarados em um algoritmo.
- Desenvolver o raciocínio lógico e abstrato de programação.
- Implementar os algoritmos em programas na Linguagem de Programação Pascal.
- Visualizar soluções computacionais para problemas.
- Praticar o processo de desenvolvimento de algoritmos.

EMENTA

Desenvolvimento de algoritmos estruturados. Tipos de dados. Expressões. Estruturas de controle: sequencial, condicional e repetição. Ambientes de programação. Aplicação de algoritmos em uma linguagem de programação.

UNIDADE I

Algoritmo e Programação

Objetivos de aprendizagem

- Introduzir os conceitos básicos e definições sobre Algoritmo e Programação.

Roteiro de estudos

- SEÇÃO 1 – Algoritmo
- SEÇÃO 2 – Programação

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Nesta unidade você tomará conhecimento das definições e conceitos básicos sobre Algoritmos e a relação abstrata deles com a concretização em Programas de Computador.

SEÇÃO 1
ALGORITMO

É a descrição de um conjunto de **ações** que, obedecidas, resultam numa sucessão finita de passos atingindo um objetivo.

Como nessa definição preliminar diz-se que os algoritmos descrevem um conjunto de **ações**, é necessário também definir o que é uma ação.

Ação

É um acontecimento que a partir de um estado inicial, após um período de tempo finito, produz um estado final previsível e bem definido.

Diante desta definição e também pelo fato de que a visão que se busca do que é um algoritmo é uma visão computacional, em que geralmente tem-se como objetivo um processo de automação, pode-se dizer que: **Automação** é o processo em que uma tarefa deixa de ser desempenhada pelo homem e passa a ser realizada por máquinas, sejam estas, dispositivos mecânicos, eletrônicos (como computadores) ou de natureza mista.

Dando sequência nesse raciocínio teórico, pode-se deduzir que quando um processo é automatizado ele também deve possuir características de repetibilidade. Neste contexto, é necessário que seja especificado com clareza e exatidão o que deve ser realizado em cada

uma das fases do processo a ser automatizado, bem como a sequência em que estas fases devem ser realizadas.

A partir das considerações anteriores é possível mesclar essas ideias em definições complementares do que são algoritmos.

- **Algoritmo –** é a especificação de uma sequência ordenada de passos que deve ser seguida para realização de uma tarefa garantindo a sua repetibilidade.
- **Algoritmo –** é uma sequência de passos que visam atingir um objetivo bem definido.

Resumidamente, quando se fala em algoritmo, algumas palavras jamais devem ser esquecidas: sequência, ordem, passos, objetivo e repetibilidade.

Pode-se perceber que as abordagens a respeito do que são algoritmos ainda são abstratas do ponto de vista computacional. A construção do algoritmo é abstrata, até que ele seja codificado em um programa por meio de uma linguagem de programação, onde se pode testá-lo e concretizá-lo em um processo computacional, em que a partir da sua utilização passa a ter as características de repetibilidade.

A definição de programação a seguir, possibilita o entendimento do que são programas de computador.

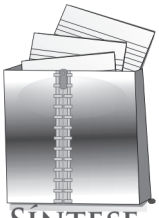
SEÇÃO 2
PROGRAMAÇÃO

Com segurança, pode-se afirmar que o conceito central da programação é o de algoritmo, pois programar é basicamente construir algoritmos.

Um programa de computador, também conhecido como *software* (do inglês, programas), pode ser visto como uma codificação de um algoritmo em uma determinada Linguagem de Programação, para execução pelo computador, ou ainda, um programa é uma sequência de instruções/

comandos para serem executados por um computador, com a finalidade de executar tarefas específicas.

Segundo Niklaus Wirth, programas de computador são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados. Desta última definição, pode-se destacar um item novo e importante nesse contexto: "representações e estruturas específicas de dados", que será abordado posteriormente com profundidade, pois faz parte da essência da concretização de um algoritmo em programa. Porém, antes disso, é necessário entender como são classificadas as linguagens de programação e como elas fazem com que os computadores funcionem.



SÍNTESE

Nesta breve unidade, você teve a oportunidade de conhecer a definição do que é um algoritmo e como ele está intimamente ligado a um programa de computador. Foi evidenciado o fato de que quando um programa de computador é desenvolvido, tem a finalidade de automatizar um processo para que haja repetibilidade.

Algumas palavras foram destacadas no que se refere a construção de algoritmos e programas: sequência, ordem, passos, objetivo e repetibilidade.

A próxima unidade vai abordar os detalhes que você precisa saber em relação às linguagens de programação de computadores.

ANOTAÇÕES

UNIDADE II

Linguagens de Programação

Objetivos de aprendizagem

- Saber classificar as linguagens de programação de computadores.
- Entender o que são compiladores e interpretadores.
- Compreender como o software de computador é classificado.

Roteiro de estudos

- SEÇÃO 1 – Linguagem de Máquina – primeira geração –1GL
- SEÇÃO 2 – Linguagem Assembly /Montagem – segunda geração – 2GL
- SEÇÃO 3 – Linguagens de Alto Nível – terceira geração – 3GL
- SEÇÃO 4 – Linguagens de quarta geração – 4GL
- SEÇÃO 5 – Compiladores
- SEÇÃO 6 – Interpretadores
- SEÇÃO 7 – Software

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Pode-se afirmar que sem linguagens de programação não há programas de computador, mas para entender essa situação, inicialmente deve-se classificá-las por meio de gerações, relacionadas com a evolução rápida que os computadores sofreram com o passar dos anos.

SEÇÃO 1
LINGUAGEM DE MÁQUINA - PRIMEIRA GERAÇÃO - 1GL

Quando os computadores executam os programas eles o fazem por meio da linguagem de máquina, que corresponde a um conjunto de instruções, também chamadas código de máquina, que ativam diretamente os dispositivos eletrônicos do computador.

Periférico corresponde a placas ou aparelhos que recebem ou transmitem informações para o computador, por exemplo, impressoras, mouses, teclados, câmeras, etc.

Os computadores possuem diversos comandos implementados em seu processador, tais como: operações matemáticas, transferência de dados entre seus periféricos, e esses comandos são acessados em forma numérica (binária ou hexadecimal), situação que dificulta o trabalho dos programadores, pois é necessário saber com exatidão qual instrução corresponde cada número. Diante das dificuldades de interpretação dos códigos apresentadas pela linguagem de máquina surgiu a ideia da linguagem simbólica, denominada *Assembly*, considerada a segunda geração.

SEÇÃO 2
LINGUAGEM ASSEMBLY / MONTAGEM - SEGUNDA GERAÇÃO - 2GL

É a linguagem em que os códigos numéricos são representados por palavras (mnemônicos) como *LOAD, STORE, MOVE, COMPARE, ADD, SUBTRACT, MULTIPLY e DIVIDE*, as quais são usadas para programar os dispositivos computacionais como microprocessadores e microcontroladores. Essa situação veio facilitar o trabalho dos programadores, pois a palavra/símbolo expressa qual a operação que aquela instrução pode executar.

SAIBA MAIS

Mnemônico é um conjunto de técnicas utilizadas para auxiliar o processo de memorização. Consiste na elaboração de suportes como os esquemas, gráficos, símbolos, palavras ou frases relacionadas com o assunto que se pretende memorizar.

O *Assembly* é uma linguagem de baixo nível, pelo fato de estar mais próxima da forma com que o computador representa os dados e instruções e geralmente, cada comando executa apenas uma instrução. A vantagem deste tipo de linguagem é a velocidade de execução dos programas e o tamanho deles, que são mais compactos.

A linguagem *Assembly* é vinculada à arquitetura da CPU (*Central Processing Unit*) ou em português UCP (Unidade Central de Processamento) do computador. Para que um programa escrito em linguagem *Assembly* possa ser executado ele precisa ser traduzido para a linguagem de máquina, que é realizado por um programa chamado *Montador* ou *Assembler*, escrito em linguagem de máquina, o qual organiza as palavras e as traduz para forma binária, criando assim o *código-objeto*.

Fique atento a este termo: *código-objeto* ou programa-objeto, é importante saber o que são e que eles se originam de um *código-fonte* ou *programa-fonte*. Ambos os itens serão muito utilizados no decorrer da disciplina.

Portanto, *código-objeto* é um arquivo gerado a partir do *código-fonte*, em que as instruções e comandos são convertidos para linguagem de máquina. Pode-se deduzir então que, neste contexto, o *código-fonte* é o arquivo que contém as instruções e os comandos da linguagem *Assembly*. Porém, o termo *código-fonte*, também é utilizado para designar o arquivo, no formato texto, que contêm as instruções e comandos de outras linguagens de programação, como exemplo: Pascal, Basic, Java, C e outras, que são chamadas de linguagens de alto nível, assunto a ser abordado na próxima seção.

SEÇÃO 3
LINGUAGENS DE ALTO NÍVEL - TERCEIRA GERAÇÃO - 3GL

A partir da linguagem *Assembly*, onde um comando desta corresponde a um comando em linguagem de máquina, se estabeleceu a relação de um comando simbólico para vários comandos em linguagem de máquina. As linguagens de programação com essa característica são consideradas de alto nível, ou seja, possuem um nível de abstração mais elevado, distante do código de máquina e mais próximo da linguagem humana.

Linguagens de alto nível são linguagens de programação projetadas para serem facilmente entendidas pelo ser humano, consideradas procedurais e estruturadas, em que a maioria delas suporta o conceito de programação estruturada. Veja a seguir, algumas das linguagens de programação de terceira geração mais utilizadas.

- 1957 – FORTRAN (*Formula Translation*) – finalidade principal, desenvolvimento de fórmulas matemáticas – de natureza técnica e científica. Teve o seu auge de utilização entre as décadas de 80 e 90, atualmente pouco difundida.

- 1959 – COBOL (*Common Business Oriented Language*) – de uso geral. Entre as décadas de 80 e 90 foi bastante utilizada para o desenvolvimento de sistemas comerciais de grandes corporações, apesar de antigos, ainda existem sistemas funcionando com esta linguagem até hoje.
- 1964 – BASIC (*Begineers All-Purpose Symbolic Instruction Code*) – de uso geral e com o objetivo inicial de ser uma linguagem mais simples que o Fortran, porém evoluiu sem um padrão definido e passou a ser a linguagem mais utilizada em microcomputadores, de uso geral. Há versões em uso atualmente, como o Visual Basic.
- 1968 – PASCAL – voltada para o ensino, mas também de uso geral. Em uso atualmente, como o caso do Pascalzim, Lazarus (Linux), Delphi e outros.
- 1977 – C (A,B e C ultimo resultado) – uso geral, nível médio. Em uso atualmente em vários ambientes, como C++ Builder, Netbeans, Dev C++ e outros.

Um modo de facilitar o entendimento operacional entre as três gerações de linguagens de programação, citadas anteriormente, é um comparativo dos códigos utilizados em cada caso, a Figura 1 exemplifica uma operação simples de adição em cada geração.

Linguagem de máquina	Linguagem de baixo nível	Linguagem de alto nível
0010 0001 1110	LOAD R1, val1	val2 = val1 + val2
0010 0010 1111	LOAD R2, val2	
0001 0001 0010	ADD, R1, R2	
0011 0001 1111	STORE R1, val2	

Figura 1: comparativo entre códigos de linguagens de programação – 1GL, 2GL e 3GL.

Observa-se, no caso da linguagem de máquina, que as instruções são números binários, enquanto na linguagem de baixo nível, são instruções em *Assembly* e variáveis, uma para cada código binário da linguagem de máquina e, na Linguagem de Alto Nível são apenas variáveis com os operadores de atribuição e de adição.

Ainda focando o processo evolutivo da informática e mais precisamente das linguagens de programação, existem ainda duas outras

gerações de linguagens, que são descritas a seguir. Apesar de não haver um consenso das características das quartas e quintas gerações de linguagens, foram escolhidas aquelas em que a maioria dos autores aborda.

SEÇÃO 4
LINGUAGENS DE QUARTA GERAÇÃO - 4GL

Segundo a literatura, uma das principais características dessas linguagens é o fato delas terem a capacidade de gerar código automaticamente para programas de computador, aplicando a técnica denominada RAD (*Rapid Application Development*) ou Desenvolvimento Rápido de Aplicação (em português). O uso dessa técnica possibilita uma maior otimização e velocidade no desenvolvimento dos códigos-fonte dos programas, por meio de uma IDE (*Integrated Development Environment*) ou Ambiente Integrado de Desenvolvimento (em português). Geralmente esses ambientes de desenvolvimento possuem uma interface utilizando-se dos recursos gráficos do sistema operacional como o Windows e Linux, alguns exemplos de linguagens com estas características são: C Builder, Delphi e Visual Basic.

Na quarta geração de linguagens, também se enquadram as linguagens orientadas a objetos, que possuem a propriedade da reutilização de partes do código-fonte para serem usadas em outros programas.

Embora seja um assunto que não será estudado nessa disciplina é importante para o aluno ter conhecimento do que seja programação orientada a objetos, pois em muitas publicações relacionadas a algoritmos e programação poderá se deparar com esse termo. Sendo assim, aqui apenas o conceito principal e uma definição para um entendimento teórico do assunto.

O conceito central da programação orientada a objetos está relacionado com a ideia de classificar, organizar e abstrair coisas, e uma

definição seria:, *A orientação a objetos significa organizar o mundo real como uma coleção de objetos que incorporam estrutura de dados e um conjunto de operações que manipulam estes dados*.

É interessante exemplificar algumas linguagens de 4GL, que atualmente estão em uso na maioria dos sistemas computacionais, isso inclui sistemas comerciais, aplicativos para internet e sistemas operacionais.

- SQL (*Structured Query Language*) – é uma linguagem padrão para manipulação e consulta de bancos de dados relacionais, também usada em conjunto com as linguagens de 3GL.
- Java – é uma linguagem de programação orientada a objeto, desenvolvida na década de 90 por uma equipe de programadores chefiada por James Gosling, na empresa Sun Microsystems. Diferentemente das linguagens convencionais, que são compiladas para código nativo, a linguagem Java é compilada para um *bytecode,* que é executado por uma máquina virtual.
- PHP (*Personal Home Page*) – é uma linguagem de programação de computadores interpretada, de uso livre e muito utilizada para gerar conteúdo dinâmico na Web, desenvolvida em meados de 1994.

Se você fez uma leitura atenta dos últimos tópicos, deve ter percebido que surgiram alguns termos novos que podem gerar indagações, como, *compiladas para código nativo, linguagem de programação de computadores interpretada.* Assuntos relacionados aos termos citados anteriormente, serão abordados em seguida como: *Compiladores* e *Interpretadores*.

Os bancos de dados relacionais são arquivos em que os dados são armazenados em tabelas. Tabelas são organizadas em colunas e cada coluna armazena um tipo de dados (inteiro, números reais, strings de caracteres, data, etc.). Os dados de uma simples *instância* de uma tabela são armazenados como uma linha.

Bytecode (do inglês, código em bytes) é o resultado de um processo semelhante ao dos compiladores de código-fonte que não é imediatamente executável, mas interpretado por uma máquina virtual para ser executado.

SEÇÃO 5
COMPILADORES

São programas de computador especialmente projetados para traduzir programas escritos em linguagem de alto nível (programa-fonte) para linguagem de máquina, gerando o programa-objeto. Após esta operação, o código-objeto deve passar por mais uma etapa que é a *linkedição*, a qual consiste em gerar o programa executável (aplicativo). O programa *linkeditor* tem também a finalidade de juntar diversos programas-objeto em um único programa executável. O compilador é específico a cada linguagem de programação, ou seja, só pode compilar programas escritos na linguagem para a qual foi projetado e, ao efetuar a compilação detecta e emite mensagens sobre os erros no uso da linguagem (erros de sintaxe, erros de semântica – uso incorreto de comandos), não gerando o programa compilado, ou seja, o programa-objeto.

Durante as atividades práticas da disciplina, você se familiarizará com a compilação dos programas quando estiver utilizando o ambiente de programação para a linguagem Pascal.

SEÇÃO 6
INTERPRETADORES

Também são programas especialmente projetados para traduzir (interpretar) programas escritos em linguagem de alto nível para linguagem de máquina, durante a execução, ou seja, não é gerado um programa executável, o próprio texto do programa é utilizado. O interpretador é específico a cada linguagem de programação e durante a execução de um programa verifica a existência de algum erro no uso da

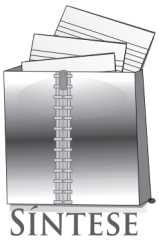
linguagem, caso haja, interrompe a execução emitindo uma mensagem apontando o erro. Pelo fato da interpretação ocorrer em duas etapas, interpretação e execução, o funcionamento de um programa interpretado é mais lento em relação a um compilado, que já se encontra em código de máquina.

Após essa introdução sobre Algoritmos e Programação, para finalizar esta unidade e consolidar esses conhecimentos é importante conhecer uma classificação dos sistemas computacionais, software, que fazem parte dos computadores.

SEÇÃO 7
SOFTWARE

É uma classificação genérica, sem aprofundamento em cada classe, mas esse conhecimento é importante para entender parte do funcionamento dos computadores.

- **Software de Sistema ou de Base** – são os programas essenciais para o funcionamento de um computador, pois sem eles o computador não funciona. Subdivide-se em **Software de Sistemas Operacionais** (exemplos: Windows e Linux) e **Software de Linguagens de Programação** (Pascal, C, Java, SQL, etc.).
- **Software de Aplicação, Aplicativo ou Utilitário** – são programas que não são essenciais ao funcionamento do computador, porém, essenciais aos usuários no desenvolvimento de suas tarefas com o computador, exemplos: editores de texto, planilhas eletrônicas, navegadores para internet, sistemas comerciais, antivírus e diversos outros.



SÍNTESE

Nesta unidade, você tomou conhecimento como as Linguagens de Programação de Computadores são classificadas e associada a esta classificação, o processo evolutivo delas, sendo: Linguagem de Máquina – primeira geração–1GL, Linguagem Assembly /Montagem – segunda geração – 2GL, Linguagens de Alto Nível – terceira geração – 3GL, Linguagens de quarta geração – 4GL.

Outro assunto importante foi saber o que são Compiladores e Interpretadores, lembrando que ambos são programas de computador, que têm por finalidade traduzir os programas-fonte em linguagem de máquina. A diferença entre eles é que, ao final de um processo de compilação, se obtém um programa executável, que nada mais é que um aplicativo. Já no processo de interpretação, há também a tradução para a linguagem de máquina, porém, o aplicativo não é gerado, e o código-fonte é interpretado e executado simultaneamente. Lembrando que este processo é mais lento na execução se comparado a um programa executável, compilado.

Para você entender como os sistemas computacionais, software, são classificados, foi descrita em uma classificação genérica: Software de Sistema ou de Base que se divide em Software de Sistemas Operacionais (exemplos: Windows e Linux) e Software de Linguagens de Programação (Pascal, C, Java, SQL, etc.); e o Software de Aplicação, Aplicativo ou Utilitário.

Uma percepção que deve ser consolidada neste momento da disciplina é o fato de que todo e qualquer computador para funcionar deve estar munido de programas, ou seja, toda e qualquer operação que o computador realiza é mediante instruções que os programas desencadeiam. Por mais simples que seja a ação, por exemplo, o simples fato de pressionar a tecla da letra ‘a’ do teclado, e ser exibida na tela do monitor de vídeo do computador, requer instruções de programas.

ANOTAÇÕES

UNIDADE III

Formas de representação de algoritmos

Objetivos de aprendizagem

- Compreender os conceitos básicos de desenvolvimento de algoritmos.
- Aprender a escrever algoritmos.

Roteiro de estudos

- SEÇÃO 1 – Descrição Narrativa
- SEÇÃO 2 – Fluxograma Convencional
- SEÇÃO 3 – Pseudocódigo - Linguagem Estruturada - Portugol

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Agora você começará a aprender como os algoritmos podem ser representados e/ou escritos, para posteriormente serem codificados em uma linguagem de programação e finalmente se transformarem em um programa/software aplicativo. Os estudos nortearão três formas, a saber: descrição narrativa, o fluxograma e o pseudocódigo.

SEÇÃO 1
DESCRIÇÃO NARRATIVA

É uma representação dos algoritmos diretamente em linguagem natural, ou seja, é como narrar o problema a ser resolvido, evidentemente buscando um entendimento de como algo pode ser resolvido computacionalmente. A descrição narrativa, apesar de pouco usada na prática, pois o uso da linguagem natural muitas vezes causa má interpretação, ambiguidades e imprecisões, pode ser útil quando não se consegue abstrair uma solução computacional para o problema apresentado. Não há um padrão, mas o exemplo a seguir, ajuda a entender como essa forma de representar um algoritmo pode ser usada, considerando que o objetivo do programa seja efetuar a soma entre dois números e exibir o resultado.

Descrição narrativa:

1. informar um número;
2. informar outro número;
3. efetuar a soma entre os dois números informados;
4. exibir o resultado da soma.

Observando a descrição narrativa, exibida anteriormente, é possível entender o problema a ser resolvido, porém gera algumas dúvidas devido à imprecisão das informações, tais como:

- De que forma os valores serão representados?
- Serão valores inteiros ou fracionários?
- Como e qual o dispositivo será usado para informar os valores de entrada?
- Como e qual dispositivo de saída será usado para exibir o resultado?

Apesar das dúvidas, percebe-se que mesmo quem não conhece algoritmos teria condições de explicar a problemática em si, ou seja, contribui para o entendimento e auxilia para passar para outra fase da elaboração do algoritmo, que poderia ser, por meio de imagens, representações geométricas.

SEÇÃO 2
FLUXOGRAMA CONVENCIONAL

Trata-se de uma representação gráfica de algoritmos, em que formas geométricas diferentes implicam ações (instruções, comandos) distintas. É uma forma intermediária entre a descrição narrativa e o pseudocódigo (item seguinte), mas é menos imprecisa que a primeira, no entanto, não se preocupa com detalhes de implementação do programa, como: tipo das variáveis usadas (item futuro). Preocupa-se com detalhes de nível físico da implementação do algoritmo, como: distinguir dispositivos onde ocorrem as operações de entrada e saída de dados. A figura 2, exemplifica as principais formas geométricas usadas em fluxogramas.

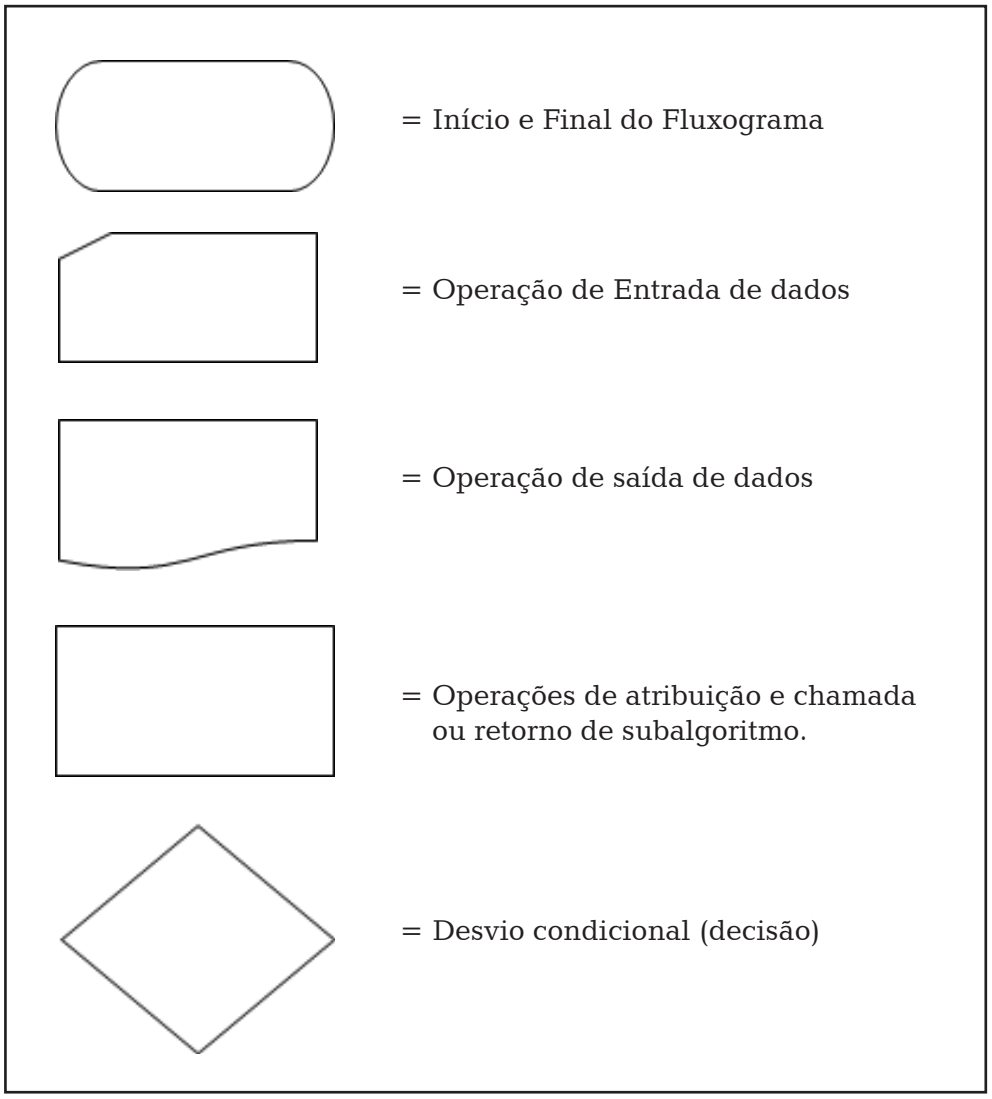


Figura 2: Fluxograma convencional - principais formas

EXEMPLO

Observe na figura 3, um exemplo de um algoritmo representado sob a forma de fluxograma em que o objetivo é calcular a soma de dois números.

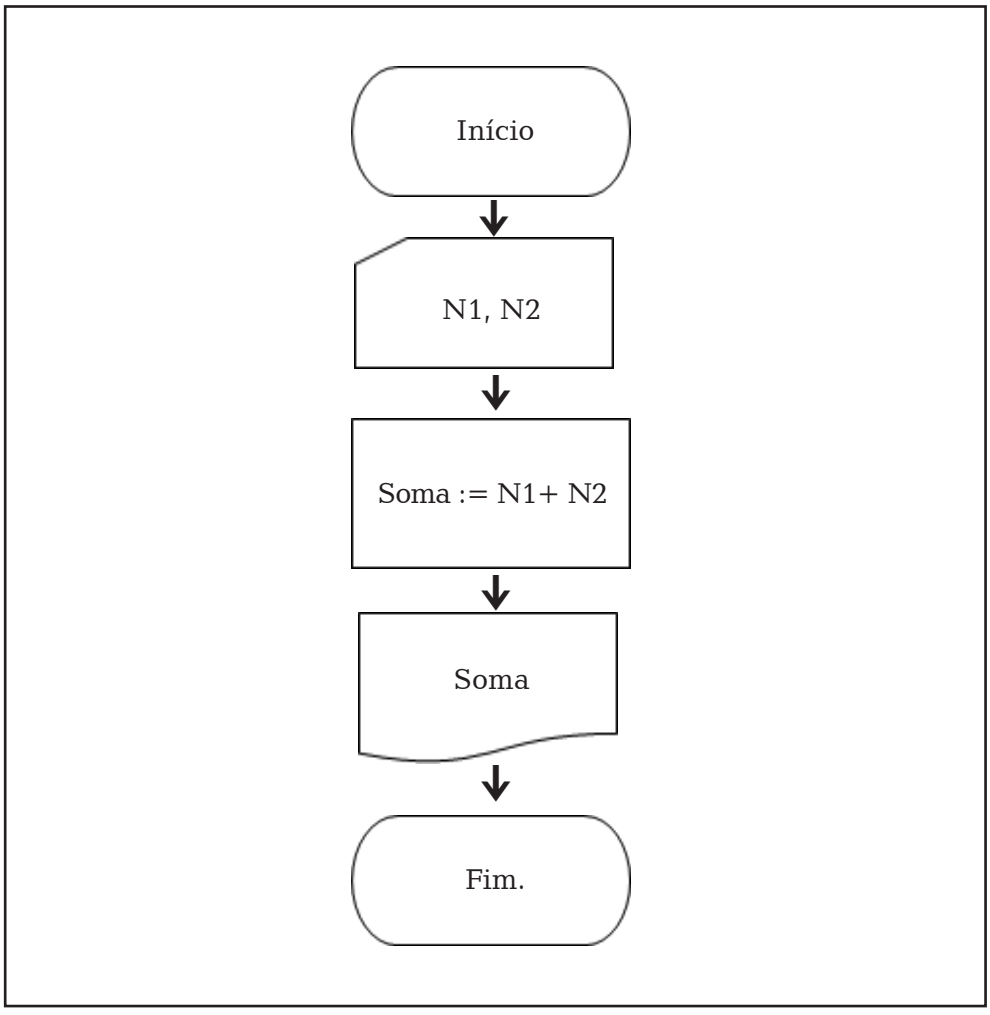


Figura 3: Cálculo da soma de dois números sob a forma de um fluxograma.

No fluxograma ilustrado anteriormente, é notório uma maior facilidade de entendimento em relação à descrição narrativa, porém ainda faltam alguns detalhes computacionais, principalmente no que se refere a representação específica dos dados que é suprida pelo pseudocódigo.

SEÇÃO 3
PSEUDOCÓDIGO - LINGUAGEM ESTRUTURADA - PORTUGOL

É a forma de representar algoritmos, possuindo todos os requisitos que as linguagens de programação necessitam, tais como, as declarações das variáveis com os respectivos tipos e as instruções e comandos similares aos usados em linguagens de programação. Resumidamente a escrita dos algoritmos é equivalente a forma como os programas de computador são expressos.

O pseudocódigo também é chamado de linguagem estruturada, pelo fato que deve seguir uma estrutura, ou seja, um padrão, o qual possui um formato semelhante ao das linguagens de programação, facilitando a codificação para qualquer linguagem que se deseje.

Visto isto, percebe-se que a proposta do algoritmo é possuir um formato genérico, não específico a uma determinada linguagem de programação. O formato de algoritmo adotado neste livro será o *portugol*, como o próprio termo expressa, a sua origem é da linguagem portuguesa.

A seguir, na figura 4, está ilustrada uma forma geral de representação de um algoritmo em pseudocódigo.

```
Algoritmo < nome_do_algoritmo >;
    Var
       < declaração_de_variáveis >;
    Início
       < corpo_do_algoritmo >;
    Fim.
```

Figura 4: Forma geral de representação de um algoritmo em pseudocódigo.

É fundamental que você analise a figura 4, leia atentamente os comentários a seguir e memorize a estrutura exibida.

- **Algoritmo** – é uma palavra que indica o início da definição de um algoritmo em forma de pseudocódigo;
- **<**nome_do_algoritmo**>** – é um nome simbólico dado ao algoritmo com a finalidade de distingui-lo dos demais;
- **<**declaração_de_variáveis**>** – consiste em uma porção opcional onde são declaradas as variáveis usadas no algoritmo;
- **Início** e **Fim** – são as palavras que, respectivamente, delimitam o início e término do conjunto de instruções do corpo do algoritmo.

A partir de agora, os estudos serão voltados aos mínimos detalhes do algoritmo e é importante relembrar, para que não fiquem dúvidas em nenhuma das Unidades e Seções, tanto as que já foram estudadas, e principalmente as que estão por vir. Caso tenham ficado dúvidas, reestude os assuntos pendentes. Esse alerta é necessário, pois você vai perceber que para escrever um algoritmo e ele ser implementado como programa, deve estar totalmente alinhado com detalhes das estruturas de programação, que serão estudadas.

Siga em frente!

Um detalhe que possivelmente você percebeu, na forma geral de representar um algoritmo, que as palavras **Algoritmo**, **Início**, **Fim** e **Var** estão em **negrito**, essas palavras correspondem as **palavras-reservadas**.

As **palavras-reservadas** em programação de computadores tanto na questão algorítmica como das linguagens de programação, tem um significado especial relacionado ao que ela representa naquele contexto, e, em razão dessa situação, não pode ser usada pelo programador para outra função, porque ela expressa uma instrução, comando ou identificador da linguagem e o uso indevido dela pode gerar um conflito. Por padronização, nos algoritmos as **palavras-reservadas** estarão destacadas em **negrito**.

Durante os estudos, gradativamente você vai aprender o que representam cada uma das palavras-reservadas, tanto do portugol como da linguagem de programação Pascal.

Agora você pode observar o primeiro algoritmo escrito em portugol, figura 5, do mesmo exemplo já abordado anteriormente em fluxograma, na figura 3.

```
Algoritmo Soma_dois_Numeros;

    Var
        N1, N2, Soma : real;

    Início
        Leia N1, N2;
        Soma := N1 + N2;
        Escreva Soma;
    Fim.
```

Figura 5: Cálculo da soma de dois números em portugol.

Aproveitando o exemplo em questão, é essencial entender como geralmente os programas de computador funcionam. Esse entendimento é facilitado pelo algoritmo que foi representado em fluxograma, figura 3, pois nas figuras é visível a sequência de passos que devem ser seguidas. Ao comparar com a representação em portugol, também é perceptível uma sequência a ser seguida, porém é oportuno simplificar este funcionamento pelo esquema [**Entrada → Processamento → Saída**], ou seja:

Entrada – dados que são enviados ao computador, por meio de seus dispositivos de entrada, tais como: teclado, mouse, tela do monitor de vídeo sensível ao toque, discos, pendrive, etc;

Processamento – a UCP reconhece os dados que foram transmitidos pelos dispositivos de entrada e efetua o processamento deles conforme as diretivas do programa;

Saída – após o processamento dos dados e também conforme as diretivas do programa, os dispositivos de saída são acionados, por exemplo: tela do monitor de vídeo, impressora, discos, pendrive, etc.

Visando uniformizar o processo [**Entrada → Processamento → Saída**] uma vez que, o objetivo principal desta disciplina são os detalhes a respeito da construção de algoritmos e programas, o dispositivo de entrada padrão sempre será o **teclado** e o de saída, a **tela do monitor de vídeo**.

O intuito agora é reconhecer no algoritmo, quais são as diretivas que definem os passos de [**Entrada → Processamento → Saída**], sendo que isso pode ser identificado pelas instruções primitivas, próxima unidade.


ANOTAÇÕES

UNIDADE IV

Instruções Primitivas

Objetivos de aprendizagem

- Entender o fluxo de execução de um programa de computador.
- Saber quais são as instruções primitivas de um algoritmo.

Roteiro de estudos

- SEÇÃO 1 – Instrução Primitiva de Entrada de Dados
- SEÇÃO 2 – Instrução Primitiva de Atribuição
- SEÇÃO 3 – Instrução Primitiva de Saída de Dados

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

As instruções primitivas são os comandos básicos que efetuam tarefas essenciais para a operação dos computadores, como entrada e saída de dados e movimentação dos mesmos na memória, comunicação com o usuário e com os dispositivos periféricos.

Neste contexto, todas as palavras que forem utilizadas como instruções primitivas (comandos/instruções) são consideradas **palavras-reservadas** e não poderão ser utilizadas como nome de variável ou outro identificador, observando que esse conceito também ocorre nas linguagens de programação.

Antes de efetivamente serem detalhadas as instruções primitivas é relevante destacar o fato de que as linguagens de programação, bem como os algoritmos em pseudocódigo possuem regras de escrita, assim como a gramática da língua portuguesa que orienta como as palavras podem ser combinadas ou modificadas, para que as pessoas possam comunicar-se. Neste sentido, em algoritmos e programação de computadores usa-se o termo **sintaxe** – que corresponde a forma como os comandos/instruções devem ser escritos, a fim de que possam ser entendidos pelos compiladores/interpretadores de programas. É fundamental saber que no uso das linguagens de programação o não cumprimento das regras sintáticas impede que um programa seja compilado em sua totalidade e, por consequência, não é possível executá-lo.

Outro termo relacionado à gramática das linguagens de programação é a **semântica** – que está relacionada ao significado, ou seja, o conjunto de ações que serão exercidas pelo computador durante a execução de um determinado comando. Uma violação das regras semânticas é similar a uma frase em linguagem natural, que por ventura esteja correta gramaticalmente, mas não faz o menor sentido, ou seja, é incompreensível.

Gradativamente, durante os estudos, principalmente na elaboração dos exercícios práticos, você terá oportunidade de conhecer os principais erros sintáticos e semânticos. A partir da próxima seção, serão detalhadas as instruções primitivas e a sintaxe correta de uso de cada uma delas.

SEÇÃO 1
INSTRUÇÃO PRIMITIVA DE ENTRADA DE DADOS

Faz com que o computador busque no dispositivo de entrada, dados que são guardados nas posições de memória, correspondentes às variáveis da lista que lhe são passadas como argumento. É a forma que um usuário pode fornecer informações ao computador .

Em pseudocódigo sua sintaxe é:

```
Leia <variável> ou <lista_de_variáveis>;
```

Possivelmente você pode estar se questionando, mas como eu represento as variáveis, fique tranquilo, em breve elas serão estudadas em um tópico específico, pois são muito importantes no contexto da produção de algoritmos e programas, neste momento, considere a <variável> como um argumento da instrução **Leia**.

Algo que deve ser foco de sua atenção agora é o fato de você fixar que a palavra **Leia** é uma instrução primitiva de entrada de dados no computador e que após ela, devem ser listadas as variáveis e o fechamento da linha de comando, que deve ser com um ponto e vírgula (**;**). Este modo de escrever a instrução **Leia**, corresponde à sintaxe correta de uso desta instrução primitiva. Então, pode-se afirmar que, caso você não coloque o ponto e vírgula ao final da sentença, estará cometendo um erro sintático.

ATENÇÃO

Há autores que na construção dos algoritmos não utilizam o ponto e vírgula, porém, como a maioria absoluta das linguagens de programação o utilizam, o uso mesmo em algoritmos torna-se um hábito útil, pois quando for escrever seus próprios programas, dificilmente vai esquecer do ponto e vírgula.

A outra instrução primitiva imprescindível em algoritmos e programas é a de atribuição.

SEÇÃO 2
INSTRUÇÃO PRIMITIVA DE ATRIBUIÇÃO

É o principal modo de se armazenar um dado numa variável. Conforme já dito, variável é um assunto posterior, neste momento concentre-se na sintaxe.

Em pseudocódigo sua sintaxe é:

```
<nome_da_variável>  :=  <expressão>;
```

Ao observar a sentença exibida anteriormente, percebe-se que o fechamento da linha também é feito por um ponto e vírgula, como na instrução primitiva de entrada e entre a variável e a expressão há um operador, **:=** , que é efetivamente o comando de atribuição. Na literatura podem ser encontrados outros símbolos para os operadores de atribuição, todos válidos, tais como: = ou ← , neste livro e na disciplina o padrão, sintaxe, será o **:=** .

O funcionamento de uma instrução de atribuição deve ser analisada da direita para a esquerda, pois inicialmente a expressão, da direita, é avaliada e o resultado é armazenado na variável, à esquerda. A <expressão> pode ser representado de quatro modos diferentes:

1. como uma expressão matemática, algo como, < 2 + 2 >;
2. pode ser uma variável;
3. uma constante;
4. ou uma combinação dos três itens anteriores.

No item 3 percebe-se que há um termo novo, **constante**!

Uma **constante** na informática tem o significado similar ao da matemática, em que uma **constante** é um valor fixo, que no decorrer do algoritmo ou execução de um programa sempre terá o mesmo valor.

Evidentemente, a noção é contrária ao de uma variável, em que o valor não é fixo.

Exemplos de constantes podem ser simplesmente números, por exemplo, 5, ou símbolos alfabéticos que a eles são atribuídos valores, por exemplo, PI, pode ter um valor a ele atribuído de 3.14159.

De um modo simplificado, pode-se dizer que a **atribuição** nessa etapa dos estudos, está representando o **Processamento**, pois é uma atividade em que o computador busca dados da memória, que foram transferidos pelo dispositivo de entrada e efetua uma atribuição a uma determinada variável.

Ainda de modo simplificado, pode-se dizer que, após o processamento, o computador efetuará uma saída de dados para o usuário, o que é visto na próxima seção.

SEÇÃO 3
INSTRUÇÃO PRIMITIVA DE SAÍDA DE DADOS

É a instrução primitiva que caracteriza a saída de dados do computador e pode ter como argumentos, uma variável ou uma lista de variáveis, literais ou uma mescla desses itens. Quando o argumento são variáveis, o valor delas é buscado na memória do computador e enviado ao dispositivo de saída, quando são literais eles são enviados diretamente no dispositivo de saída.

Lembrando que, previamente neste livro convencionou-se que o monitor de tela de vídeo do computador é o dispositivo de saída padrão. Pode-se resumir o processo de saída de dados, como o meio pelo qual as informações que estão contidas no computador podem ser colocadas nos dispositivos de saída, para que o usuário possa consultá-las.

Em pseudocódigo sua sintaxe é:

```
Escreva <lista_de_variáveis>;
ou
Escreva <literal>;
```

Em uma mesma linha de instruções pode-se misturar nomes de variáveis com literais na lista.

Você deve ter percebido que novamente há um termo novo e ainda não explicado, que é o **literal**, é um assunto que será tratado com mais detalhes na próxima unidade, agora simplesmente considere como sendo uma sequência ou cadeia de caracteres alfanuméricos que geralmente é utilizado para representar palavras, frases ou textos, sendo esses itens delimitados por aspas, conforme o exemplo a seguir, figura 6, com um trecho de um algoritmo, que também possui uma variável.

```
x:= 10;
Escreva "O valor de x é ", x;
```

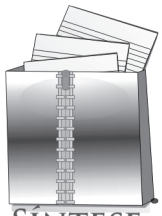
Figura 6: Exemplo de uso de literais com instrução primitiva de saída.

Considerando que as linhas do algoritmo da figura 6, fossem codificadas em alguma linguagem de programação, quando fossem executadas pelo computador, a expectativa de saída na tela do computador seria o ilustrado na figura 7.

```
O valor de x é 10
```

Figura 7: Exemplo de saída de programa com literais na instrução primitiva

Observe na figura 7, que o computador exibe apenas a frase que está entre as aspas e o conteúdo da variável x, 10. Isto significa que, as demais partes fazem parte do código-fonte do programa e somente os argumentos da instrução primitiva de saída são exibidos durante a execução do programa.



SÍNTESE

Nesta unidade, você pôde conhecer as instruções primitivas, com as quais o computador pode comunicar-se com o usuário e o mundo exterior, pois, permitem que o usuário forneça dados ao computador, com a instrução **Leia**, que podem de algum modo ser processadas, neste contexto foi abordado à atribuição de dados para variáveis, **:=**, e a exibição do resultado por meio da instrução **Escreva**.

Agora é oportuno relembrar o que foi dito na Unidade I, quando da definição de Niklaus Wirth: *programas de computador são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados*, foi dito também que sobre a representação dos dados, haveria um estudo mais detalhado, então chegou a hora e este estudo inicia-se na próxima Unidade com os Tipos de Dados.


ANOTAÇÕES

UNIDADE V

Tipos de dados

Objetivos de aprendizagem

- Aprimorar o entendimento sobre o fluxo de execução de um programa de computador.
- Conhecer os tipos de dados que são declarados em um algoritmo.

Roteiro de estudos

- SEÇÃO 1 – Dados Numéricos
- SEÇÃO 2 – Dados Literais (Alfanuméricos)
- SEÇÃO 3 – Dados Lógicos

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Antes, de efetivamente, falar sobre os tipos de dados é importante você ter em mente como os computadores funcionam, mesmo que seja uma ideia simplificada. Basicamente, todo o processamento que é realizado por um computador é executado pela manipulação das informações contidas em sua memória, cujas informações, resumidamente, podem ser classificadas em dois tipos:

- **Instruções/Comandos** – que são os responsáveis pelo funcionamento do computador e determinam como os dados devem ser tratados.
- **Dados** – correspondem aos conjuntos de informações que serão processadas pelo computador.

Algo que ajuda a entender o funcionamento dos computadores é fazer uma analogia em relação ao ser humano, no que se refere ao tratamento das informações. Evidentemente é um comparativo distante, o intuito é apenas facilitar o entendimento de partes do computador, por exemplo:

- **Memória** – nas seções anteriores foi citado várias vezes que o computador tem uma memória, por sua vez, todos os seres humanos também tem, então, há uma semelhança.
- **Cérebro** – nele que os seres humanos possuem o conhecimento de como processar as diversas informações que são captadas do ambiente, por exemplo: por meio visual pode-se ler um texto, entrada e armazená-lo em nossa memória, lembrança, ou simplesmente, fazer a leitura em voz alta, saída, do texto lido. A decisão e o processamento dessas informações, são gerenciadas pelo cérebro, assim como, os computadores fazem por meio da UCP. Há aqui, mais uma analogia.
- **Dispositivos de entrada** – como dito anteriormente, a percepção visual do ambiente é um modo de captar informações, a audição pode ser outro. Analogicamente falando, é similar à entrada de dados do computador, quando ele recebe informações para serem processadas posteriormente.

- **Dispositivos de saída** – o fato do ser humano falar ou escrever algo é um modo de dar a saída das informações, que foram previamente processadas em seu cérebro, similarmente ao que o computador faz quando, por exemplo, exibe um resultado de processamento da tela do monitor de vídeo.
- **Dispositivos de armazenamento** – sabe-se que os computadores possuem unidades de disco, CDs, DVDs, que têm por finalidade fazer um armazenamento de dados de forma permanente, para uma posterior consulta ou processamento. O ser humano possui atividades parecidas, quando faz anotações em papel, ou mesmo, quando usa o computador para guardar informações de forma permanente, para uso posterior.

A figura 8 auxilia no entendimento dessa analogia entre **computador x humano**, pois fica nítido que realmente existe uma semelhança e ela ajuda a abstrair a ideia de como os computadores funcionam, sendo algo que ajuda na resolução de problemas do ponto de vista computacional, quando do desenvolvimento de programas. Algo que também fica claro é o fato que tanto os seres humanos como os computadores recebem dados e informações e as processam, no caso dos computadores os programas são o conhecimento de como os dados serão processados. No caso dos seres humanos é o conhecimento que adquire-se durante a vida, inclusive de como escrever os programas.

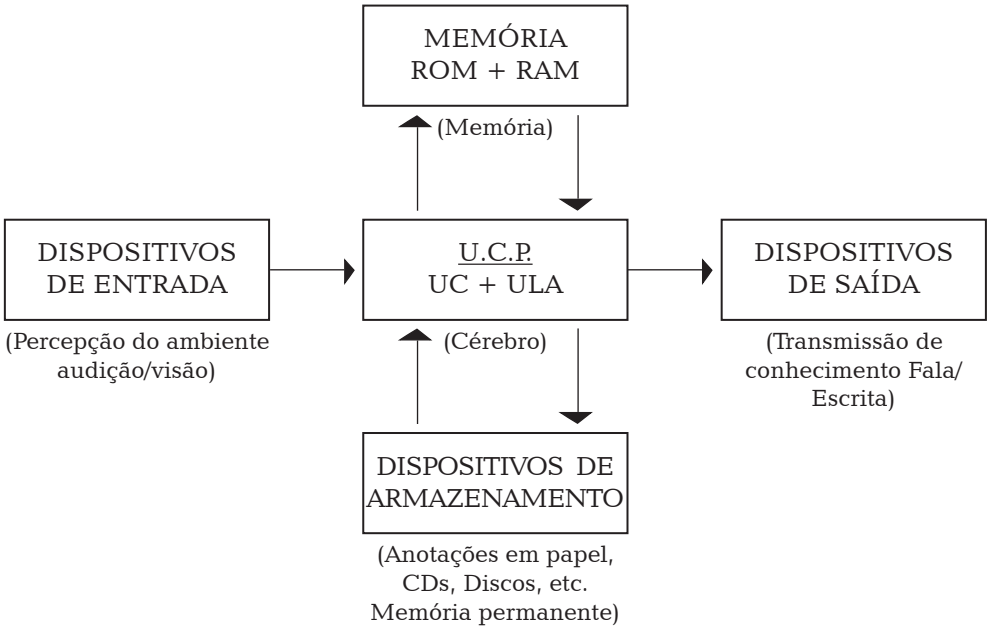


Figura 8: Analogia entre *computador* x *humano*

Uma vez entendida a analogia descrita anteriormente, antes de continuar com as instruções, o foco é como os dados podem ser classificados para que os computadores possam entendê-los e processá-los. Embora os computadores sejam capazes de reconhecer apenas dois estados diferentes, situação que em sua arquitetura é representada pelo sistema numérico binário, do ponto de vista computacional os dados dividem-se em três tipos: **numéricos**, **literais** e **lógicos**.

SEÇÃO 1
DADOS NUMÉRICOS

Apesar do sistema numérico nativo do computador ser o binário, pode-se afirmar que os números, de um modo geral, podem ser representados em qualquer tipo de base, por exemplo, o sistema decimal que possui dez dígitos {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, sendo o sistema que mais se utiliza no dia a dia e por isso é o sistema numérico utilizado para representar os dados numéricos, assim como ocorre na matemática, em que são classificados em: números naturais, inteiros, fracionários e reais.

- Conjunto dos números **naturais** representados por **N**;

  N = {0,1,2,3,4,5.....}

É o conjunto de números com os quais se pode representar quantidades e medidas, efetuar cálculos, ou seja, são números que permitem diversos tipos de quantificações.

- Conjunto dos números **inteiros** (**Z)**;

  Z = {...,-4,-3,-2,-1,0,1,2,3,4,...}

São números que também permitem diversas formas de quantificações, como os números naturais, mas agora se incluem operações em que se pode representar números negativos.

- Conjunto dos números **fracionários** (**Q**);

  Q = {p/q | p,q pertencem a **Z**}

Além das formas de representações possíveis, como dos números naturais e inteiros, os números fracionários possibilitam como o próprio termo expressa, fracionar e/ou dividir os números inteiros de modo que as representações incluam frações.

- Conjunto dos números **reais** (**R**);

Dando sequência ao raciocínio de abrangência dos conjuntos de números anteriormente descritos, os números reais criam uma relação de pertinência entre os conjuntos de números naturais, inteiros e fracionários, conforme ilustrado na figura 9.

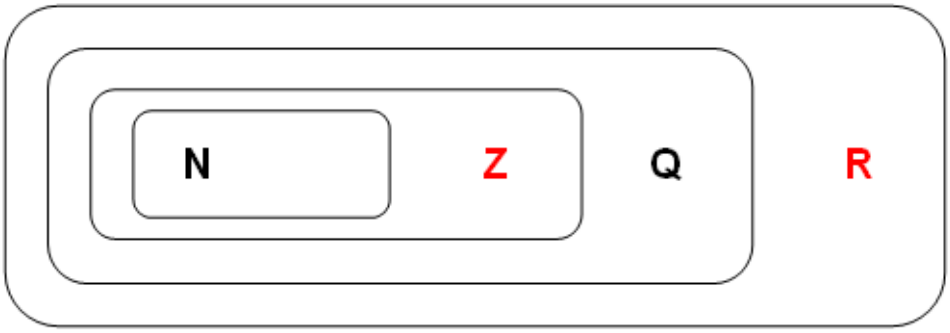


Figura 9: Classificação dos dados numéricos.

Do ponto de vista do computacional (algorítmico), os dados numéricos representáveis num computador são divididos em apenas duas classes: **inteiros** e **reais**. Essas duas classes de números são suportadas por todas as linguagens de programação, bem como, em várias delas pode haver uma subdivisão de categorias em cada classe.

Uma vez que você já conhece um pouco sobre a representação de dados numéricos, agora vai aprender um pouco mais sobre dados literais, os quais foram citados quando foram estudadas as instruções primitivas.

SEÇÃO 2
DADOS LITERAIS (ALFANUMÉRICOS)

Os dados literais também são chamados de **alfanuméricos**, **cadeia** (ou **cordão**) **de caracteres** ou ainda do inglês ***strings***, sendo o termo em inglês o mais usual entre os programadores. São constituídos por uma sequência de caracteres contendo letras, dígitos e/ou símbolos especiais, e usualmente são representados nos algoritmos e nas linguagens de programação pela coleção de caracteres, delimitada em seu início e término com o caractere **aspas duplo** (") ou **aspas simples - plicas** ('). O uso de aspas duplo ou simples pode diferir entre as linguagens de programação, nos estudos desta disciplina o padrão na escrita dos algoritmos em portugol será o aspas simples ('), por ser a sintaxe utilizada na linguagem de programação Pascal, a ser utilizada nos exercícios práticos.

Um fator de destaque em diversas situações, quando do uso dos literais é conhecer o **tamanho** ou **comprimento** deles, que é definido pela quantidade de caracteres, inclusive os espaços em branco, mesmo que não sejam visíveis.

Dos três tipos de dados básicos, o próximo são os dados lógicos, sendo por meio dos quais o computador geralmente adquire a capacidade de tomar decisões.

SEÇÃO 3
DADOS LÓGICOS

Também chamados de **booleanos**, em razão da significativa contribuição de George Boole, filósofo britânico criador da álgebra booleana. Os estudos de Boole na área da lógica matemática foram fundamentais para o desenvolvimento da computação, pois a manipulação de dados lógicos representa, em grande parte, a maneira como os computadores funcionam, principalmente em situações em que o computador toma decisões.

Os dois únicos valores possíveis para representar os dados lógicos são: **Verdadeiro/Falso**. Contudo, quando se trata de algoritmos e, mesmo em linguagens de programação, outras representações podem ser contempladas na literatura que do mesmo modo representam apenas dois valores lógicos, tais como: **Sim/Não – 1/0** – **True/False – .T./.F.**. Nesta disciplina, a convenção sintática a ser usada na escrita dos algoritmos em portugol será:

- **.V.** - valor lógico verdadeiro.
- **.F.** - valor lógico falso.

Um bom modo de consolidar os conhecimentos desta unidade é por meio de uma imagem, uma vez que, este também é um item que deve ser fixado, pois sem o conhecimento dos tipos de dados computacionais não é possível escrever algoritmos e por consequência programas de computador.

Observe com atenção a figura 10 e procure memorizar essa estrutura.

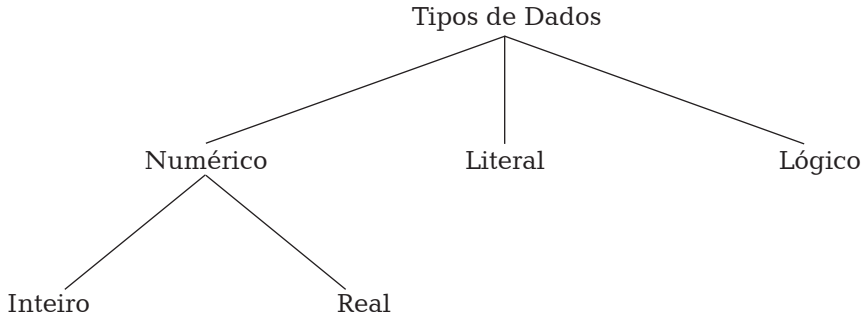


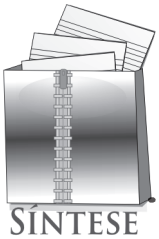
Figura 10: Classificação dos tipos de dados.

Chegou o momento de realizar alguns exercícios, isso o ajudará a fixar os conhecimentos desta unidade. Após concluir a atividade, você pode comparar suas respostas com as apresentadas no final deste livro.

Caso não haja coincidências em alguma de suas respostas refaça o exercício, analisando o porquê da diferença.

→ Classifique os dados especificados, a seguir, de acordo com seu tipo, assinalando com (I) os dados do tipo INTEIRO, com (R) os REAIS, com (L) os LITERAIS, com (B) os LÓGICOS (booleanos), e com (N) aqueles para os quais não é possível, a priori, definir um tipo de dado do ponto de vista computacional.

( ) 0	( ) –0.001	( ) –0.0
( ) 1	( ) +0.05	( ) .V.
( ) 0.0	( ) +3257	( ) V
( ) 0.	( ) “a”	( ) ‘abc’
( ) –1	( ) “+3257”	( ) F
( ) –32	( ) ‘+3257.’	( ) .F
( ) +36	( ) “–0.0”	( ) “V”
( ) +32.	( ) ‘.F.’	( ) .F.



SÍNTESE

Nesta etapa dos estudos, você pôde ampliar os seus conhecimentos a respeito de como o computador funciona e também sobre o fluxo de execução dos programas do computador.

Outro assunto fundamental foi sobre os tipos de dados que são utilizados no desenvolvimento dos algoritmos e programas. Sobre esse tópico é de suma importância que você memorize a estrutura da figura 10, pois se não souber como os dados são classificados não é possível escrever algoritmos, que efetivamente tenham alguma finalidade. Se houver dúvidas, reestude a unidade.

É possível que a medida que você foi estudando esta unidade, possa ter surgido um questionamento – mas como vou trabalhar com os tipos de dados e onde eles se encaixam em um algoritmo ou programa? – pois bem, essa resposta está na unidade que trata das **variáveis**, porém, para que o entendimento sobre variáveis possa ser completo, antes é necessário ter uma noção de como o computador representa as informações em seus circuitos e como isso pode ser mensurado.

ANOTAÇÕES

UNIDADE VI

Representação da Informação

Objetivos de aprendizagem

- Entender como os dados são armazenados na memória do computador.
- Entender e saber como mensurar as unidades de medida das memórias dos computadores.
- Saber o espaço necessário para cada tipo de dados.

Roteiro de estudos

- SEÇÃO 1 – Armazenamento de dados na memória principal
- SEÇÃO 2 – Armazenamento de Dados do Tipo Literal
- SEÇÃO 3 – Armazenamento de Dados do Tipo Lógico
- SEÇÃO 4 – Armazenamento de Dados do Tipo Inteiro
- SEÇÃO 5 – Armazenamento de Dados do Tipo Real

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

O usuário, à medida que vai intensificando o uso do computador é comum preocupar-se com a capacidade de armazenamento de dados que o computador pode suportar, pois, à medida que os trabalhos vão sendo realizados, os dados geralmente são guardados em arquivos, sejam documentos, músicas, fotos, vídeos e os mais diversos tipos de imagem, e gradativamente esse processo vai esgotando a capacidade dos discos e das demais mídias de armazenamento que o computador pode possuir.

Nesse contexto, é fundamental ter conhecimento de como mensurar o espaço ocupado pelos dados, bem como, o que resta de espaço disponível. Diante disto, é crucial conhecer quais são as unidades de medida e como são aplicadas nas memórias dos computadores.

Quando se refere às memórias, é pelo fato que o termo memória é genérico para determinar as partes do computador ou dos dispositivos periféricos onde os dados e programas são armazenados de forma permanente ou temporária. Pode-se afirmar que, sem as memórias, o computador não teria como funcionar, pois não haveria de onde obter os programas e os dados, para que fossem lidos, e/ou escritos e/ou processados.

As memórias do computador dividem-se basicamente em duas categorias:

- **Principal** ou **Primária** – é aquela destinada ao armazenamento de informações, sejam elas dados ou programas para uso imediato, e sem a memória principal o computador não pode funcionar. O setor básico desta memória é denominado RAM (*Random Access Memory*), isto é, memória de acesso aleatório e que necessita de energia elétrica para manter os dados, por isso também possui a característica de ser volátil. Outro setor da memória principal é a ROM (*Read Only Memory*) ou memória apenas de leitura, que contém dados permanentes, por isso não

é volátil. Nela o fabricante do processador grava as instruções básicas que coordenarão o funcionamento do computador.

- **Secundária** – também chamadas memórias de massa, são aquelas que têm como finalidade principal o armazenamento de dados de forma permanente, por essa razão não são voláteis. Exemplos: discos rígidos (HDs - *Hard Disk*), pendrives, cartões de memória, SSDs, CDs e DVDs. Para que as informações sejam gravadas na memória secundária, antes devem ser carregadas na memória primária, posteriormente o processador efetua o processamento e a partir daí, conforme as diretivas do programa o resultado do processo pode ser gravado em alguma memória de armazenamento permanente.

SSD (*Solid-State Drive*) é uma tecnologia recente de armazenamento de dados que não possui partes móveis e é construído em torno de um circuito integrado semicondutor, o qual é responsável pelo armazenamento, diferentemente dos sistemas magnéticos como os discos rígidos.

Apesar do cérebro humano ainda ser um mistério em vários aspectos, é possível conjecturar que um dos modos que os seres humanos armazenam informações na memória é por meio de imagens. Porém, a tecnologia para que os computadores pudessem armazenar informações de forma similar ao cérebro humano ainda não foi desenvolvida. Mas a tecnologia digital dos computadores atuais até tenta simular de forma distante o comportamento do cérebro humano, utilizando como base o sistema binário, em que, apenas dois estados (valores) são possíveis, algo como: ligado/desligado ou verdadeiro/falso. A partir destes dois valores, que a representação da informação dos computadores se inicia.

BIT

Unidade básica da informação é o **BIT** (**BI***nar*y *digi***T**), a porção de informação fornecida por um verdadeiro ou falso, uma resposta sim/não, como uma chave liga/desliga, simbolicamente os dígitos 0 e 1 são utilizados.

BYTE (OCTETO)

BYTE (**B***inar***Y TE***rm*), constituído por um conjunto de oito bits. Desta forma, é possível representar **256 (2^8)** combinações diferentes, que incluem o alfabeto como um todo, os números do sistema decimal e diversos símbolos.

KILOBYTE (KBYTE)

Um kbyte equivale a **1024** octetos ou bytes (8192 bits). O fato de corresponder a 1024 e não a 1000 se deve a que 1 kbyte é igual a $\mathbf{2^{10}}$ (base do sistema binário elevada a décima potência), além disso, essa correspondência faz com que o kbyte seja múltiplo de 8.

MEGABYTE (MBYTE)

Equivale a **1.048.576** de bytes (octetos), ou seja, $\mathbf{2^{20}}$ (base do sistema binário elevada a vigésima potência), para representação de grande volume de memória.

GIGABYTE (GBYTE)

Equivale a **1.073.741.824** bytes - $\mathbf{2^{30}}$ (base do sistema binário elevada a trigésima potência).

TERABYTE (TBYTE)

Equivale a **1.099.511.627.776** bytes - $\mathbf{2^{40}}$ (base do sistema binário elevado a quadragésima potência).

PETABYTE (PBYTE)

Equivale a **1.125.899.906.842.624** bytes - $\mathbf{2^{50}}$ (base do sistema binário elevado a quinquagésima potência).

Ao estudar as unidades de medida usadas pelos computadores, foi fácil perceber que é possível armazenar um volume expressivo de informações e por outro lado, é interessante refletir sobre a genialidade com que isso ocorre, uma vez que, a base de tudo são apenas dois valores, 0 e 1.

O passo seguinte é entender como as informações são organizadas na memória do computador e com relação aos tipos de dados, qual o consumo de memória de cada um.

SEÇÃO 1
ARMAZENAMENTO DE DADOS NA MEMÓRIA PRINCIPAL

De forma geral, a memória de um computador é formada por um conjunto ordenado de células (bytes), sendo que cada uma delas é identificada por um número inteiro positivo distinto, conhecido como **endereço (ou ponteiro)**, que caracteriza de maneira única a posição de cada informação armazenada na memória.

Diante desse contexto, pode-se generalizar que a unidade de medida que deve ser usada como referência a partir de agora é o byte, e que ele corresponde ao espaço necessário para armazenar uma letra, número ou algum símbolo especial. Essa referência auxilia como mensurar o quanto de memória cada tipo de dado necessita, para ser usado durante a execução de um programa de computador.

SEÇÃO 2
ARMAZENAMENTO DE DADOS DO TIPO LITERAL

Em razão do que já foi estudado a respeito de literais, subtende-se que cada caractere de um literal é representado por um byte, sendo assim, como é possível representar 256 (2^8) combinações diferentes, uma vez que, o byte possui 8 bits. Com base nesse fato, o cientista da computação norte-americano Robert William Bemer propôs associar cada caractere a um número (código) diferente variando de 0 a 255 (256 possibilidades). Essa convenção é representada na forma de uma tabela de mapeamento de caracteres e números, denominada **ASCII** (*American Standard Code for Information Interchange*/Código Padrão Americano para Intercâmbio de informações), ilustradas nas figuras 11 e 12.

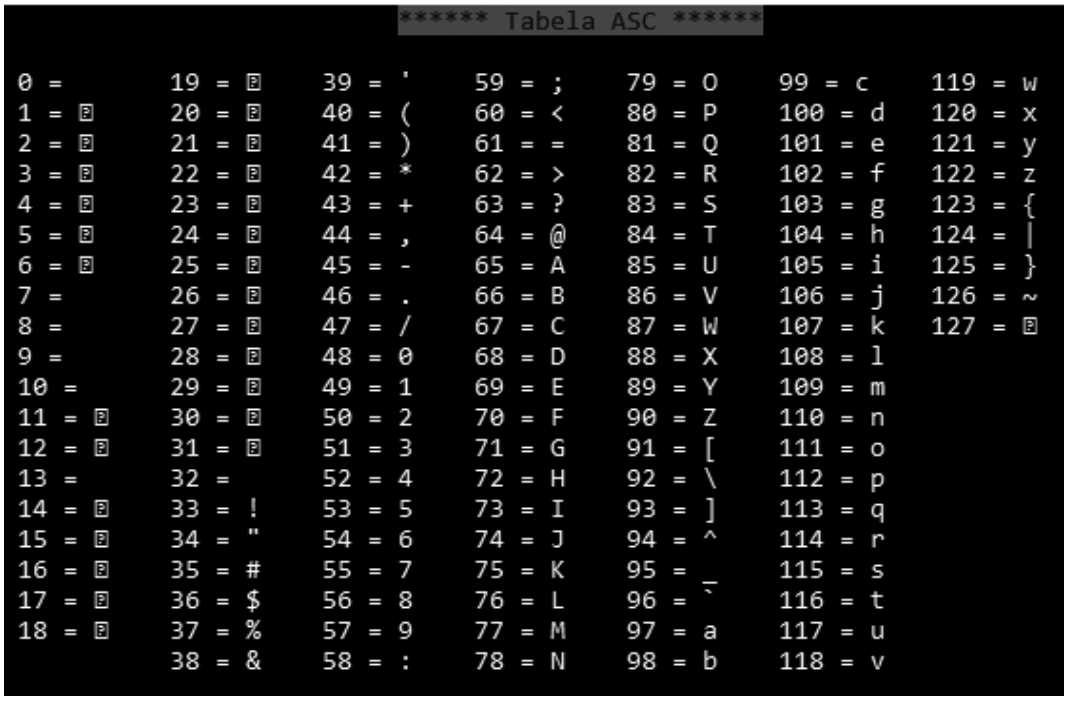


Figura 11: Tabela ASC

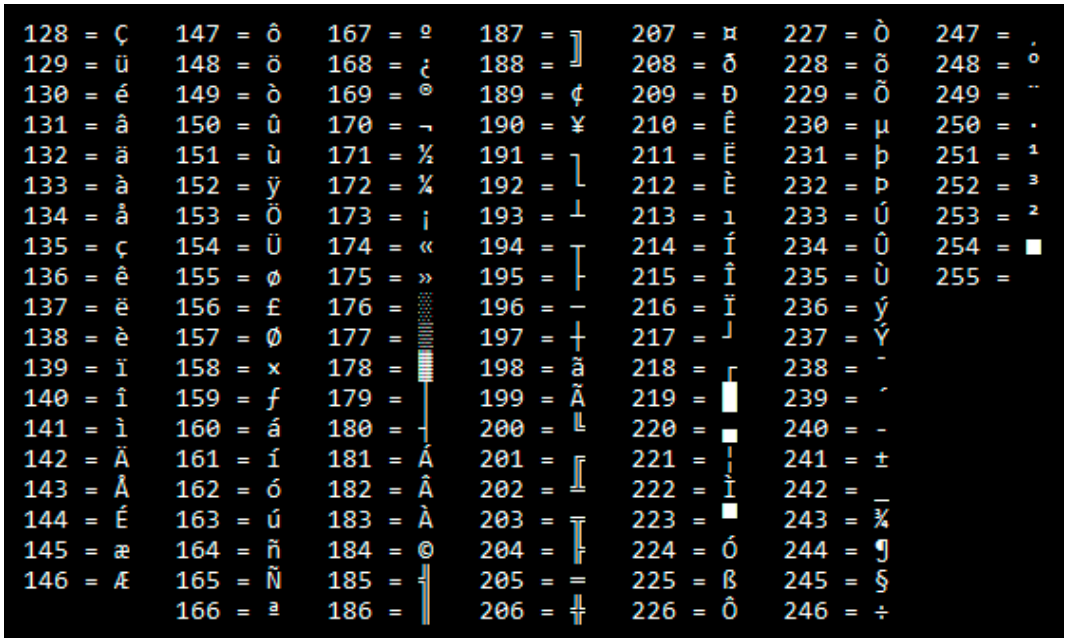


Figura 12: Tabela ASCII - estendida

A tabela ASC é formada pelos caracteres de controle de 0 a 31, não imprimíveis, alguns deles relacionados ao teclado do computador como o *Enter* (13), *Tab* (9), *Esc* (27); de 32 a 127, estão os caracteres de texto plano; de 128 a 255 são representados os caracteres especiais, que fazem parte da tabela ASCII estendida.

Em resumo, cada caractere do tipo literal é constituído por um byte e como seu tamanho é definido pelo comprimento (quantidade de caracteres/bytes). Portanto, para guardar um literal na memória do computador deve-se alocar (reservar) um espaço contíguo de memória igual ao seu comprimento.

SEÇÃO 3
ARMAZENAMENTO DE DADOS DO TIPO LÓGICO

Apesar dos dados lógicos possuírem dois valores apenas, como um *Verdadeiro* ou *Falso*, necessitam de um byte para serem armazenados na memória do computador. Pois, esta situação reflete como a arquitetura dos computadores é constituída.

Sobre os dados numéricos, diferentemente dos dados lógicos, você vai perceber que possuem algumas particularidades na forma como são armazenados na memória, pois, além da quantidade de bytes que consomem, deve-se também levar em consideração quais são os valores possíveis de serem representados.

SEÇÃO 4
ARMAZENAMENTO DE DADOS DO TIPO INTERNO

Na maioria das linguagens de programação são armazenados em 2 bytes, que equivalem a (2^8 x 2^8 = 2^{16}), 65536 possibilidades, com as seguintes situações de números possíveis de representar:

(-32767, -32766, ..., -2,-1,0,1,2,..., 32766, 32767) – com sinal.

(0,1,2,..., 65534, 65535) – sem sinal.

Então, percebe-se que dada a combinação de bits possíveis, (2^{16}), quando são tratados apenas números inteiros sem sinal, apenas positivos, pode-se representar valores entre 0 até 65535, se for com sinal o valor máximo positivo ou negativo será 32767.

Contudo, existem linguagens de programação que admitem representar dados do tipo inteiro em 4 ou 8 bytes, ditos **inteiros longos** ou **estendidos**, em que os valores possíveis serão maiores.

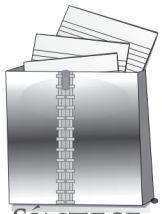
Do ponto de vista algorítmico e por padronização, nessa disciplina os números inteiros serão sempre armazenados em 2 bytes.

SEÇÃO 5
ARMAZENAMENTO DE DADOS DO TIPO REAL

Os números reais possuem características similares à dos números inteiros, porém normalmente são armazenados em 4 bytes, (2^8 x 2^8 x 2^8 x 2^8 = 2^{32}), ou seja, o número real máximo que pode ser representado é 4.294.967.296.

Assim como, com os números inteiros, existem linguagens de programação que admitem o **real estendido** com 8 bytes (2^{64}) 18.446.744.073.709.551.616 possibilidades, ou 16 bytes (2^{128}) 3,40282366920938463463374607431768e+38 possibilidades.

Também por padrão, nessa disciplina os números reais a serem considerados, serão sempre armazenados em 4 bytes.



SÍNTESE

O que foi abordado nesta unidade, permitiu a você conhecer e entender como as informações são representadas pelos computadores, tanto do ponto de vista das unidades de medida, que são utilizadas para mensurar o espaço ocupado nas memórias do computador, como o armazenamento dos tipos de dados.

Sobre as unidades de medida, foram vistas desde o bit, que é a menor unidade do computador, como o byte, que generalizando corresponde a um caractere, o qual será muito utilizado no desenvolvimento dos algoritmos. Outras medidas foram vistas, como o Kilobyte, Megabyte, Gigabyte e Petabyte.

Com relação ao espaço de memória necessário para armazenar os tipos de dados: para os literais, o comprimento é proporcional a quantidade de bytes; para os lógicos, um byte é suficiente para cada dado deste tipo; para os numéricos, os inteiros foram padronizados em 2 bytes e os reais em 4 bytes, para o armazenamento de cada número destes tipos, respectivamente.

Assim como já foi comentado em outros assuntos, o que foi estudado nesta unidade precisa ser memorizado, para a sequência dos estudos. Caso haja alguma incerteza, reestude a unidade.

A próxima unidade vai nortear como vincular os tipos de dados às variáveis e detalhar como elas são declaradas e utilizadas no desenvolvimento dos algoritmos.


ANOTAÇÕES

UNIDADE VII

Variáveis

Objetivos de aprendizagem

- Conceituar as variáveis.
- Declarar variáveis.
- Conhecer a sintaxe de uso das variáveis.

Roteiro de estudos

- SEÇÃO 1 – Declaração de variáveis em Algoritmos
- SEÇÃO 2 – Mapeamento de variáveis na memória

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

Variáveis em programas realmente são imprescindíveis, pois sem elas praticamente os programas não teriam finalidade, uma vez que, dificilmente os programas **não** manipulam dados, por mais simples que sejam. As variáveis possibilitam que dados sejam tratados, armazenados e poderão sofrer alterações em seus valores. Como o próprio termo expressa, variável é aquilo que é mutável.

Uma variável é uma entidade que possui dois vínculos principais, o **dado** ao qual ela está relacionada e o **endereço** na memória do computador, onde efetivamente a informação está armazenada. Sintetizando a ideia, para que o computador não **esqueça** das informações, ele inicialmente precisa guardá-las em sua memória principal, para que os programas, por meio de suas instruções, possam efetivamente realizar os processamentos requeridos.

Além dos detalhes vistos anteriormente, as variáveis possuem três atributos, para que se possa criá-las:

- **Nome (identificação) –** de uma forma geral, as regras para criação de nomes de variáveis são as seguintes: deve começar necessariamente com uma letra e não deve conter nenhum símbolo especial exceto a sublinha (_), não pode haver espaços em branco entre as palavras que compõem o nome da variável e números podem ser utilizados.
- **Tipo de dado –** inteiro, real, literal ou lógico.
- **Informação (dado) –** sendo o único elemento que poderá sofrer alterações durante a execução do programa.

É importante salientar que estes três atributos são indissociáveis e todos devem ser utilizados no momento em que as variáveis são criadas, em programação de computadores chama-se **declaração de variáveis**.

SEÇÃO 1
DECLARAÇÃO DE VARIÁVEIS EM ALGORITMOS

Como regra geral, por ser o que ocorre na maioria das linguagens de programação, as variáveis devem ser declaradas no início do programa, antes de serem utilizadas, para que o compilador ou interpretador reserve espaço na memória para elas. Embora, as sintaxes para declaração das variáveis possam diferir entre as diferentes linguagens de programação, o mesmo ocorre entre os algoritmos, neste livro será adotada a sintaxe:

VAR

<nome_da_variável> : <tipo_da_variável>;

ou

VAR

<nomes_das_variáveis> : <tipo_das_variáveis>;

Sintaxe:

- A palavra-reservada **VAR** deverá estar presente e será utilizada uma única vez, na declaração de um conjunto de variáveis ou de apenas uma.
- Numa mesma linha, poderão ser declaradas 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.
- Cada declaração de variável e/ou lista de variáveis devem ter a sentença, linha, encerrada por um ponto e vírgula.

A seguir, um exemplo prático de declaração de variáveis em portugol, com os tipos de dados já estudados.

EXEMPLO

```
VAR
    NOME_CLIENTE : literal[30];
    IDADE_CLIENTE : inteiro;
    SALARIO_CLIENTE : real;
    TEM_FILHOS_CLIENTE : lógico;
```

No exemplo ilustrado anteriormente, estão declaradas quatro variáveis, conforme detalhamento a seguir:

- **NOME_CLIENTE** – variável do tipo literal, para a qual o compilador reservará 30 bytes na memória do computador, que corresponde ao tamanho estabelecido na declaração **literal [30]**.
- **IDADE_CLIENTE** – variável do tipo inteiro, que possuirá 2 bytes reservados na memória. Neste exemplo, cabem algumas analogias preliminares a respeito do tipo de dado que foi escolhido: quando da escolha do tipo das variáveis, os programadores devem preocupar-se em escolher um tipo que represente integralmente a informação que será armazenada por aquela variável, mas também que não desperdice memória, ou seja, para a qual seja reservado espaço da memória além do que realmente é necessário – exemplificando, se fosse escolhido o tipo real, que necessita de 4 bytes, resultaria num gasto desnecessário de 2 bytes, além do que, estaria sendo usado um número fracionário para uma variável, idade, que originalmente necessita apenas de números inteiros.
- **SALARIO_CLIENTE** – variável do tipo real, que necessitará de 4 bytes de memória. Percebe-se que é um tipo de dado adequado a informação, principalmente pelo fato de que para representar salários, são necessários números fracionários em razão dos centavos, além do que, o valor máximo que pode ser representado por tipos reais é mais do que suficiente para representar salários, mesmo que sejam altos.
- **TEM_FILHOS_CLIENTE** – variável do tipo lógico, que utilizará apenas um byte de memória. Como aparentemente o propósito

dessa variável é apenas saber se o cliente possui filhos, o tipo é totalmente adequado, pois os valores possíveis para este tipo são *Verdadeiro* ou *Falso*, os quais também podem expressar algo como: *Sim* ou *Não*. Contudo, caso o objetivo fosse a quantidade de filhos, evidentemente o tipo mais adequado seria o inteiro, que consumiria 2 bytes de memória.

Além do que já foi estudado, existem mais algumas regras básicas que devem ser respeitadas quando da utilização de variáveis nos algoritmos e programas.

Quando da utilização da variável em operações de atribuição ou em expressões matemáticas, todos os elementos envolvidos, variáveis e constantes devem ser do mesmo tipo.

- **Coerção** – é a exceção em relação à regra anterior, onde há uma variável do tipo real pode ser atribuído o resultado de tipo inteiro da avaliação de uma expressão. Neste caso, o resultado de tipo inteiro é convertido para o tipo real e posteriormente armazenado na variável.
- Não há ordem definida para os tipos de dados, quando da declaração das variáveis.
- Na informática, existe um termo denominado *case sensitive*, que corresponde ao fato de um determinado editor ou ambiente de programação ser sensível a maiúsculas e minúsculas. Isto quer dizer, que aquele ambiente diferencia as letras maiúsculas das minúsculas e portanto, no caso das variáveis, uma vez, declaradas maiúsculas ou minúsculas, no decorrer do algoritmo ou programa devem ser utilizadas do mesmo modo que foram escritas na declaração. Apesar de existirem ambientes de programação que não são *case sensitive*, como é o caso do Pascalzim, a sugestão é que os algoritmos sejam escritos como se fossem. Esta prática evita erros futuros, quando o algoritmo foi codificado em alguma linguagem de programação que seja *case sensitive*.
- Uma variável sempre armazena um único valor, o qual pode ser substituído por outro, de mesmo tipo, durante um processo de atribuição.

- A escolha do nome da variável deve, quando possível, expressar a informação que será armazenada por ela.
- A escolha do tipo da variável deve ser compatível com as características da informação que será armazenada por ela.
- **Não usar acentuação e nem cedilha, como no caso do < ç >**, embora nos algoritmos seja permitido, nas linguagens de programação não é, pois elas utilizam como idioma principal o inglês. Diante disso, como sugestão, mesmo em algoritmos, usar os nomes de variáveis sem acentuação, bem como os respectivos nomes dos algoritmos.

Como você deve ter percebido, a respeito de variáveis há uma série de detalhes os quais terão que ser seguidos à risca quando do desenvolvimento dos algoritmos. Caso você esteja inseguro sobre algum dos tópicos anteriores, reestude antes de prosseguir. Isso é fundamental nesta disciplina, pois os assuntos são todos interligados e deve ter sido fácil de perceber que um dependo do outro para que efetivamente se aprenda.

O desfecho desta unidade é focado em como as variáveis ficam distribuídas na memória do computador, a partir da declaração delas, isso se chama mapeamento de variáveis na memória.

SEÇÃO 2
DECLARAÇÃO DE VARIÁVEIS NA MEMÓRIA

Como já foi estudado, as variáveis possuem dois vínculos principais, o **dado** ao qual ela está relacionada e o **endereço** na memória do computador, onde efetivamente a informação está armazenada. Com base nesta situação, é importante que o programador tenha uma ideia de como os dados das variáveis ficam armazenados e alocados na memória do computador, mesmo que de forma abstrata. O processo se inicia com o compilador ou interpretador, que montam uma tabela de símbolos, relacionando as variáveis e os respectivos endereços de cada uma, com base nos tipos de dados. O exemplo, figura 13, a seguir, ilustra essa situação, tomando por base um endereço inicial hipotético, zero em decimal.

É oportuno relembrar que os endereços não são acessados pelo sistema decimal, mas hexadecimal ou binário, por isso é hipotético.

Nome simbólico	Posição inicial	Tipo de dado
NOME_CLIENTE	0	LITERAL[30]
IDADE_CLIENTE	30	INTEIRO
SALARIO_CLIENTE	32	REAL
TEM_FILHOS_CLIENTE	36	LÓGICO

Figura 13: Mapeamento de variáveis na memória.

Apesar do exemplo anteriormente exibido ser uma abstração, auxilia o programador a entender como os dados ficam distribuídos na memória do computador e nas situações futuras, quando for trabalhar com **variáveis compostas homogêneas**, esse conhecimento será ainda mais útil.

Variáveis Compostas Homogêneas fazem parte do conteúdo programático da disciplina Algoritmo e Programação II, mas é interessante conhecer o conceito do que são: "Variáveis Compostas Homogêneas correspondem a um conjunto de variáveis do mesmo tipo, referenciáveis pelo mesmo nome e diferenciáveis entre si por meio de sua posição dentro do conjunto (índice)".

Siga em frente!

Voltando a análise para o exemplo recentemente citado, parte-se do princípio que cada endereço aponta para um byte e a alocação dos dados na memória se inicia no endereço zero, < 0 >, e sempre pela ordem em que as variáveis são declaradas, conforme o detalhamento a seguir:

- **NOME_CLIENTE** – é a primeira variável a ter o espaço alocado na memória e tem o seu primeiro byte apontado pelo endereço inicial < 0 > e final < 29 >. Percebe-se que foram alocados 30 bytes para a variável em referência, a razão é que na declaração foi definido que ela é um LITERAL [30] com tamanho de 30 bytes.
- **IDADE_CLIENTE** – os endereços alocados para esta variável são apenas dois, endereços < 30 > e o < 31 >, pois é uma variável declarada como numérica e do tipo inteiro, que requer apenas 2 bytes para ser armazenada.
- **SALARIO_CLIENTE** – terceira variável a ter os endereços alocados, iniciando-se em < 32 > e encerrado em < 35 >, já que a variável anterior, IDADE_CLIENTE, teve seu último endereço alocado em < 31 >. Como esta variável é numérica e do tipo real necessita de 4 bytes para ser armazenada.
- **TEM_FILHOS_CLIENTE** – por ser uma variável do tipo lógico, requer apenas 1 byte, o endereço alocado para ela é o < 36 >, após o último endereço da variável anterior SALARIO_CLIENTE.

Após a conclusão dos estudos desta unidade, é hora de fazer alguns exercícios. Ao final da atividade, compare suas respostas com as apresentadas no final deste livro. Não havendo coincidência nas respostas, refaça com atenção a atividade.

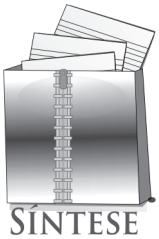
→ 1) Na lista a seguir, assinale com (V) os nomes de variáveis VÁLIDOS e com (N) os INVÁLIDOS:

( ) abc	( ) 3abc	( ) a
( ) 123a	( ) _a	( ) acd1
( ) _	( ) Aa	( ) 1
( ) A123	( ) _1	( ) A0123
( ) a123	( ) _a123	( ) b312
( ) AB CDE	( ) guarda-chuva	( ) etc.

→ 2) Dada a declaração de variáveis a seguir, preencha a tabela de símbolos correspondente, como seria realizado por um compilador, considerando o armazenamento na memória do computador, para cada tipo de dado:

```
VAR        X, Y      : INTEIRO;
           NOME,  PROFISSAO: LITERAL[20];
           RUA       : LITERAL[30];
           NUMERO : INTEIRO;
           RENDA : REAL;
```

Nome simbólico	Posição inicial	Tipo de dado



SÍNTESE

Neste momento, ressalta-se algumas características importantes para consolidar esse assunto e, também outros detalhes, em relação ao que foi exposto sobre mapeamento de variáveis na memória:

- A ordem de alocação das variáveis na memória do computador, sempre ocorre conforme a ordem descrita na declaração delas e são distribuídas sequencialmente na memória, uma após a outra.
- O espaço alocado para cada variável é de acordo com o tipo de dado associado a cada uma delas.
- Do ponto de vista algorítmico: dados literais requerem um 1 byte para cada caractere; dados lógicos consomem apenas 1 byte; dados numéricos do tipo inteiro 2 bytes e do tipo real 4 bytes. Lembrando que, quando se trata de linguagens de programação, os dados numéricos podem ser subdivididos em outros subtipos.
- Os locais da memória do computador onde os dados são alocados, bem como os endereços, têm seu controle e gerenciamento realizados pelo sistema operacional e compilador/interpretador.

Nesta unidade, você aprendeu também como as variáveis devem ser declaradas e que além do uso da palavra-reservada **Var** é necessário preocupar-se com os tipos associados aos dados, bem como em relação à informação que a variável armazenará. Outro detalhe é com relação a nomenclatura das variáveis, que não devem conter símbolos especiais, exceto sublinha, devem começar necessariamente com uma letra, não usar acentuação e nem cedilha e devem procurar expressar o respectivo conteúdo.

Para o programador de computadores, o conhecimento sobre variáveis é indispensável e nesta unidade você estudou os principais detalhes a respeito delas. Gradativamente, conforme a disciplina avança, esses conhecimentos serão necessários e outros serão incorporados conforme o aprofundamento na área. A próxima unidade, que aborda **Expressões**, faz parte desse aprofundamento. Sempre lembrando, havendo dúvidas, não prossiga, reestude a unidade.


ANOTAÇÕES

UNIDADE VIII

Expressões

Objetivos de aprendizagem

- Saber como desenvolver os diferentes tipos de expressões.
- Conhecer os diversos tipos de operandos e operadores.
- Aprender a avaliar as expressões.

Roteiro de estudos

- SEÇÃO 1 – Expressões Aritméticas
- SEÇÃO 2 – Expressões Lógicas
- SEÇÃO 3 – Expressões relacionais
- SEÇÃO 4 – Expressões Literais
- SEÇÃO 5 – Avaliação de Expressões

UAB UNIVERSIDADE ABERTA DO BRASIL

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

As expressões ou fórmulas no contexto da informática têm um significado similar ao da matemática, no que se refere ao fato de que variáveis e constantes numéricas relacionam-se por meio de operadores, e a partir da avaliação obtém-se um valor como resultado.

Do ponto de vista geral, pode-se resumir que uma expressão é uma combinação de **operandos** e **operadores**, e que, uma vez avaliada, resulta num valor, por exemplo: fórmula de cálculo da área de um triângulo.

ÁREA = 0,5 * (B * H);

onde:

B, H → variáveis que respectivamente correspondem à base e altura;

0,5 → constante;

***** → operador de multiplicação.

Os **operadores** são elementos funcionais que atuam sobre os **operandos** e de acordo com o número deles, podem ser classificados em:

- **Binários** – quando atuam sobre dois operandos, por exemplo os operadores básicos das operações aritméticas: soma, subtração, multiplicação e divisão;
- **Unários** – quando atuam sobre um único operando, exemplo o sinal negativo (–) antes de um número, cuja função pode ser a inversão do sinal.

Computacionalmente, as expressões são classificadas de acordo com o tipo de valor resultante da avaliação, que são: **aritméticas**, **lógicas**, **relacionais** e **literais**. Essa situação ocorre em função de que os **operandos** estão diretamente relacionados aos tipos de dados, que podem ser: **numéricos, lógicos e literais**.

SEÇÃO 1
EXPRESSÕES ARITMÉTICAS

São aquelas em que o resultado da avaliação é do tipo numérico e apenas operadores aritméticos são permitidos, bem como, os operandos devem ser variáveis e/ou constantes numéricas.

No caso das variáveis e/ou constantes numéricas, os tipos podem ser **inteiro** ou **real** e ambos podem ser combinados. Para o caso de uma expressão com tipos inteiros o resultado da avaliação também será do tipo inteiro, quando os dois tipos são combinados o resultado da avaliação é do tipo real, relembrando, esse é o caso da coerção.

As expressões aritméticas possuem como operadores os mesmos da matemática, assim como a precedência (prioridade) em que as operações devem ocorrer. A figura 14 ilustra os operadores aritméticos, com as respectivas precedências (em ordem crescente), tipos e descrições.

Operador	Tipo	Operação	Precedência (ordem)
+	Unário	Manutenção de sinal	1ª
–	Unário	Inversão de sinal	1ª
** ou ^	Binário	Exponenciação Potenciação	2ª
SQRT	Unário	Raiz Quadrada	2ª
*	Binário	Multiplicação	3ª
/	Binário	Divisão	3ª
MOD	Binário	Resto de divisão	3ª
+	Binário	Adição	4ª
–	Binário	Subtração	4ª

Figura 14: Operadores aritméticos.

Com relação à precedência, é importante destacar que representa a ordem de prioridade em que os operadores devem ser avaliados, iniciando-se pela 1ª, ou se o operador correspondente não ocorrer na expressão, iniciar da menor para a maior.

EXEMPLO

Exemplo de uma expressão aritmética resolvida:

Considerando X, Y e Z como variáveis inteiras, com os valores 5, 11 e 20, respectivamente, obter o resultado da expressão a seguir:

X * Y – Z =
5 * 11 – 20 =
55 – 20 =
= 35

Neste exemplo simples, você deve ter percebido que inicialmente foi realizada a multiplicação (*) que tem prioridade sobre a subtração (–), a qual foi realizada por último.

Como visto, as expressões aritméticas possuem como resultado um valor numérico, agora o foco da abordagem são as expressões que possuem como resultado um valor lógico.

SEÇÃO 2
EXPRESSÕES LÓGICAS

São aquelas em que o resultado da avaliação da expressão é um valor lógico (.**V**. ou .**F**.). Neste caso, os operadores lógicos são baseados na álgebra Booleana, que possui três operações básicas: .E. (conjunção) – .OU. (disjunção) – .NÃO. (negação). Na literatura podem ser encontradas formas diferentes de representar os operadores lógicos, tais como, .AND. – .OR. – .NOT.. A figura 15 exibe os operadores lógicos em referência, com as respectivas precedências (em ordem crescente), tipos e descrições.

Operador	Tipo	Operação	Precedência (ordem)
.NÃO.	Unário	Negação	**1ª**
.E.	Binário	Conjunção	**2ª**
.OU.	Binário	Disjunção	**3ª**

Figura 15: Operadores lógicos.

Apesar de ser uma classe específica de expressões, as operações lógicas ocorrem de forma similar às da aritmética, em que os operandos podem ser variáveis e/ou constantes do tipo lógico, que são avaliados conforme a respectiva precedência e tipo.

As particularidades das expressões lógicas estão nas operações e nos respectivos operadores, também chamados de conectivos lógicos. Por meio de uma tabela, intitulada tabela-verdade, pode-se determinar os valores lógicos possíveis de uma fórmula com valores binários e os operadores lógicos em questão.

Em síntese, a tabela-verdade possui todas as possibilidades combinatórias entre os valores .V. ou .F. com os respectivos operadores lógicos. O número de combinações que as variáveis podem assumir pode ser calculado por 2^n, onde n é o número de variáveis de entrada.

A figura 16 ilustra uma tabela-verdade com os operadores lógicos descritos na figura 15, e com os valores lógicos representados por .V. (Verdadeiro) e .F. (Falso).

A	B	.NÃO. A	.NÃO. B	A .OU. B	A .E. B
.V.	.V.	.F.	.F.	.V.	.V.
.V.	.F.	.F.	.V.	.V.	.F.
.F.	.V.	.V.	.F.	.V.	.F.
.F.	.F.	.V.	.V.	.F.	.F.

Figura 16: Tabela-verdade – .V./.F.

Note-se, nesta tabela, que no cabeçalho, primeira linha – primeira e segunda colunas há duas variáveis de entrada A e B respectivamente, e nas demais colunas estão as operações lógicas correspondentes, nas demais linhas estão os possíveis valores que as variáveis podem receber, bem como o resultado das operações.

A seguir, figura 17, está ilustrada uma tabela-verdade com as mesmas operações da anterior, figura 16, porém os valores lógicos estão representados em binário, < 1 > e < 0 >, representando *Verdadeiro* e *Falso* respectivamente para cada valor.

A	B	.NÃO. A	.NÃO. B	A .OU. B	A .E. B
1	1	0	0	1	1
1	0	0	1	1	0
0	1	1	0	1	0
0	0	1	1	0	0

Figura 17: Tabela-verdade – 0/1.

Comparando as duas tabelas exibidas anteriormente, percebe-se que os resultados são coincidentes, porém, a tabela-verdade com os valores lógicos representados por < 1 > e < 0 >, oferece um modo que ajuda a memorizar como a tabela é elaborada. Veja a seguir, as explicações para cada um dos operadores.

- **Operação .OU. – disjunção – adição lógica** – considerar a disjunção como adição facilita o entendimento, pois a operação .OU. resulta 1 se, pelo menos, uma das variáveis de entrada possuir valor 1, e resulta 0 nos demais casos. Como regra, a soma de qualquer valor com zero, resultado no maior valor, como na álgebra booleana o maior valor é 1, portanto mesmo 1 + 1 = 1. Pode-se observar as quatro combinações possíveis, para duas variáveis de entrada:

$$1 + 1 = 1$$
$$1 + 0 = 1$$
$$0 + 1 = 1$$
$$0 + 0 = 0$$

- **Operação .E. – conjunção – multiplicação lógica** – a disjunção pode ser definida da seguinte forma, se, pelo menos, uma das variáveis possuir o valor 0, o resultado será 0, e resultará 1 se todas as variáveis estiverem com o valor 1. Como regra, sabe-se também que, qualquer valor multiplicado por 0, resulta 0. A seguir, uma lista com as quatro combinações possíveis, para duas variáveis de entrada:

$$1 * 1 = 1$$
$$1 * 0 = 0$$
$$0 * 1 = 0$$
$$0 * 0 = 0$$

É fundamental salientar que para as duas regras anteriormente descritas, vale qualquer número de variáveis de entrada, evidentemente, o número de combinações, operações, muda, pois como o número de operações é calculado por $2^{\text{número de variáveis}}$, por exemplo, se na expressão forem três variáveis entrada, o número de combinações é $2^3 = 8$.

- **Operação .NÃO. – negação – inversão – complementação** – é a operação em que o resultado é simplesmente o valor complementar ao que a variável apresenta. Como uma variável Booleana poder assumir somente um entre dois valores, o valor complementar será 1 se a variável possuir 0 e será 0 se a variável possui valor 1.

Por padrão, nesta disciplina, para os exercícios com **algoritmos** serão adotados os símbolos .V. (Verdadeiro) e .F. (Falso), porém, na prática e na literatura, há símbolos diferentes, bem como, linguagens de programação que utilizam < *True* > e < *False* > e também os valores 1 e 0. Evidentemente quando o programador for codificar um algoritmo em alguma linguagem de programação, previamente deve buscar saber como os valores lógicos são representados naquele ambiente de programação em que estiver trabalhando.

Veja, a seguir, um exemplo de expressão lógica, resolvida.

Considerando A, B e C como variáveis lógicas, com os valores, .V. , .F. e .V., respectivamente, obter o resultado da expressão:

B .E. A .OU. C =
.F. .E. .V. .OU. .V. =
.F. .OU. .V. =
= .V.

Observe que no exemplo acima, inicialmente foi realizada a operação de conjunção (.E.) que tem prioridade sobre a disjunção (.OU.), que foi realizada por último.

Na próxima seção, você vai aprender sobre expressões em que o resultado também é um valor lógico, que são as relacionais.

SEÇÃO 3
EXPRESSÕES RELACIONADAS

As **expressões relacionais**, ou simplesmente **relações**, são aquelas em que são feitas comparações entre variáveis de um mesmo tipo (numéricas, lógicas ou literais), cujo resultado é sempre um valor lógico. Em uma mesma expressão podem ter variáveis e/ou constantes de tipos diferentes, mas as comparações devem ser com operandos do mesmo tipo. Os operadores relacionais, tipos de operações e as precedências (em ordem crescente) estão descritas na figura 18, a seguir.

Operador	Comparação	Precedência (ordem)
<	Menor	1ª
<=	Menor ou igual	1ª
>	Maior	1ª
>=	Maior ou igual	1ª
=	Igual	2ª
<>	Diferente	2ª

Figura 18: Operadores relacionais.

Em relação à simbologia dos operadores relacionais, assim como já foi destacado com outros operadores, podem haver diferenças tanto na literatura como em algumas linguagens de programação. E também para padronizar os procedimentos da disciplina, na construção de algoritmos os símbolos dos operadores relacionais serão utilizados os descritos na figura 18.

Veja alguns exemplos de expressões relacionais.

Suponha que A, B e C sejam variáveis do tipo inteiro, podem ser formuladas expressões como:

A = 2
A > B
C <= B

Na expressão A = 2, compara-se o valor da variável A com a constante 2. O resultado desta comparação pode ser valor lógico .V. se o valor de A for 2, caso contrário, o resultado é .F.. As demais comparações, A > B e C <=B, terão os resultados .V. ou .F. conforme os valores atribuídos as variáveis A, B e C. Por exemplo, caso o valor de A seja 2 e o de B seja 3, o resultado da expressão A > B é .F., pois 2 não é maior do que 3.

As relações podem envolver expressões aritméticas, em casos como os descritos a seguir, considerando A, B e C como variáveis do tipo inteiro:

A > (B * C)
C <= (5 + A)

Na relação A > (B * C), compara-se o valor de A com o valor resultante da expressão aritmética (B * C). No segundo caso C <= (5 + A), está sendo comparado o valor de C com o valor resultante da expressão aritmética com constante (5 + A). Evidentemente o resultado de cada expressão sempre será .V. ou .F., dependendo dos valores atribuídos a cada variável.

Há situações em que a relação pode envolver mais expressões aritméticas, como o exemplo a seguir:

(B * C) <= (C + 10)

No exemplo anterior, compara-se o valor da expressão aritmética do lado esquerdo do operador relacional com o valor da expressão aritmética do lado direito. Uma particularidade da relação em destaque é o fato de que ela poderia ser escrita sem os parênteses, como exemplificado:

B * C <= C + 10

Em ambos os casos, a mesma expressão relacional, com ou sem parênteses, com o mesmo valor das variáveis, o resultado seria o mesmo nas duas relações. Pois, inicialmente as subexpressões aritméticas têm prioridade sobre as relacionais, que devem ser resolvidas por último.

Veja agora exemplos com variáveis do tipo caractere e literais.

Suponha que A e B são duas variáveis do tipo caractere, com os valores ‘i’ e ‘v’, respectivamente, e as expressões relacionais a seguir.

A = B
A < B

Na relação, A = B, é evidente que o resultado é .F., pois 'i' e 'v' são duas letras diferentes. Na expressão, A < B, de certo modo, também fica evidente o resultado .V., uma vez que 'i' é menor do que 'v' do ponto de vista da ordem alfabética. Porém, é importante você saber que o computador distingue a ordem das letras pelos códigos da tabela ASC. Observe na figura 11 que a letra 'i' possui o código 105 e o 'v' 118. Na tabela ASC são encontrados, tanto o alfabeto minúsculo (códigos ASC 97 a 122) como o maiúsculo (códigos ASC 65 a 90). Ambos os alfabetos estão em ordem alfabética, como consequência da codificação numérica, possibilitando ao computador fazer as distinções.

Veja outros exemplos, considere que A e B são duas variáveis do tipo literal, com os conteúdos 'informática' e 'computador', respectivamente, e as expressões relacionais:

A > B
A < B

Na relação A > B, possivelmente você deve ter percebido que o resultado é .V., pois observando a ordem alfabética, a palavra 'informática' pelo simples fato de iniciar em 'i', já demonstra estar após a letra 'c'. Consequentemente, na relação A < B o resultado é o oposto, .F., pois A (informática) não esta antes de B (computador), na ordem alfabética.

Nesse contexto, a ordem **alfabética** também pode ser chamada de ordem **lexicográfica**, que corresponde a ordem em que as palavras em um dicionário ficam dispostas. Essa ordem é obtida pelo computador, comparando as cadeias de caracteres com base nos códigos ASC de cada letra, e assim são determinados os resultados das comparações.

A utilização de relações com dados lógicos, basicamente fica restrita aos operadores de igualdade (=) e diferença (<>), pois, não faz sentido determinar se o valor lógico .V. é menor, maior, menor ou igual ou maior ou igual a .F..

A última categoria de expressões no contexto algorítmico e da programação de computadores são as expressões literais.

SEÇÃO 4
EXPRESSÕES LITERAIS

Nas expressões literais, os operandos são literais e o resultado da avaliação é um valor literal, ou seja, todos os elementos devem ser do mesmo tipo.

Pode até parecer estranho realizar operações com literais e no formato de expressões, até porque, na maioria das linguagens de programação elas ocorrem com funções específicas. Porém, há casos, como no Pascal em que o operador aritmético de adição (**+**) é utilizado para a concatenação de literais, onde se toma duas literais e acrescenta-se (concatena-se) a segunda no final da primeira, conforme exemplo:

'OSMAR' + 'MOTA' = 'OSMARMOTA'

As operações mais comuns com expressões literais, onde o resultado é o tipo literal, são: concatenação e cópia. Porém, há outras operações com literais, como visto na seção anterior, em que o resultado é um valor lógico, quando dois literais são comparados e esse tipo de operação é classificada como uma operação relacional. Há ainda, o caso em que se calcula o tamanho (comprimento) de um literal, nessa situação o resultado é um número inteiro. Por razões como as citadas anteriormente, que o tratamento de literais difere entre as diversas linguagens de programação. Por exemplo, em linguagem C a concatenação se faz com a função *strcat(string1,string2)*, em que o conteúdo de *string2* é acrescido ao final de *string1*.

Continuando, a próxima seção, em parte, comenta aquilo que você já deve ter estudado no ensino médio, porém, com algumas adaptações para as expressões e fórmulas no contexto da informática.

SEÇÃO 5
AVALIAÇÃO DE EXPRESSÕES

Ao avaliar expressões que apresentam apenas um único operador elas podem ser avaliadas diretamente, sem a necessidade de observar precedência. Entretanto, quando as expressões são mais complexas, com mais operandos e operadores numa mesma fórmula é necessária que a avaliação seja realizada passo a passo, avaliando um operador por vez. A ordem dos passos é definida de acordo com o formato geral da expressão, levando-se em considerando a prioridade (precedência) de avaliação de seus operadores e a existência ou não de parênteses.

Portanto, pode-se resumir a avaliação das expressões em algumas regras:

- Os parênteses usados em expressões, tem prioridade maior em relação aos demais operadores, situação que força a avaliação da(s) subexpressão(ões) dos parênteses mais internos.
- Deve-se observar a prioridade (ordem) de avaliação dos operadores, conforme mostrado anteriormente nos quadros, figuras, de cada um dos tipos de expressões. Se houver empate com relação à precedência, então a avaliação se faz considerando-se a expressão da esquerda para direita.
- Como podem ocorrer expressões com os quatro grupos de operadores, aritmético, lógico, literal e relacional, há uma prioridade de avaliação entre eles: os aritméticos e literais devem ser avaliados primeiro; a seguir, são avaliadas as subexpressões com operadores relacionais e, por último, os operadores lógicos, conforme descrito na figura 19.

Operação	Precedência (ordem)
Operadores aritméticos/literais	**1ª**
Operadores relacionais	**2ª**
Operadores lógicos	**3ª**

Figura 19: Prioridade (precedência) entre todos os operadores

Veja um exemplo de expressão, resolvida, com operadores aritméticos, relacionais e lógicos.

Considerando X e Y como variáveis reais, A e B variáveis lógicas, com os respectivos valores: X = 2.0, Y = 3.0, A = .V. e B = .F.. Obter o resultado da expressão:

B .E. (A .OU. X <> Y / 2) =
F .E. (V .OU. 2 <> 3/2) =
F .E. (V .OU. 2 <> 1.5) =
F .E. (V .OU. V) =
F .E. V =
= F

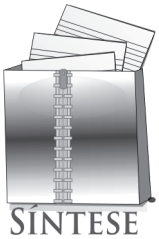
Para ajudar a consolidar os conhecimentos desta seção, resolva os exercícios propostos a seguir.

→ Sejam A e B variáveis lógicas, X e Y variáveis reais, e R, S e T variáveis literais, com os respectivos valores:

A = .V., B = .F.;
X = 2.5, Y = 5.0;
R = 'JOSÉ', S = 'JOÃO', T = 'JOÃOZINHO';

→ Avalie as expressões lógicas/relacionais do quadro a seguir, com base nos valores anteriormente exibidos:

Expressão	Resultado
A .OU. B	
A .E. B	
.NÃO. A	
X = Y	
X = (Y/2)	
R = S	
S = T	
R <> S	
R > S	
S > T	
((A .OU. B) .OU. (X = Y) .OU. (S = T))	



SÍNTESE

É possível que você esteja se questionando, a respeito do fato de que, em diversos momentos deste livro, fala-se de diferenças entre as formas de usar determinados comandos, operadores e outras situações, tanto na literatura, do ponto de vista algorítmico, ou entre as diversas linguagens de programação. Fique despreocupado, pois com relação ao formato dos algoritmos propostos nesta disciplina, mesmo que sejam diferentes das formas encontradas na literatura, os algoritmos são universais e passíveis de serem codificados em qualquer linguagem de programação. Evidentemente, aquelas voltadas às mesmas finalidades propostas no algoritmo em desenvolvimento, como: sistemas comerciais, científicos, para web, etc.

No que tange as linguagens de programação, também fique tranquilo, pois nesta disciplina você vai aprender a usar a **Linguagem de Programação Pascal**, que é uma linguagem estruturada com os recursos similares a qualquer outra. De modo geral, o que difere as linguagens entre si, são as sintaxes dos comandos, a finalidade e os ambientes de programação. Neste quesito, agora você conhecerá um pouco do Pascalzim e também da Linguagem de Programação Pascal.

ANOTAÇÕES

UNIDADE IX

Pascalzim

UNIVERSIDADE ABERTA DO BRASIL
UAB

PARA INÍCIO DE CONVERSA

Caro (a) Aluno (a),

A linguagem Pascal foi desenvolvida pelo professor Niklaus Wirth no ano de 1972, na cidade de Genebra, Suíça. O nome da linguagem foi uma homenagem ao filósofo e matemático Blaise Pascal (1623-1662), inventor da primeira calculadora mecânica. O desejo de Wirth era dispor, para o ensino de programação, de uma nova linguagem que fosse simples, coerente e capaz de incentivar a confecção de programas claros e facilmente legíveis, favorecendo a utilização de boas técnicas de programação.

A linguagem Pascal se tornou amplamente conhecida e utilizada com o lançamento da mundialmente famosa série de compiladores Turbo Pascal pela Borland, em 1985, devido uma combinação de simplicidade e poder de processamento.

O compilador Pascalzim, desenvolvido no Departamento de Ciências da Computação da Universidade de Brasília, é fruto de muitos anos de pesquisa e trabalho na área de tradutores e linguagens de programação. Adotado como ferramenta de apoio ao ensino e aprendizagem da linguagem Pascal pelos alunos matriculados no curso de Introdução à Ciência da Computação daquela instituição, o compilador foi utilizado no primeiro semestre do ano 2000.

No segundo semestre de 2001, o Pascalzim foi utilizado pelos alunos do Instituto de Ensino Superior de Brasília - IESB para o aprendizado da disciplina Algoritmos e Programação Estruturada.

O compilador Pascalzim implementa um subconjunto da linguagem Pascal e contém as estruturas de dados, funções e comandos mais utilizados por iniciantes no estudo dessa linguagem. O arquivo de ajuda que acompanha o produto especifica as instruções suportadas.

O ambiente de programação Pascalzim foi concebido com finalidade meramente educacional e sua distribuição é livre. Os ambientes de programação são chamados também de IDE (*Integrated Development Environment*) ou ainda Ambiente de Desenvolvimento Integrado.

Nesta unidade, serão abordados apenas os detalhes básicos do Pascalzim, suficientes para o desenvolvimento das atividades práticas propostas neste livro. Demais assuntos relacionados, devem ser consultados no material disponível na área do aluno, bem como, o respectivo instalador do Pascalzim, o qual, caso possa, o ideal é você instalar em seu próprio computador.

O processo de instalação do Pascalzim resume-se em baixar o arquivo compactado < pascalzim603.zip > da área do aluno, copiá-lo na unidade < Disco Local (C:) > do seu computador, descompactar o arquivo previamente baixado, após a descompactação no disco em referência será criada uma pasta pascalzim , pelo *windows explorer* acesse o conteúdo desta pasta e crie um atalho na área de trabalho do arquivo Pzim. Quando o respectivo atalho é executado, a janela ilustrada na figura 20 é exibida, que corresponde a um editor de textos, para escrever os programas em Pascal. Na área de edição, o ambiente automaticamente cria um código-fonte resumido, que corresponde ao formato básico de um programa em Pascal. Esse assunto será estudado na próxima unidade, juntamente com os detalhamentos sobre algoritmos.

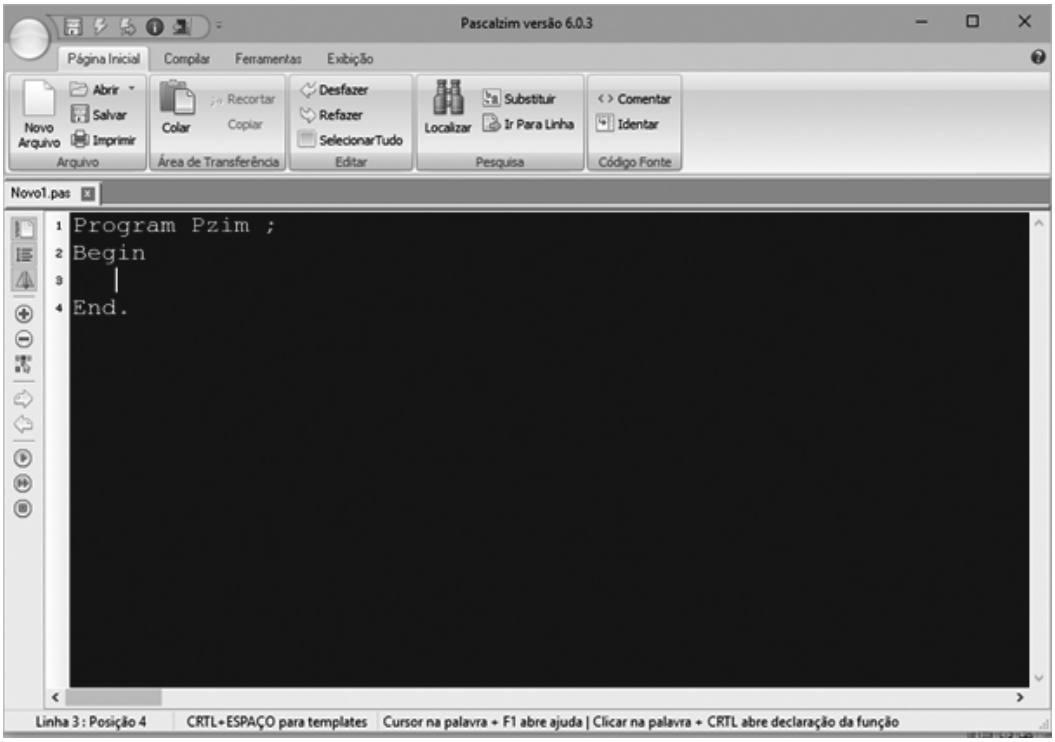


Figura 20: Tela inicial do Pascalzim

Observe que o editor possui na aba *Arquivo*, as opções necessárias para: *Novo Arquivo, Abrir, Salvar* e *Imprimir*. Levando-se em consideração que os demais comandos para edição de textos são inerentes ao editor em questão.

Quando você estiver editando os seus arquivos-fonte, escolha uma pasta em seu computador para salvar os respectivos programas, procurando utilizar nomes de modo que posteriormente saiba a que correspondem. Por padrão, os arquivos-fonte salvos pelo Pascalzim possuem a extensão < .pas >, por exemplo, um programa chamado < Tabuada >, aparecerá no *windows explorer* como Tabuada.pas .

Na figura 21, note no canto superior esquerdo que há o ícone o qual possibilita a compilação e execução de um programa em código-fonte, bem como, para a mesma operação possui como tecla de atalho o < F9 >. O ícone em questão, quando acionado faz a compilação do código-fonte que estiver ativo e posteriormente o executa, desde que esteja livre de erros. Situação que também será estudada na próxima unidade.

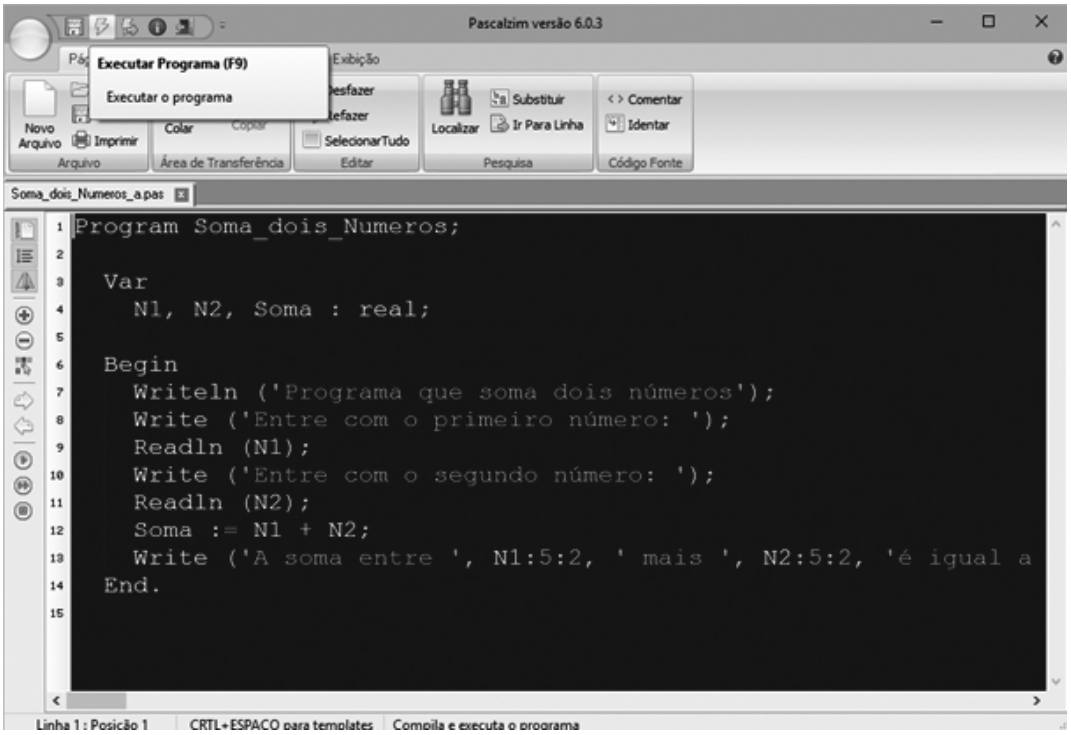
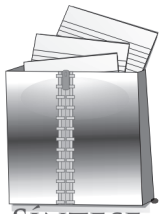


Figura 21: Tela inicial do Pascalzim - compilação/execução de programa



SÍNTESE

O que foi abordado nesta unidade é suficiente para que você consiga editar um programa e o executar/compilar. Para aprender demais detalhes, você deve baixar as apostilas disponíveis na área do aluno e estudá-las, porém, como o assunto é amplo, a princípio procure focar os temas que estão relacionados aos que estão sendo estudados em cada momento da disciplina.

Após você ter estudado as unidades anteriores, já está habilitado a dar os primeiros passos na escrita de algoritmos e os respectivos programas. Contudo, é imprescindível que todos os exemplos em Pascal que forem apresentados, neste livro, devem ser implementados em Pascalzim e executados. Caso contrário, não é possível acompanhar o que está sendo ensinado, uma vez que, nessa etapa dos estudos, praticar é primordial.

Os estudos serão conduzidos passo a passo, para que todos os detalhes sejam efetivamente aprendidos e entendidos. Por outro lado, à medida que você vai se exercitando na construção de algoritmos/programas, vai percebendo que as estruturas de programação, resumem-se em: sequenciais, de decisão e repetição.


ANOTAÇÕES

Estruturas de Controle do Fluxo de Execução

UNIDADE X

Roteiro de estudos

- SEÇÃO 1- Estrutura Sequencial
- SEÇÃO 2- Estrutura de Decisão
- SEÇÃO 3- Estruturas de Repetição

PARA INÍCIO DE CONVERSA

Nesta unidade, como o próprio título expressa, você vai estudar as estruturas de programação que permitem definir a sequência de execução dos comandos de um programa, em função das instruções do mesmo, sendo classificadas em:

- **Estruturas sequenciais**
- **Estruturas de decisão**
- **Estruturas de repetição**

Um conceito novo que precisa estar sempre em mente, a partir deste momento, é o *comando composto* ou *bloco*, que corresponde a um conjunto de dois ou mais comandos (ou instruções) simples, como atribuições e instruções primitivas de entrada e/ou saída.

Outro detalhe que também deve ser sempre lembrado, quando houver referência a *algoritmo*, serão exemplos em Portugol e quando forem programas em linguagem de programação Pascal com a sintaxe do Pascalzim, a referência será apenas *em Pascal*.

A primeira estrutura de programação a ser estudada é a Estrutura Sequencial e nas explanações dela, e também nas demais, você encontrará os respectivos exemplos de algoritmos e os códigos-fonte em Pascal, com as respectivas explicações e comparações entre eles.

SEÇÃO 1
ESTRUTURA SEQUENCIAL

Nesta estrutura de programação, os comandos de um algoritmo/programa são executados numa sequência preestabelecida. Cada comando é executado somente após o término do comando anterior, ou seja, de forma sequencial, de cima para baixo e na ordem em que foram definidos.

Analise o exemplo de algoritmo a seguir, com estrutura sequencial de execução.


```
Algoritmo Soma_dois_numeros;

    Var
        N1, N2, Soma : real;

    Início
        Leia N1, N2;
        Soma := N1 + N2;
        Escreva Soma;
    Fim.
```

No algoritmo exibido anteriormente é fácil perceber que é uma estrutura sequencial, pois, basta fazer uma leitura do texto para inferir que os comandos devem ser executados na ordem que estão descritos, para que o algoritmo atinja o objetivo; que é efetuar a soma entre dois números e exibir o resultado.

Agora, observe o algoritmo em referência codificado em Pascal. Compare os dois códigos.


```
Program Soma_dois_numeros;

 Var
     N1, N2, Soma : real;

 Begin
     read (N1, N2);
     Soma := N1 + N2;
     write (Soma);
 End.
```

Possivelmente você já deve ter detectado as diferenças e, pode-se dizer que não são tão significativas, a saber:

- no caso das palavras-reservadas, a única diferença é o fato de estarem em inglês;
- na sintaxe do *Read*, que equivale ao Leia, as variáveis estão entre parênteses, o mesmo ocorre com o *Write* – Escreva.

Você já sabe que quando se escreve algoritmos, há uma certa flexibilidade com relação à sintaxe dos comandos. Diante disto, sobre o uso dos parênteses para os argumentos dos comandos *Read* e *Write*, pode-se a partir de agora adotar o uso parênteses também para o **Leia** e **Escreva**, do mesmo modo que em Pascal. Essa prática pode evitar erros futuros na codificação dos programas.

Ainda no quesito flexibilidade na escrita dos algoritmos, geralmente quando se escreve algoritmos, procura-se ser o mais sucinto possível, ou seja, não preocupar-se com a interface do programa para o usuário, e isto é válido, pois nesta fase do desenvolvimento o foco é a solução do problema, principalmente do ponto de vista lógico. Porém, quando um algoritmo é codificado em uma linguagem de programação, o ideal e incluir mensagens na tela do computador, para que durante a execução do programa o usuário saiba o que o programa requer em situações de entrada de dados e também quando exibe os respectivos resultados.

Agora, observe o mesmo programa em Pascal anteriormente exemplificado, com algumas melhorias na interface, com mensagens que o tornam mais agradável e elucidativo para o usuário.

EXEMPLO

```
Program Soma_dois_numeros;

  Var
    N1, N2, Soma : real;

  Begin
    writeln ('Programa que soma dois números');
    write ('Entre com o primeiro número: ');
    readln (N1);
    write ('Entre com o segundo número: ');
    readln (N2);
    Soma := N1 + N2;
    write ('A soma entre ', N1:5:2, ' mais ', N2:5:2, 'é igual a ', Soma:5:2);
  End.
```

Além da execução do programa em destaque no Pascalzim, é fundamental fazer a comparação entre as três versões dele, uma em algoritmo e as duas em Pascal, observando atentamente os detalhes descritos:

- dois comandos novos aparecem, *Writeln* e *Readln*, porém, ambos desempenham as mesmas funções do *Write* e *Read*, respectivamente. A única diferença é que após a execução deles o cursor fica na mesma linha e com o < *ln* > o cursor vai para a linha de seguinte;
- outra diferença é a utilização de dados literais, *strings*, acompanhados de *write* e *writeln*, por exemplo, < Writeln ('Programa que soma dois números'); >. Observe que a delimitação das *strings* é aspas simples - plicas < ' >, e conforme já comentado anteriormente, há outros ambientes de programação que utilizam aspas duplas < " >. Portanto, do ponto de vista do algoritmo, ambas as formas são válidas, porém na codificação em Pascal obrigatoriamente terão que ser aspas simples, sendo assim, pode-se, a partir de agora manter esta sintaxe para os algoritmos;
- outro detalhe em relação ao uso do *write* e *strings*, é o fato de ter sido usado acentuação na palavra "números". Conforme já

mencionado, não se pode usar acentuação em nomes de variáveis, mas em literais, *strings*, é permitido, uma vez que são cadeias de caracteres alfanuméricos;

- com relação ao *Readln*, nota-se que em vez de um *Read* para as duas variáveis, N1 e N2, foram utilizados dois comandos, separando a entrada de cada uma das variáveis, o que facilitou a execução para o usuário e a exibição das mensagens;
- na linha com as seguintes instruções < Write ('A soma entre ', N1:5:2, ' mais ', N2:5:2, 'é igual a ', Soma:5:2); >, os números ao lado de cada variável, por exemplo < Soma:5:2 >, tem a finalidade de formatar a exibição do valor contido na variável, neste caso o 5 representa o número de dígitos antes do separador decimal e o 2 os após o separador decimal;
- indentação – ao observar todos os códigos ilustrados até agora, sejam algoritmos ou programas em Pascal, você deve ter notado que possuem recuos (tabulações) nas linhas, cujos recuos definem as estruturas que o algoritmo e o programa possuem. Por exemplo, as palavras *Begin* e *End*, têm o mesmo recuo e as linhas que estão entre essas palavras, um recuo maior, destacando que pertencem aquela estrutura. O objetivo da indentação é organizar o código, para que o programador possa identificar e entender a construção das diferentes estruturas de um algoritmo ou programa, visando a sua manutenção. À medida que os programas vão se tornando mais complexos, fica cada vez mais difícil a manutenção e a leitura do código, a indentação auxilia nesse trabalho. Quando você estiver codificando os programas no Pascalzim, vai perceber que o editor dele já é dotado desse recurso.

Diante dos comentários que foram feitos, a respeito das diferenças entre o algoritmo que deu origem aos dois últimos programas, pode-se assegurar que um algoritmo pode ser implementado em uma linguagem de programação, com códigos-fonte diferentes, mas que ao serem executados geralmente obtêm-se resultados iguais.

Habituando-se com erros sintáticos.

Quando os algoritmos são codificados nas linguagens de programação, é comum ocorrerem erros de digitação, que desencadeiam erros de sintaxe no código-fonte. Talvez já tenha ocorrido quando você digitou os exemplos anteriores, no Pascalzim, e tentou executá-los. Visando conhecer os erros mais comuns e usando o último programa como modelo, faça as alterações a seguir sugeridas, e após cada uma, experimente executar/compilar o programa, para entender a mensagem de erro que o Pascalzim retorna.

É importante que o programa, **Program** Soma_dois_numeros, que será usado para testar os erros sintáticos, seja escrito exatamente como está ilustrado, principalmente com as mesmas linhas em branco, para que nas indicações de erros das linhas haja coincidência.

Siga as orientações a seguir, para realizar os experimentos com erros de programação.

- **Linha 1** – **Program** Soma_dois_numeros; – delete o < ; > e tecle F9. Possivelmente essa linha ficou marcada em vermelho, indicando algo errado, e no rodapé da janela, na barra de mensagens, as seguintes advertências: ⇒ Erro sintático na linha 1... / ⇒ 'VAR' não esperado! Faltou um ponto e vírgula no final dessa linha? – ambas as mensagens são explicativas em relação ao ocorrido. Corrija o erro, para continuar.
- **Linha 4** – N1, N2, Soma : real; – deleto os < : > e tecle F9. A linha 4 deve ter ficada marcada em vermelho e devem ter aparecido as seguintes mensagens: ⇒ Erro sintático na linha 4... / ⇒ 'REAL' não esperado! – as mensagens exibidas, indicam um erro sintático, mas não esclarecem qual é. Essa situação é comum na maioria dos retornos de erro, em que aponta o tipo

de erro, mas não exatamente onde é. Por isso é fundamental conhecer a sintaxe do ambiente de programação que está trabalhando. Corrija para continuar.

- **Linha 8** – **Write** (‘Entre com o primeiro número: ‘); – delete a segunda plica < ‘ > e tecle F9. Neste caso, a linha que ficou marcada em vermelho deve ter sido a linha 10 e as mensagens de erro devem ter sido: ⇒ Erro sintático na linha 10... / ⇒ ‘ENTRE’ não esperado! – novamente as mensagens de erro não são tão elucidativas e ainda por cima, apontam uma linha que não possui erro algum. Pois bem, a analogia que você deve fazer é a seguinte: o compilador quando encontra uma plica, considera que ali se inicia uma *string*, como a segunda plica somente foi encontrada na linha 10, e como depois da plica geralmente dever ter um fechamento de parênteses ou uma vírgula e ele encontrou um ‘ENTRE’, que não era esperado. Diante disso, para encontrar os erros é fundamental conhecer a sintaxe da linguagem, como já foi dito, e a medida que vai se exercitando também se deve fazer abstrações de como o compilador analisa o código-fonte dos programas. Recoloque a plica para prosseguir.
- **Linha 10 – Write** (‘Entre com o segundo número: ‘); – retire o < e > do comando *Write* e tecle F9. As mensagens de erro devem ser: ⇒ Erro semântico na linha 10... / ⇒ O identificador ‘WRIT’ não foi declarado dentro do escopo de PROGRAM! – trata-se de um erro fácil de identificar, porém tratado como erro semântico. Corrija o erro para continuar.

→ Atividade prática - exercícios com erros de programação:

Realize outros experimentos com erros, alterando o código-fonte em destaque, conforme sugestões e analise cuidadosamente a forma com que o compilador trata cada erro. Essa prática é fundamental para sua evolução como programador.

- Linha 1 – **Program** Soma_dois_numeros; – insira um espaço em branco no nome do programa **Program** Soma_dois_ numeros – delete o < ; > e tecle F9;
- Linha 3 – retire a palavra-reservada **Var** – e tecle F9;
- Linha 6 – retire a palavra-reservada **Begin** – e tecle F9;
- Linha 9 – **ReadIn**(N1); – substitua a variável N1 por X e tecle F9;
- Linha 12 – Soma := N1 + N2; – retire o sinal de adição < + > e tecle F9;
- Linha 14 – **End**.– retire o ponto < . > e tecle F9;

Após você ter realizado os exercícios propostos, analise os exemplos de algoritmo e programa em Pascal, a seguir, em que estão declarados os principais tipos de variáveis.


```
Algoritmo Exemplos_declaracao_de_variaveis;

Var
     NOME   : literal[30];
     IDADE  : inteiro;
     ALTURA: real;
     CASADO: lógico;

 Início
     NOME   := 'Ivo Mario Mathias';
     IDADE  := 55;
     ALTURA := 1.75;
     CASADO:= .V.;

     Escreva ('Nome completo: ', NOME);
     Escreva ('Idade: ', IDADE);
     Escreva ('Altura: ',ALTURA);
     Escreva ('Casado (S/N): ',CASADO);
 Fim.
```

EXEMPLO

```
Program Exemplos_declaracao_de_variaveis;

Var
     NOME  : string[30];
     IDADE : integer;
     ALTURA: real;
     CASADO: boolean;

 Begin
     NOME  := 'Ivo Mario Mathias';
     IDADE := 55;
     ALTURA:= 1.75;
     CASADO:= TRUE;

     writeln ('Nome completo: ', NOME);
     writeln ('Idade: ', IDADE);
     writeln ('Altura: ',ALTURA);
     writeln ('Casado (S/N): ',CASADO);
     readkey;

 End.
```

Nos exemplos exibidos anteriormente, há novos detalhes a serem estudados e aprendidos. Antes de prosseguir com a leitura, faça uma comparação analítica entre os dois casos e identifique aquilo que você ainda não conhece, para então seguir adiante.

- Com relação à declaração de literais, pode-se observar que neste caso, embora não seja uma tradução exata, a palavra usada em Pascal é *string* e a sintaxe coincide com a que está sendo utilizada em portugol.
- No caso da declaração de inteiros, percebe-se que houve uma tradução do termo em português para o inglês, *integer*.
- Para a declaração das variáveis reais, como já foi visto em outro exemplo, o termo em ambos os idiomas é o mesmo, *real*.
- Em relação à declaração das variáveis lógicas, também é usado um termo em inglês, sem tradução exata, *boolean*.
- Nota-se que logo após o *Begin* há atribuições de valores a todas as variáveis que foram previamente declaradas. A este procedimento, que geralmente ocorre no início do programa, dá-se o nome de **inicialização.** Tem por finalidade atribuir

valores iniciais às variáveis, principalmente quando elas poderiam receber atribuições de si mesmas, por exemplo, < IDADE := IDADE + 1; >, não é o que acontece no exemplo em referência, apenas para exemplificar. Neste caso, quando a variável é utilizada pela primeira vez, e como a variável aponta para uma posição de memória, pode ocorrer que aquela posição de memória não esteja vazia, mas com um valor indesejado, que estaria sendo atribuído à própria variável e somada de 1, gerando assim resultados indesejáveis.

- Ainda com relação à inicialização, percebe-se que no caso da variável do tipo *boolean* o valor também foi definido em inglês, *TRUE* (Verdadeiro).
- O último item, que merece destaque, é a inclusão de um comando novo, *readkey*, que tem por finalidade pausar a execução do programa, aguardando que qualquer tecla seja pressionada para prosseguir com a execução.

→ Atividade prática:

Incremente o programa que tem sido usado como exemplo, **Program** Soma_dois_numeros, de modo tal que além da soma, efetue e exiba também a subtração, multiplicação e divisão entre os valores que serão inseridos via teclado. Inicialmente deve-se fazer as alterações no respectivo algoritmo e, posteriormente, a implementação em Pascal.

Tenha sempre em mente, de não pular conteúdos e nem mesmo deixar assuntos com dúvidas, pois isso faz com que você não entenda o assunto seguinte. Em algoritmos e programação todos os assuntos são encadeados, um depende do outro.

Na mesma dinâmica de estudos, o próximo assunto são as estruturas de decisões, aquelas que permitem que o fluxo de um programa seja alterado mediante uma avaliação lógica.

SEÇÃO 2
ESTRUTURAS DE DECISÃO

SAIBA MAIS

A Inteligência Artificial tem como principal objetivo a representação do comportamento humano por meio de modelos computacionais, constituindo-se em um campo de pesquisa aberto e dinâmico que trata do estudo da solução de problemas através da distribuição de conhecimento entre diversas entidades.

Nas estruturas de decisão o computador adquire um certo ar de inteligência, não confundir esta metáfora com Inteligência Artificial, pois ele passa a tomar decisões e executar determinadas operações simplesmente em função de uma condição lógica, que corresponde a uma expressão lógica. Neste tipo de estrutura, o fluxo de execução das instruções de um programa pode ser escolhido, em função do resultado da avaliação lógica de uma ou mais condições.

As condições lógicas consistem na avaliação do estado de variáveis ou de expressões, avaliações estas que sempre terão como resultado um valor lógico .V. (Verdadeiro) ou .F. (Falso).

As estruturas de decisão são classificadas de acordo com o número de condições que devem ser avaliadas, resultando em dois tipos: **Se**, **Escolha**.

Estruturas de Decisão do tipo Se

Na estrutura de decisão do tipo **Se,** apenas uma expressão lógica (condição) é avaliada. Quando o resultado da condição lógica for *Verdadeiro*, então um determinado comando simples ou conjunto de instruções (comando composto) é executado, caso contrário, quando falso, um comando ou conjunto de instruções diferente é executado, podendo ser um conjunto vazio, ou seja, não existir comando(s) para a condição de falso e o programa prossegue a execução.

A visualização da sintaxe em pseudocódigo, a seguir, facilita o entendimento:

```
se <condição>
   então
         <Comando_composto_1>;
   senão
         <Comando_composto_2>;
fim_se;

ou

se <condição>
   então
         <Comando_composto_1>;
fim_se;
```

Leia atentamente os comentários:

- Inicialmente nota-se o surgimento de novas palavras-reservadas: **se**, **então**, **senão** e **fim_se.**
- No primeiro exemplo, percebe-se que se a condição for verdadeira o Comando_composto_1 será executado, se for falsa a execução é do Comando_composto_2.
- No segundo exemplo, a estrutura de decisão está vinculada a apenas um comando composto – Comando_composto_1. Significa que se for verdadeira ele será executado, caso contrário nada é executado que tenha vínculo àquela estrutura de decisão e o programa prossegue a execução com a próxima instrução após o **fim_se.**
- Em ambos os casos, fica visível que diferentemente da estrutura sequencial, em que todos os comandos do programa são executados, na estrutura de decisão isso não ocorre, pois partes do programa podem ou não ser executados, dependendo dos resultados das condições lógicas.

A seguir, serão ilustrados alguns exemplos clássicos com o uso de

estruturas de decisão, em algoritmo e Pascal, com as respectivas análises de cada um dos casos. Acompanhe atentamente cada exemplo, bem como os comentários a respeito de cada situação, não esquecendo que os exemplos em Pascal devem ser digitados no Pascalzim e compilados/executados, para conferir o resultado de cada programa.

EXEMPLO

```
Algoritmo Exemplo_decisao_1;

 Início

   se (2 > 1)então
      Escreva('VERDADEIRO')
   senão
      Escreva ('FALSO');
   fim_se;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_1;

 Begin

   if (2 > 1)then
       writeln('VERDADEIRO')
   else
       writeln('FALSO');

   readkey;

 End.
```

- Os exemplos em referência tem como uma das características, não possuir variáveis declaradas, em que duas constantes estão sendo avaliadas na expressão lógica para definir a saída do programa.

- As novas palavras-reservadas em Pascal equivalentes às do algoritmo < **se**, **então**, **senão** e **fim_se >** são < **if**, **then**, **else >**. Observe, que a princípio não há um **fim_se** equivalente em Pascal, porém, as demais instruções são as traduções do português para o inglês.
- Outra análise a ser feita no exemplo em destaque, é com relação ao valor lógico da expressão lógica `(2 > 1)`. Um modo de facilitar o entendimento da condição é fazer uma descrição dela, onde o **if**`(2 > 1)` equivale dizer, "se 2 é maior do que 1", pois bem, é evidente que 2 não é maior do que 1, portanto a expressão resulta num valor lógico *Falso*, sendo assim, neste caso a saída do programa será o comando após o **else**, `'FALSO'`.

Veja outros exemplos:

EXEMPLO

```
Algoritmo Exemplo_decisao_2;

Var
  N1, N2, Media: real;

Início

   Leia N1, N2;

   Media = (N1 + N2) / 2;

   se (Media >= 7)então
        Escreva "Aprovado";
   senão
        Escreva "Reprovado";
   fim_se;
Fim.
```

EXEMPLO

```
Program Exemplo_decisao_2;

 Var
   N1, N2, Media : real;

 Begin

   write('Entre com a primeira nota: ');
   read (N1);

   write('Entre com a segunda  nota: ');
   read (N2);

   Media := (N1 + N2) / 2;

   writeln;

   if (Media >= 7)then
       write('-> Aprovado')
   else
       write('-> Reprovado');

   readkey;

 End.
```

- Estes últimos exemplos, possuem como características estarem recebendo dois valores de entrada, notas, das quais é calculada a média e conforme o valor resultado desencadeia a saída do programa.
- A condição que é avaliada **if**(Media >= 7), que pode ser descrita como "se a Media é maior ou igual a 7". Como ainda não sabe os valores, porém, pode-se assegurar que se a condição for Verdadeira, a saída do programa é '-> Aprovado', se for Falsa '-> Reprovado'.

Mais exemplos:

EXEMPLO

```
Algoritmo Exemplo_decisao_3;

Var
   a,b,c : inteiro;

 Início

   a:=1;
   b:=2;

   se (a<b) então
     Início
       a:=b;
       c:=2;
     fim
   senão
     Início
       b:=a;
       c:=4;
     fim;
   fim_se;

   Escreva('Valores de a,b,c são: ',a,b,c);

 End.
```

EXEMPLO

```
Program Exemplo_decisao_3;

Var
   a,b,c : integer;

 Begin

   a:=1;
   b:=2;

   if (a<b) then
     Begin
      a:=b;
      c:=2;
     end
   else
     Begin
      b:=a;
      c:=4;
     end;

   write('Valores de a,b,c são: ',a,b,c);

   readkey;

 End.
```

- Nestes dois últimos exemplos, uma das características é possuírem três variáveis numéricas inteiras de entrada, porém sem valores a serem digitados via teclado. Observe que das três variáveis, duas são inicializadas.
- Um característica que esses exemplos tem em relação aos outros dois anteriores é o fato dos comandos compostos serem formados por mais de uma linha em cada resultado condicional. Quando essa situação ocorre, cada comando composto deve ser delimitado, por **Início–Fim** em algoritmos, e **Begin-End** nos programas em Pascal. Trata-se de uma sintaxe que deve ser respeitada, pois, em casos de comandos compostos com mais de uma linha, se não houver as delimitações descritas, a estrutura de decisão pode não funcionar de modo adequado.

Atividade: determine qual a saída do programa exibido anteriormente.

Saída é _____

Outros exemplos:


```
Algoritmo Exemplo_decisao_4 ;

 Var
   numero : inteiro;

 Início
   Escreva ('Entre com um número positivo menor do que 100: ');
   Leia(numero);

   se (numero >= 0) então
     se (numero <= 100) então
       Escreva(' você ACERTOU')
     senão
       Escreva(' você ERROU');
     fim_se;
   fim_se;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_4 ;

 Var
   numero : integer;

 Begin
   write ('Entre com um número positivo menor do que 100: ');
   read(numero);

   if (numero >= 0) then
     if (numero <= 100) then
       writeln(' você ACERTOU')
     else
       writeln(' você ERROU');

   readkey;

 End.
```

- Estes exemplos possuem apenas uma variável numérica do tipo inteiro, *numero*, em que o valor deverá ser digitado via teclado. Observe que após o valor ser inserido no computador, há duas estruturas de decisão **aninhadas**, que serão executadas conforme o valor digitado, porém, note que, se a primeira estrutura de decisão, a mais externa, não for satisfeita, a estrutura de decisão interna jamais será executada;
- Na primeira estrutura de decisão, a expressão lógica possui uma condição em que verifica se o valor digitado é maior ou igual a zero, (numero>= 0), a segunda possui a condição em que se o número é menor ou igual a cem;
- É possível perceber nessas duas estruturas de decisão, que está sendo testado um intervalo de valores, em que o valor digitado deve ser maior ou igual a zero e menor ou igual a cem.

ATENÇÃO

Antes de prosseguir, aqui cabe um destaque para uma situação que ocorreu a pouco, o **aninhamento** ou **embutimento** de estruturas de decisão, que corresponde ao fato de uma estrutura de programação estar dentro do conjunto de comandos (comando composto) de uma outra construção.

Agora, analise os códigos ilustrados nos próximos exemplos.

EXEMPLO

```
Algoritmo Exemplo_decisao_5 ;

  Var
   numero : inteiro;

 Início
  Escreva ('Entre com um número positivo menor do que 100: ');
  Leia(numero);

  se ((numero >= 0) .E. (numero <= 100)) então
      Escreva(' você ACERTOU')
    senão
      Escreva(' você ERROU');
  fim_se;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_5 ;

  Var
   numero : integer;

 Begin
   write ('Entre com um número positivo menor do que 100: ');
   read(numero);

   if ((numero >= 0) and (numero <= 100)) then
      writeln(' você ACERTOU')
   else
      writeln(' você ERROU');

   readkey;

 End.
```

- Pode-se afirmar que o `Exemplo_decisao_5` é um algoritmo/programa que quando executado teria como saída os mesmos resultados do `Exemplo_decisao_4`. Note que a diferença é a eliminação da estrutura de decisão interna, sendo ela substituída pela conjunção < .E. >/< and >. Lembrando-se da tabela-verdade, em uma conjunção para que na expressão como um todo obtenha-se o valor *Verdadeiro*, ambas as subexpressões também devem ser Verdadeiras. Então, numa situação similar aos exemplos em referência, duas estruturas de decisão equivalem a uma estrutura de decisão com duas condições onde o operador é uma conjunção.

Os próximos exemplos seguem a mesma analogia dos anteriores, em que há estruturas de decisão aninhadas, observe a seguir.

EXEMPLO

```
Algoritmo Exemplo_decisao_6;

 Var
    numero : inteiro;

 Início
    Escreva('Digite um número: ');
    Leia(numero);

    se (numero < 0) então
       Escreva ('valor menor que zero')
    senão se (numero < 10) então
             Escreva ('valor => 0 e < 10')
          senão se (numero < 100) então
                   Escreva ('valor => 10 e < 100')
                senão
                   Escreva ('valor => 100');
                fim_se;
          fim_se;
    fim_se;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_6;

 Var
    numero : integer;

 Begin
    write('Digite um número: ');
    read(numero);

    if (numero < 0) then
       writeln ('valor menor que zero')
    else if (numero < 10) then
       writeln ('valor => 0 e < 10')
    else if (numero < 100) then
       writeln ('valor => 10 e < 100')
    else
       writeln ('valor => 100');

    readkey;

 End.
```

- Antes de prosseguir com a leitura, analise cuidadosamente os exemplos anteriores.
- Possivelmente você deve ter percebido que há três estruturas de decisão aninhadas e, que também é um programa que testa intervalos de valores, que pode ser ampliado para diversos valores.
- Observe também, que há testes de todos os valores, consistências, inclusive para valores negativos, ou seja, para qualquer valor inserido o programa tem uma saída.
- Outra analogia que se pode fazer é que estruturas de decisão como essas últimas, são úteis na prática, pois situações onde há necessidade de testar valores são comuns, por exemplo, imagine um sistema onde é necessário verificar faixas de valores de produtos, idades, pesos, faixas etárias, etc.

Os próximos exemplos possuem algumas características similares aos anteriores, porém com tipos de dados diferentes, literais/*strings*, observe que há algumas situações novas, no que se refere ao uso de instruções e um tipo de dado novo em Pascal.

EXEMPLO

```
Algoritmo Exemplo_decisao_7;

 Var
   letra : literal[1];

 Início

   Escreva (' Tecle a consoante da palavra "aula" -> ');
   Leia(letra);

   se (letra = 'l') então
      Escreva (' você ACERTOU - letra L)
   senão
      Escreva (' você ERROU ');
   fim_se;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_7;

 Var
   letra : char;

 Begin

   write (' Tecle a consoante da palavra "aula" -> ');
   letra := readkey;

   if (letra = 'l') then
      write (' você ACERTOU - letra L)
   else
      write (' você ERROU ');

   readkey;

 End.
```

- Inicialmente, observe o tipo de dado novo em Pascal, *char*, esse tipo de dado é utilizado quando se quer declarar apenas uma letra, ocupando um byte apenas na memória, outra situação associada a esse tipo é usar uma instrução de entrada de dados em que não há necessidade de teclar *enter*, que é o caso do *readkey*. Diante disso, percebe-se que o *readkey* possui outra finalidade, além daquela de simplesmente pausar a execução de um programa.
- Ainda sobre o tipo *char*, como a forma de escrever algoritmos é flexível, embora não conste na classificação algorítmica como tipo de dado padrão, pode-se usar o tipo *caractere* para representar o *char*, em vez de *literal[1]*.
- Nos exemplos anteriores, falou-se em consistências, para que o programa filtre o máximo de possibilidades digitadas, evitando entradas de dados indesejadas e verificando situações adversas. Nessa ideia, imagine qual seria a saída do programa se o usuário tecla-se o "L" maiúsculo. Pois bem, se você observar atentamente o programa, vai perceber que a saída seria ' você ERROU ', mesmo sendo digitado um "l";
- Veja nos próximos exemplos, como essa falha do programa pode ser sanada de forma relativamente simples.

EXEMPLO


```
Algoritmo Exemplo_decisao_8;

 Var
   letra : caracter;

 Início

   Escreva (' Tecle a consoante da palavra "aula" -> ');
   Leia := (letra);

   se ((letra = 'l') .OU. (letra = 'L')) então
       Escreva (' você ACERTOU - letra L')
   senão
       Escreva (' você ERROU ');
   fim_se;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_8;

 Var
      letra : char;

 Begin

  write (' Tecle a consoante da palavra "aula" -> ');
  letra := readkey;

   if ((letra = 'l') or (letra = 'L')) then
       write (' você ACERTOU - letra L')
   else
       write (' você ERRROU ');

   readkey;

 End.
```

- Então neste caso, uma solução possível foi uma alteração na estrutura condicional incluindo uma verificação para o "L" maiúsculo, em que a expressão passa a ser complementada pelo operador lógico, .OU., de disjunção. Lembrando que, pela tabela-verdade, para a disjunção basta apenas um dos operadores ser verdade para que a expressão como um todo seja verdadeira.

Os exemplos para testar intervalos, vistos até agora, foram com valores de tipos numéricos, porém a mesma necessidade pode ocorrer com literais. Observe os exemplos como esta característica pode ser implementada.

EXEMPLO

```
Algoritmo Exemplo_decisao_9;

 Var
     letra : caractere;

 Início

   Escreva ('Tecle uma letra entre A e Z: ');
   Leia := (letra);

   se ((letra >= 'A' ) .E. (letra <= 'Z')) então
      Escreva(' você ACERTOU - letra MAIÚSCULA')
   senão
      Escreva(' você ERROU');
   fim_se;

Fim.
```

EXEMPLO

```
Program Exemplo_decisao_9;

 Var
     letra : char;

 Begin

   write ('Tecle uma letra entre A e Z: ');
   letra := readkey;

   if ((letra >= 'A' ) and (letra <= 'Z')) then
      writeln(' você ACERTOU - letra MAIÚSCULA')
   else
      writeln(' você ERROU');

   readkey;

 End.
```

- Antes de prosseguir com a leitura, procure entender exatamente como esses algoritmo/programa funcionam, inclusive testando o código-fonte no Pascalzim.
- Partindo do princípio que tenha entendido perfeitamente o objetivo desses exemplos, deve ter percebido que mesmo que seja digitada uma letra entre 'A' e 'Z', mas **minúscula** o programa retornará como saída ' você ERROU'.
- A situação comentada anteriormente, pode ser até certo ponto obvia, porém é importante relembrar o por que. Observe na figura 11, da tabela ASC, que as letras do alfabeto, tanto minúsculas como maiúsculas, estão sequencialmente distribuídas na tabela. Quando um programa testa intervalos de letras do alfabeto, sempre se deve levar em conta se são minúsculas ou maiúsculas, pois elas não se encontram na mesma sequência.

Aproveitando os comentários anteriores, imagine um programa que teste separadamente as letras do alfabeto, minúsculas/maiúsculas, veja a seguir, uma possível alternativa de algoritmo/programa com esta característica.

EXEMPLO

```
Algoritmo Exemplo_decisao_10;

 Var
     letra : caractere;

 Início
   Escreva ('Tecle uma letra entre a e z: ');
   Leia := (letra);

   se ((letra >= 'A' ) .E. (letra <= 'Z')) então
      Escreva(' você teclou - ',letra,' - MAIÚSCULA')
   senão se ((letra >= 'a' ) .E. (letra <= 'z')) então
        Escreva(' você teclou - ',letra,' - minúscula')
      senão
        Escreva(' Caractere inválido, não é uma letra....');
      fim_se;
   fim_se;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_10;

 Var
     letra : char;

 Begin
   write ('Tecle uma letra entre a e z: ');
   letra := readkey;

   if ((letra >= 'A' ) and (letra <= 'Z')) then
      writeln(' você teclou - ',letra,' - MAIÚSCULA')
   else if ((letra >= 'a' ) and (letra <= 'z')) then
           writeln(' você teclou - ',letra,' - minúscula')
        else
           writeln(' Caractere inválido, não é uma letra....');

  readkey;

 End.
```

- Observe, analise e teste a solução proposta anteriormente. Note que, para qualquer tecla que seja digitada o programa consegue fazer a consistência e retornar uma mensagem correspondente a cada caso.

Outra modalidade de estrutura de decisão são as do tipo **Escolha**, que será abordada em seguida.

Estruturas de Decisão do tipo Escolha

Na estrutura de decisão do tipo Escolha, pode haver uma ou mais condições a serem testadas e um comando composto diferente associado a cada uma destas escolhas. Também permite que a condição avaliada resulte em valores diferentes de *Verdadeiro* ou *Falso*, pelo fato que pode existir uma sequência de ações que serão executadas de acordo com o resultado da expressão avaliada.

Em pseudocódigo a sintaxe é:

```
Escolha
        Caso   <Condição_1>
               <Comando_composto_1>;
        Caso   <Condição_2>
               <Comando_composto_2>;
        .....
Caso    <Condição_n>
               <Comando_composto_n>;
        Senão
               <Comando_composto_s>;
Fim_escolha;
```

A estrutura de decisão do tipo Escolha, pode ser traduzida do seguinte modo: caso o resultado da expressão seja igual a Condição_1 executa a sequência de comandos do Comando_composto_1, se o resultado da expressão for igual a Condição_2 executa a sequência de comandos do Comando_composto_2, e assim por diante, conforme o número de escolhas e condições.

Analise detalhadamente os exemplos a seguir, pois nesta modalidade de estrutura de decisão há diferenças significativas do algoritmo em relação ao código-fonte em Pascal, mas o resultado de execução do programa é o mesmo proposto no algoritmo.

EXEMPLO

```
Algoritmo Exemplo_decisao_11;

 Var
   operando_1,operando_2 : real;
   operador : literal[1];

 Início
   Escreva ('Digite: número operador número = ');
   Leia (operando_1, operador, operando_2);

   Escolha (operador)
      caso operador = '+'
           Escreva (' = ', operando_1 + operando_);
      caso operador = '-'
           Escreva (' = ', operando_1 - operando_);
      caso operador = '*'
           Escreva (' = ', operando_1 * operando_);
      caso operador = '/'
           Escreva (' = ', operando_1 / operando_);
      senão
           Escreva (' ...Operador inválido...');
   Fim_escolha;

 Fim.
```

EXEMPLO

```
Program Exemplo_decisao_11;

 Var
   operando_1,operando_2 : real;
   operador : char;

 Begin
       writeln(' Calculadora: ');
       write(' Primeiro operando: ');
       read(operando_1);
       write(' Segundo  operando: ');
       read(operando_2);
       write(' Entre com o operador (+ - * /): ');
       operador := readkey;

       case operador of
             '+' : write (' = ', operando_1 + operando_2);
              '-' : write (' = ', operando_1 - operando_2);
              '*' : write (' = ', operando_1 * operando_2);
              '/' : write (' = ', operando_1 / operando_2);
              else
                 begin
                   writeln;
                   writeln (' ...Operador inválido...');
                 end;
       end;

       readkey;
 End.
```

- Observe que a diferença ocorre na sintaxe e nos comandos de como codificar a estrutura de decisão do tipo escolha em Pascal, em que se usa **case** operador **of,** e em algoritmo **Escolha (**operador**)** e a diferença também ocorre no modo de representar as condições, conforme pode ser observado nos exemplos anteriores.

Evidentemente, nos exemplos que foram apresentados, não foram abordadas todas as possíveis situações de uso de estruturas de decisão. Inúmeras outras possibilidades existem e sempre lembrando, as situações de uso estão diretamente ligadas ao problema que está sendo resolvido por meio de um programa de computador.

A próxima seção tem como foco as estruturas de **repetição**, que de maneira direta possui vínculo com as estruturas de **decisão**.

Sempre reforçando, o entendimento integral de um assunto é fundamental para aprender o próximo. Portanto, caso você não esteja seguro de ter aprendido as estruturas de decisão, retorne os estudos para a seção anterior, antes de prosseguir.

SEÇÃO 3
ESTRUTURAS DE REPETIÇÃO

São também chamadas de **Laços** ou **Loops** e correspondem a uma modalidade de estrutura de programação em que o objetivo é repetir, iterar, determinado trecho de um programa, certo número de vezes. A repetição de partes de um programa é uma situação que além de útil é muito comum em programação de computadores, pois facilita a reutilização de códigos-fonte, trechos de programas. Por exemplo, um programa que executa um cálculo financeiro em que a fórmula seja a mesma, mas os valores podem ser diferentes e incrementados em razão de uma situação temporal.

As estruturas de repetição possuem uma característica peculiar, pelo fato de estarem diretamente ligadas às estruturas de decisão, pois, conforme será visto, o controle das repetições e interrupções ocorre em função de expressões lógicas e condições.

As estruturas de repetições são classificadas pelo conhecimento prévio, ou não, do número de vezes em que um conjunto de comandos compostos deverá ser executado, **laços**, dividindo-se em:

- **Laços contados** – quando se conhece, previamente, quantas vezes o comando composto no interior do laço deverá ser executado.
- **Laços condicionais** – quando não se conhece o número de vezes que o comando composto no interior do laço será repetido, pelo

fato de que a interrupção está vinculada a uma condição sujeita a modificações pelas instruções do interior do laço.

Laços Contados

São aqueles em que há um mecanismo para contar o número de vezes que o corpo do laço, comando composto, em seu interior, deverá ser executado. Os laços contados também são chamados de **Para-passo**.

Em pseudocódigo a sintaxe é:

Para <var> **de** <início> **até** <final> **incr de** <inc> **faça**
    <Comando_composto>;
Fim_para;

Observações importantes:

- **<var>** – é uma variável que seu valor é alterado a cada iteração (volta do laço);
- **<início> –** pode ser uma constante ou variável que define o valor inicial das iterações**;**
- **<fim>** – também pode ser uma constante ou variável que determina o valor do encerramento das iterações. Esse valor é avaliado pelo mecanismo da instrução **Para**, em que o valor de **<var>** é comparado com o de **<fim>;**
- **<inc> –** pode ser uma constante ou variável, em que o valor que é adicionado à variável <**var**> ao final de cada iteração do laço. Há linguagens de programação que permitem que lhe seja atribuído um valor negativo, de modo que o valor da variável <**var**> diminui a cada iteração. Neste caso, deve-se atentar à necessidade de inversão do sinal de comparação (de > para <) que é feito a cada volta do laço.

Do mesmo modo que ocorreu com as estruturas de decisão, há diferenças entre o algoritmo e o código-fonte em Pascal, que poderão ser observadas e analisadas nos exemplos a seguir. Lembrando que todos os exemplos em Pascal, devem ser digitados no Pascalzim e executados.

EXEMPLO

```
Algoritmo Exemplo_repeticao_1;

 var
   i : inteiro;

 Inicio

   Para i de 1 até 1000 incr de 1 faça
       Escreva(i);
   Fim_para;

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_1;

 var
   i : integer;

 Begin

   for i := 1 to 1000 do
       write(' ', i);

   readkey;

 End.
```

- O primeiro item a ser analisado é a linha do algoritmo que contém a instrução **Para** *i* **de** 1 **até** 1000 **incr de** 1 **faça,** observa-se que a leitura dessa linha expressa o objetivo que é incrementar a variável *i*, iniciando em 1, incrementando em 1, até chegar em 1000.
- Agora, analisando a linha correspondente em Pascal, **for** i **:=** 1 **to** 1000 **do,** fazendo uma tradução seria, para i de 1 até 1000 faça. Nesse caso, e isso é padrão nas linguagens de programação, quando o incremento não está destacado sempre é 1.
- Em ambos os casos, é notório que o comando composto corresponde simplesmente a exibição do conteúdo da variável *i*, sendo ela incrementada de 1 até 1000.

Analise os próximos exemplos:

EXEMPLO

```
Algoritmo Exemplo_repeticao_2;

 var
   i : inteiro;

 Inicio

   Para i de 0 até 100 incr de 2 faça
      Escreva(i);
   Fim_para;

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_2;

 var
   i : integer;

 Begin

   for i := 0 to 100 do
     begin
       if ((i MOD 2) = 0) then
          write(' ', i);
     end;

   readkey;
 End.
```

- Os exemplos exibidos demonstram adaptações que devem ser feitas no código-fonte, quando não se consegue implementar integralmente o algoritmo na linguagem de programação em uso.
- Partindo da análise do algoritmo, na linha **Para** i **de** 0 **até** 100 **incr de** 2 **faça,** nota-se que o objetivo desta linha é incrementar a variável *i*, iniciando em 0, incrementando em 2, até chegar a

100. Pode-se perceber, neste caso, que o objetivo do algoritmo ao exibir os valores de i da forma descrita é exibir os valores pares entre 0 a 100, sendo então o resultado esperado em um programa implementado com esse algoritmo.

- Agora, ao observar o exemplo correspondente em Pascal, percebe-se que na linha, **for** i **:=** 0 **to** 100 **do,** está expresso que, para i de 0 até 100 faça, porém como já comentado anteriormente, quando não está indicado o incremento, o padrão é 1. Porém, o solicitado no algoritmo é 2. O Pascalzim é um ambiente de programação, em que não está implementado o incremento em 2, ou outro valor, para o comando *For*. Para contornar essa situação, optou-se em atingir o objetivo do algoritmo de outra forma, ver o exemplo, em que aparece uma instrução nova *MOD*, que corresponde ao módulo de uma divisão, resto de valor inteiro, e como nesse caso é testado quando o resultado correspondente é 0 em uma divisão por 2, resultam em apenas os valores pares exibidos pelo programa.

Como já é sabido, para um determinado problema, pode-se implementá-lo de modos diferentes, tanto o algoritmo como o código-fonte em alguma linguagem e obter resultados iguais. Observe os próximos exemplos com essa característica, em que o objetivo, assim como foi nos anteriores, é a exibição dos números pares entre 0 e 100.


```
Algoritmo Exemplo_repeticao_3;

 var
   i : inteiro;

 Início

   Para i de 0 até 50 incr de 1 faça
        Escreva (i*2);
   Fim_para;

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_3;

 var
   i : integer;

 Begin

   for i := 0 to 50 do
       write(' ',i*2);

    readkey;
 End.
```

- Partindo do princípio que o programador sabe que a linguagem de programação que será utilizada para implementar o algoritmo em questão, não possui estruturas de repetição com incremento diferente de 1. Buscou uma solução, conforme os exemplos anteriores exibidos, relativamente simples para o problema, que seria incrementar os valores entre 0 a 50, simultaneamente, multiplicando-os por 2.
- Reforçando a ideia que tem sido comentada, para um mesmo problema podem ser encontradas soluções diferentes com o mesmo resultado, sendo que, até para o problema em questão outras soluções são possíveis.

Outros exemplos, analise-os e execute-os antes de ler os comentários, pois possuem novas características.

EXEMPLO

```
Algoritmo Exemplo_repeticao_4;

 Var
   i, soma : inteiro;

 Início

   soma := 0;
   Para i de 1 até 10 incr de 1 faça
        soma := soma + i;
        Escreva('nro. = ', i ,' soma = ',soma);
   Fim_para;

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_4;

 Var
   i, soma : integer;

 Begin

   soma := 0;
   for i := 1 to 10 do
   Begin
     soma := soma + i;
     writeln('nro. = ', i ,' soma = ',soma);
   end;

   readkey;

 End.
```

- Você deve ter observado que os exemplos em referência possuem duas variáveis sendo utilizadas nessa estrutura de repetição. A variável soma, tem como finalidade fazer uma soma cumulativa dos valores que i obterá a medida que é incrementado em 1.
- Um detalhe que deve ser destacado é o fato da variável soma ser **inicializada** em 0, antes de ser utilizada no comando composto da estrutura de repetição.

ATENÇÃO

Para relembrar – inicializar variáveis tem por finalidade atribuir valores iniciais às variáveis, principalmente quando elas podem estar recebendo atribuições de si mesmas. Isso evita que valores indesejados, comumente chamados de sujeira, que podem estar na memória do computador, sejam atribuídos à variável e corrompam os valores esperados.

Observe e analise detalhadamente os próximos exemplos.

EXEMPLO

```
Algoritmo Exemplo_repeticao_5;

 Var
   i, j : inteiro;

 Início
   Para i de 1 até 5 incr de 1 faça
      Escreva ('-> Executando o laço EXTERNO - ',i,' vez(es)');
      Para j de 1 até 3 incr de 1 faça
         Escreva ('----> Executando o laço INTERNO - ',j,' vez(es)');
      Fim_para;
   Fim_para;
 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_5;

 Var
   i, j : integer;

 Begin

   for i := 1 to 5 do
      begin
       writeln;writeln ('-> Executando o laço EXTERNO - ',i,' vez(es)');
       for j := 1 to 3 do
         writeln ('---> Executando o laço INTERNO - ',j,' vez(es)');
       end;

   readkey;

 End.
```

- Os dois exemplos anteriores, possuem como característica o fato de serem duas estruturas de repetição aninhadas.
- É fundamental você entender o funcionamento destas construções, pois se trata de uma situação muito comum em programação.
- Basicamente, pode-se notar que o laço interno é executado tantas vezes quanto a variável do laço externo for incrementada.

Os próximos exemplos, abordam o uso de caracteres em estruturas de repetição.

EXEMPLO

```
Algoritmo Exemplo_repeticao_6;

 var
   letra: caracter;

 Início

   Para letra de 'a' até 'z' incr de 1 faça
       Escreva(letra);
   Fim_para;

   Para letra de 'A' até 'Z' incr de 1 faça
       Escreva(letra);
   Fim_para;

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_6;

 var
   letra : char;

 Begin

   writeln('Letras minúsculas');
   For letra := 'a' to 'z' do
      write(letra,' ');

   writeln;writeln('Letras Maiúsculas');
   For letra := 'A' to 'Z' do
      write(letra,' ');

   readkey;

 End.
```

- Como característica principal, os exemplos em questão têm o fato da variável que receberá o incremento para a repetição, não ser uma variável numérica do tipo inteiro e um caractere, *char*.
- Como segunda observação, você deve ter percebido que é possível representar tanto o alfabeto minúsculo como maiúsculo, basta, indicar o início e o fim conforme deseje.

Agora, você estudará como funcionam as estruturas de repetição com laços condicionados.

Laços Condicionados

São aqueles, que o comando composto no interior do laço é executado até que uma determinada condição seja satisfeita. Diferentemente dos laços contados, em que a condição de parada já está expressa previamente no próprio comando.

Nos laços condicionais, ao contrário do que acontece nos laços contados, não se sabe, previamente, quantas vezes o corpo do laço será executado, pois está sujeito a alterações em dados internos do comando composto que a cada iteração são avaliados para determinar se a repetição deve continuar ou ser interrompida.

Nesse contexto, nos laços condicionais a variável que é testada, podendo ser no início quanto no final do laço, deve sempre estar associada a um comando que a atualize no interior do laço. Caso isso não ocorra, o programa poderá ficar repetindo indefinidamente o laço, gerando uma situação conhecida como "laço infinito".

As construções mais comuns de laços condicionados são: **Enquanto** e **Repita**.

Construção Enquanto

Tem como característica principal o fato de ser uma estrutura que realiza o teste lógico, condição, no início do laço, verificando se é permitido ou não executar o conjunto de comandos no interior do laço. Sendo assim, é necessário que exista uma ou mais variáveis com valores associados antes da execução da estrutura de repetição em si.

Em pseudocódigo a sintaxe é:

Enquanto <condição> **faça**
  <comando_composto>;
Fim_enquanto;

Com relação ao funcionamento da construção **Enquanto**, como a condição é testada no início, se o resultado for falso, então o comando composto no seu interior não é executado e a execução prossegue normalmente pela instrução seguinte ao final da estrutura de repetição, **Fim_enquanto**.

Caso a condição seja verdadeira, o comando composto é executado e, ao término, é realizado outro teste da condição, este processo se repete enquanto a condição for verdadeira. Quando a condição for falsa, o processo de repetição é interrompido e o fluxo de execução do algoritmo/programa prossegue normalmente pela instrução seguinte ao final da estrutura de repetição.

O programador sempre deve estar atento ao fato de que em algum momento no comando composto, possa ocorrer em que a condição assuma a condição de falso, caso contrário o programa permanecerá executando indefinidamente o comando composto, laço infinito.

Estude e implemente em Pascal os exemplos a seguir, sempre observando os detalhes teóricos descritos anteriormente, preferencialmente antes de ler os comentários sobre cada caso.

EXEMPLO

```
Algoritmo Exemplo_repeticao_7;

 Var
   num : inteiro;

 Início

   num := 0;

   Enquanto (num <= 100) faça
      Escreva(num);
      num := num + 1;
   Fim_enquanto;

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_7;

 var
   num : integer;

 Begin

   num := 0;

   while (num <= 100) do
     Begin
        write (' ',  num);
        num := num + 1;
     end;

  readkey;
 End.
```

- Possivelmente você deve ter percebido que os exemplos descritos anteriormente, têm como finalidade exibir os números entre 0 a 100.
- Detectada a finalidade dos exemplos em questão, identifique onde está a inicialização da variável, *num*, de controle do teste lógico que controla a execução do laço.
- Outra observação importante é analisar o teste condicional em si, que determina a execução ou interrupção dos laços.
- Como a estrutura de repetição em questão, trata-se de uma construção em que o programador deve controlar integralmente a execução dos laços, perceba e identifique qual a variável que é utilizada no teste condicional deve sofrer alterações de modo que em algum momento interrompa a execução dos laços. Nesses casos, específicos a ela é somado 1, até que atinja o valor 101, quando o laço é interrompido.

Outros exemplos para estudar:

EXEMPLO

```
Algoritmo Exemplo_repeticao_8;

 Var
    num, soma : inteiro;

 Início

    num := 1;
    soma := 0;

  Escreva(' A soma de: ');

    Enquanto (num <= 10)
      Escreva(num);
      soma := soma + num;
      num := num + 1;
    Fim_enquanto;

    Escreva(' = ',  soma);

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_8;

 var
    num, soma : integer;

 Begin

    num := 1;
    soma := 0;

  write (' A soma de: ');

    while (num <= 10) do
       Begin
         write (' ', num);
         soma := soma + num;
         num := num + 1;
       end;

    write (' = ', soma);

    readkey;

 End.
```

- Os exemplos em questão possuem similaridades com os anteriores, porém o objetivo proposto é uma soma cumulativa dos valores entre 1 a 100 e a exibição dos respectivos valores.
- Observe que como as duas variáveis estão sendo utilizadas em somas de si mesmas, neste caso a inicialização é necessária.

Os exemplos a seguir, devem ser estudados com uma atenção ainda maior do que os demais já exibidos. A razão é o fato que a medida que o aprofundamento na disciplina avança, obviamente o conteúdo se torna mais complexo e com uma abrangência maior, no que se refere a construção de algoritmos/programas. Pois, para que os programas sejam eficientes e mais completos, uma carga maior de conhecimento deve ser dedicada a eles. É uma situação que você já deve ter notado no decorrer da disciplina.

EXEMPLO

```
Algoritmo Exemplo_repeticao_9;

  var
    n_car, n_dig : inteiro;
    carac : caracter;

  Início
    n_car := 0;
    n_dig := 0;
    carac := asc(32);

    Escreva (' Digite uma frase encerre com <enter>: ');

    Enquanto (carac <> asc(13))
      Leia (carac);
      se (carac <> asc(13)) então
          n_car := n_car + 1;
      se ((carac >= '0') .E. (carac <= '9')) então
          n_dig := n_dig + 1;
      fim_se;
    Fim_enquanto;

    Escreva ('Número total de caracteres : ', n_car);
    Escreva ('Número de dígitos numéricos: ', n_dig);

  Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_9;

  var
    n_car, n_dig : integer;
    carac : char;

  Begin
    n_car := 0;
    n_dig := 0;
    carac := chr(32);

    write (' Digite uma frase encerre com <enter>: ');

    while (carac <> chr(13)) do
      Begin
        carac := readkey;
        write(carac);
        if (carac <> chr(13)) then
            n_car := n_car + 1;
        if ((carac >= '0') and (carac <= '9')) then
            n_dig := n_dig + 1;
      end;

        writeln;
        writeln ('Número total de caracteres : ', n_car);
        writeln ('Número de dígitos numéricos: ', n_dig);

      readkey;

  End.
```

- Espera-se que você tenha percebido que a finalidade desse programa é a contagem total de caracteres digitados e quantos destes são números, sendo que isso deve ocorrer antes que seja teclado o *Enter*.
- Uma dúvida que talvez você tenha, quem é o *Enter* nesse contexto. Pois bem, o *Enter* nesses exemplos é representado pelo *asc(13)* no algoritmo e em Pascal pelo *chr(13)*. Ele é comparado com a variável *carac* nas construções – **Enquanto (**carac <> asc(13)**)/ while (carac <> chr(13)) do** – respectivamente. Essas comparações querem dizer, enquanto

o *Enter* não for digitado, continua com a repetição da estrutura. E evidentemente a tecla *Enter* na tabela ASC é representada pelo número inteiro 13.

- Outro detalhe que é importante você ficar atento é com relação à inicialização da variável *carac* nos dois casos – *carac* := **asc(32);** / *carac* **:= chr(32);** – observa-se que o valor inicial é o código 32 da tabela ASC, o qual representa um espaço em branco, sendo diferente de *Enter*, permite que a estrutura de repetição seja executada pelo menos uma vez, as demais serão avaliadas pelo que for digitado.

Continuando com a metodologia, em que os exemplos possuem particularidades. Procure identificar, antes de ler os comentários, o que você encontra de diferente em relação ao que já foi estudado até agora e evidentemente como o programa funciona e qual o objetivo dele.

EXEMPLO

```
Algoritmo Exemplo_repeticao_10;

 Var
    num, soma : inteiro;

 Início

  soma := 0;
  num := 2;

  Enquanto ((num MÓDULO 2) = 0)
      Escreva ('Entre com um número par inteiro: ');
      Leia(num);
      soma := soma + num;
  Fim_enquanto

  Escreva (' --> A soma dos números digitados é : ', soma);


 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_10;

 var
    num, soma : integer;

 Begin

   soma := 0;
   num := 2;

   while ((num MOD 2) = 0) do
     Begin
       write ('Entre com um número par inteiro: ');
       readln(num);
       soma := soma + num;
     end;

   write (' --> A soma dos números digitados é : ', soma);

   readkey;

 End.
```

- Possivelmente você deve ter percebido que a novidade são as condições das construções – **Enquanto ((**num **MÓDULO** 2**) =** 0**)** – **while ((**num **MOD** 2**) =** 0**) do** – , em que o critério de entrada nas estruturas de repetição é que a variável *num* seja um número par, lembrando que a função MÓDULO/MOD é aquela que retorna o resto de uma divisão de números inteiros.
- Então, o teste lógico da condição verifica se a variável *num* é um número par, continua a execução, caso contrário, se for ímpar a execução da estrutura de repetição será interrompida e nessa situação o programa segue o fluxo exibindo o resultado.
- Por essa razão, a variável *num* foi inicializada antes da estrutura com o número par 2.

Fique atento, a próxima modalidade de laços condicionados a ser estudada é a construção **Repita**. Ela tem particularidades significativas que a diferenciam da **Enquanto**.

Construção Repita

Tem como característica principal o fato de ser uma estrutura que realiza o teste lógico, condição, no final do laço, verificando se é permitido ou não executar novamente o conjunto de comandos no interior do laço.

Em pseudocódigo a sintaxe é:

Repita
  <comando_composto>;
Até_que <condição>;

O funcionamento da construção **Repita** é similar ao **Enquanto,** porém o comando composto é executado uma vez e após a execução ocorre o teste lógico da condição, **se for FALSO**, **o comando composto é executado novamente** e esse processo é repetido até que a condição seja verdadeira, quando então a execução prossegue normalmente pela instrução seguinte ao final da estrutura de repetição.

Uma particularidade que ocorre em relação às estruturas Repita e Enquanto é com relação ao momento de inicializar as variáveis que controlam os processos de iterações. Na estrutura Enquanto, é necessário que a variável seja inicializada antes do laço, visto que, a condição de entrada no laço ocorre no início da construção e, se a variável não estiver com um valor que torne a condição verdadeira a execução do comando composto não vai ocorrer. Diferentemente da estrutura Repita, em que o teste lógico ocorre no final, portanto a variável pode ser inicializada dentro do próprio laço.

Resumindo, as estruturas de repetição **Repita** e **Enquanto** diferem uma da outra pelo fato de a primeira, **Repita**, efetuar o teste de condição no final e portanto, o comando composto é executado ao menos uma vez, na **Enquanto**, o teste da condição é feito no início, dessa forma, o comando composto será executado nenhuma ou mais vezes.

Continuando com os procedimentos anteriores, analise atentamente, implemente em Pascal e execute cada um dos exemplos que serão apresentados na sequência.

Nos exemplos, serão usados como referência os mesmos das construções **Enquanto**. Ou seja, o objetivo é que você possa comparar os modos diferentes em que foram desenvolvidos os algoritmos/programas para os mesmos propósitos.

Inicialmente o Exemplo_repeticao_10, desenvolvido com a estrutura **Repita**.

EXEMPLO

```
Algoritmo Exemplo_repeticao_11;

 Var
    num, soma : inteiro;

 Início

    soma := 0;

    Repita
       Escreva ('Entre com um número par inteiro: ');
       Leia(num);
       soma := soma + num;
    até que ((num MOD 2) <> 0);

    Escreva (' --> A soma dos números digitados é : ', soma);

 Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_11;

 var
    num, soma : integer;

 Begin

    soma := 0;

    Repeat
       write ('Entre com um número par inteiro: ');
       readln(num);
       soma := soma + num;
    until ((num MOD 2) <> 0);

  write (' --> A soma dos números digitados é : ', soma);

  readkey;

End.
```

- Você deve ter observado que a forma de escrever a estrutura **Repita** em Pascal, além de ser uma tradução para o inglês, dispensa o *Begin* e o *end*.
- Outro detalhe fundamental que você sempre deve ficar atento, para as construções **Repita**, é o fato de que não é preciso inicializar a variável, ou as variáveis, que controlam o teste condicional, uma vez que o teste é realizado no final da estrutura.
- Outro fato de destaque é que a avaliação lógica muda, pois, analisando literalmente o próprio comando, uma vez quando se diz **até que**, significa que seja *Verdadeiro*. Portanto, ao invés de – **Enquanto ((**num MÓDULO 2**) = 0) / while ((num MOD 2) = 0) do** – tem-se **até que ((**num **MOD** 2**) <>** 0**) / until ((**num **MOD** 2**) <>** 0**).**
- Resumindo, fazendo uma analogia, quer dizer, na estrutura – **Enquanto** for *Verdadeiro* **faça** – na construção **Repita** até que seja *Verdadeiro* **interrompa**.
- Outro item que também deve ser observado, em relação ao exemplo em referência, Exemplo_repeticao_10, especificamente comparando os códigos em Pascal, houve uma redução em duas

linhas de código, uma relativa a não inicialização da variável *num* e a outra a supressão do *Begin*.

Os próximos modelos são em relação ao Exemplo_repeticao_9, desenvolvido com a estrutura **Repita**.

EXEMPLO

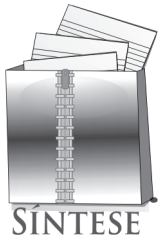
```
Algoritmo Exemplo_repeticao_12;

  var
    n_car, n_dig : inteiro;
    carac : caracter;

  Início
    n_car := 0;
    n_dig := 0;

    Escreva (' Digite uma frase encerre com <enter>: ');

    Repita
      Leia (carac);
      se (carac <> asc(13)) então
          n_car := n_car + 1;
      se ((carac >= '0') .E. (carac <= '9')) então
          n_dig := n_dig + 1;
      fim_se;
    até que (carac = asc(13));

    Escreva ('Número total de caracteres: ', n_car);
    Escreva ('Número de dígitos numéricos: ', n_dig);

  Fim.
```

EXEMPLO

```
Program Exemplo_repeticao_12;

  var
    n_car, n_dig : integer;
    carac : char;

  Begin
    n_car := 0;
    n_dig := 0;

   write (' Digite uma frase encerre com <enter>: ');

    repeat
      carac := readkey;
      write(carac);
      if (carac <> chr(13)) then
           n_car := n_car + 1;
      if ((carac >= '0') and (carac <= '9')) then
           n_dig := n_dig + 1;
    until (carac = chr(13));

    writeln;
    writeln ('Número total de caracteres : ', n_car);
    writeln ('Número de dígitos numéricos: ', n_dig);

    readkey;

  End.
```

- Sintetizando, as diferenças seguem o mesmo raciocínio das comentadas em relação ao Exemplo_repeticao_12, em que não houve a inicialização da variável que controla o teste condicional e a avaliação lógica fica invertida em relação a construção **Enquanto**.

→ **Atividade prática** - reescreva os algoritmos e programas dos Exemplos_repeticao_7 e Exemplo_repeticao_8 – substituindo a estrutura de repetição Enquanto/*While* pela Repita/*Repeat*.



Ao completar os estudos desta unidade, você consolidou aqui todos os conhecimentos que adquiriu nas unidades anteriores a esta. Pois, os assuntos que foram abordados são justamente as estruturas de programação em si, que envolvem as sequenciais, decisão e repetição.

Quando se fala em estruturas de programação, significa que você já estudou sobre linguagens de programação, formas de representação de algoritmos, tipos de dados, representação da informação no computador, variáveis e expressões, porque, sem esses conhecimentos você não teria concluído a unidade.

Ao estudar as estruturas sequenciais, aprendeu que o fluxo de execução de um algoritmo/programa é como se fosse por gravidade, iniciando nas primeiras linhas e se encerrando na última, sem interrupções ou desvios.

Nos estudos das estruturas de decisão, você percebeu que são aquelas que refletem a forma com que os computadores “pensam” para tomar as decisões. Nessa modalidade de construção, foram abordadas as estruturas de decisão do tipo **Se** e as do tipo **Escolha**. Foram mostrados diversos exemplos, tanto em algoritmo como em Pascal, com as principais formas de representar os códigos.

Quando dos estudos das estruturas de repetição, você aprendeu a construir os laços **Contados** e os **Condicionados** e, nesses, as construções **Enquanto** e **Repita**. Diversos exemplos em forma de algoritmos e programas foram disponibilizados e comentados com as principais características de cada código.

Como você deve ter implementado em Pascal no ambiente de programação Pascalzim, todos os exemplos, deve ter notado a importância em organizar os códigos com indentações, de forma tal que os códigos-fonte fiquem organizados, facilitando as manutenções, principalmente nas estruturas que são aninhadas.

Para finalizar este livro, em seguida você vai aprender como simular a execução de um algoritmo, sem usar o computador, essa técnica chama-se **Teste de Mesa**.


ANOTAÇÕES

Teste de Mesa

UNIDADE XI

UAB UNIVERSIDADE ABERTA DO BRASIL

Após a elaboração de um algoritmo/programa é importante testá-lo realizando simulações com o propósito de verificar se ele está ou não correto e se atinge os objetivos esperados. Há situações inclusive, em que você desenvolve a ideia, mas não consegue visualizar se realmente aquele algoritmo atingirá o resultado esperado, ou ainda, podem estar ocorrendo erros de lógica ou de construção do algoritmo, em que não se consegue abstrair integralmente o funcionamento do futuro programa, mesmo após implementá-lo em uma linguagem de programação.

Nesse contexto, um dos modos de realizar testes e análises do algoritmo/programa é por meio da técnica denominada **Teste de Mesa**, que permite a simulação do processo de execução de um algoritmo utilizando apenas papel e caneta/lápis.

Não há exatamente um padrão para realizar os passos de execução de um teste de mesa, a seguir, estão descritos os passos que a maioria dos autores utilizam.

- Inicialmente, deve-se identificar as variáveis envolvidas no algoritmo/programa que será simulado.
- O passo seguinte é montar um quadro com linhas e colunas, em que, cada coluna representará uma variável.
- Pode-se criar duas colunas a mais e reservar a primeira, para indicar as linhas do algoritmo/programa, que correspondem as instruções das variáveis a serem observadas. A última coluna pode ser utilizada para indicar as saídas do algoritmo/programa que sejam relevantes, nem sempre é necessário se preocupar com todas as mensagens a serem exibidas pelo programa.
- As linhas de cada variável serão utilizadas para indicar os valores que as variáveis assumem após a execução de cada instrução.
- A simulação pode ser realizada do **Início** em direção ao **Final** do algoritmo/programa, conforme a sequência de execução, ou quando o código é muito extenso, pode-se fazer o teste de mesa em trechos do programa. Deve-se preencher cada uma das linhas do quadro, com o número da linha que identifica cada instrução, seguido dos valores assumidos pelas variáveis após a execução das respectivas instruções.

Para exemplificar o **Teste de Mesa** será utilizado o algoritmo intitulado: **Media_notas_turma**, a seguir, em que os testes foram simulados para 3 entradas de notas, a saber:

Entradas	N1	N2	N3	N4
1	7	8	5	6
2	8	7	8	6
3	6	7	8	9

```
 1. Algoritmo Media_notas_turma;
 2.
 3. Var
 4.   N1, N2, N3, N4, Soma, Media, Media_total : real;
 5.   Alunos, Alunos_AP, Alunos_RP : inteiro;
 6.   continuar : caractere;
 7.
 8. Início
 9.   Alunos := 0;
10.   Alunos_AP := 0;
11.   Alunos_RP := 0;
12.   Soma   := 0;
13.
14.   Repita
15.      Escreva('Curso - Licenciatura em Computação - UEPG/EAD');
16.      Escreva('Algoritmos e Programação I');
17.      Escreva('Programa de calcular a média de 4 notas bimestrais - média 7');
18.
19.      Escreva('Entre com as quatro notas: ');
20.      Leia(N1, N2, N3, N4);
21.
22.      Media := (N1 + N2 + N3 + N4) / 4;
23.      Soma := Soma + Media;
24.      Alunos := Alunos + 1;
25.
26.      Se (Media >= 7) então
27.          Escreva('Aprovado - média: ',Media);
28.          Alunos_AP := Alunos_AP + 1;
29.      senão
30.          Escreva('Reprovado - média: ',Media);
31.          Alunos_RP := Alunos_RP + 1;
32.      Fim_se;
33.
34.      Escreva('Tecle <s> p/Sair, ou qualquer tecla p/continuar');
35.      Leia(continuar);
36.
37.   Até_que (continuar = 's');
38.
39.   Media_total := Soma / Alunos;
40.
41.   Escreva('Média total: ', Media_total);
42.   Escreva('Total de Alunos: ', Alunos);
43.   Escreva('Total de Alunos Aprovados  : ', Alunos_AP);
44.   Escreva('Total de Alunos Reprovados: ', Alunos_RP);
45.
46. Fim.
```

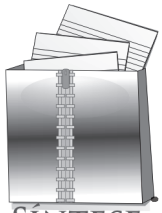
A seguir, o quadro do Teste de Mesa, para o algoritmo em questão, com os respectivos resultados, tanto das variáveis, como o resultado final de execução dele. Estude sequencialmente os valores inseridos no quadro abaixo e compare com as expressões no programa.

Linhas	N1	N2	N3	N4	Media	Soma	Alunos	Alunos_AP	Alunos_RP	Media_total	continuar	Saída
9							0					
10								0				
11									0			
12						0						
20	7	8	5	6								
22					6.5							
23						6.5						
24							1					
30												Reprovado
31									1			
35											Enter	
20	8	7	8	6								
22					7.25							
23						13.75						
24							2					
27												Aprovado
28								1				
35											Enter	
20	6	7	8	9								
22					7.5							
23						21.25						
24							3					
27												Aprovado
28								2				
35											s	
39										7.08		

Observe que o algoritmo em referência, utilizado para exemplificar o teste de mesa, possui implementado em seu código as três estruturas de programação estudadas até aqui, sequencial, decisão e repetição, que poderá ser utilizado por você como modelo para exercícios futuros.

ATIVIDADES

→ **Atividade** – implemente o algoritmo em referência Media_notas_turma no Pascalzim, melhorando-o naquilo que você julgar necessário.



SÍNTESE

Como visto a utilização da técnica do teste de mesa pode auxiliá-lo no desenvolvimento dos algoritmos e programas. É um modo também de testar o raciocínio lógico empregado em determinadas estruturas que você desenvolver.

Com relação à atividade proposta nesta unidade, ou seja, implementar o algoritmo Media_notas_turma em Pascal, além de ser um modo de exercitar a codificação dos algoritmos em programas, outro intuito é que você use esse código-fonte como um arquivo padrão, para o desenvolvimento dos demais exercícios. A proposta é fazer uma cópia dele com outro nome, por exemplo, programa_modelo. Como ele tem as três estruturas de programação implementadas, tem também uma sugestão de cabeçalho e um controle de parada e continuação de execução do programa, o que facilita o desenvolvimento de programas futuros. O intuito é reutilizar as partes de código já prontas, que podem ser comuns e escrever somente aquilo que faz parte da essência do programa que será desenvolvido, é um modo de otimizar o seu trabalho.

PALAVRAS FINAIS

Caro (a) aluno (a),

Parabéns!

Você concluiu os estudos da disciplina Algoritmos e Programa I.

Com este livro e com os demais recursos e materiais disponibilizados, você aprendeu a desenvolver algoritmos nas três estruturas de programação, sequenciais, decisão e repetição. Aprendeu também, a codificar os algoritmos na linguagem de programação Pascal, por meio do ambiente de programação do Pascalzim. Portanto, está apto a iniciar a disciplina Algoritmos e Programação II.

Possivelmente você deve ter percebido, que desenvolver algoritmos e programas tem uma certa semelhança com *arte*, pois, é necessário aplicar conhecimentos, raciocínio, lógica, usando talento e habilidade, na demonstração de uma ideia, que ficam incorporados nos códigos-fonte desenvolvidos.

Como este livro é seu, é importante guardá-lo com cuidado. Mas consultá-lo sempre que tiver dúvidas, principalmente nas próximas disciplinas, pois algumas respostas podem ser encontradas aqui.

Continue estudando e sendo perseverante!

A recompensa, tanto minha como sua, será vê-lo (a) formado (a).

Ivo Mario Mathias.

REFERÊNCIAS

Araújo, E.C. de; Hoffmann, A.B.G. **C++ Builder: implementação de algoritmos e técnicas para ambientes visuais**. Florianópolis: Visual Books, 2006.

Berg, A.C.; Figueiró, J.P. ***Lógica de Programação***. Canoas: Ed. ULBRA, 1998.

Farrer, H. et al. ***Algoritmos Estruturados***. Rio de Janeiro: Ed. Guanabara, 1985.

Forbellone, A.L.V.; Eberspächer, H.F. ***A construção de algoritmos e estrutura de dados***. São Paulo: Makron Books, 1993.

Forbellone, A.L.V.; Eberspächer, H.F. ***Lógica de Programação***. São Paulo: Makron Books, 2000.

Guimarães, A.M.; Lages, N.A.C. ***Algoritmos e Estruturas de Dados***. Rio de Janeiro: LTC, 1985.

Manzano, J.A.; Oliveira, J.F. ***Algoritmos, Lógica para Desenvolvimento de Programação***. São Paulo: Ed. Érica, 1996.

Manzano, J.A; Oliveira, J.F. ***Algoritmos: Estudo Dirigido***. Ed. Érica, 1997 .

Pintacuda, N. ***Algoritmos elementares: procedimentos básicos de programação***. Lisboa: Ed. Presença, 1988.

Pascalzim, Apostila versão 6.0.3, 2016.

Pinto, W.L. ***Introdução ao desenvolvimento de algoritmos e estruturas de dados***. São Paulo: Érica, 1990.

Saliba, W.L.C. ***Técnicas de programação: uma abordagem estruturada***. São Paulo: Makron-Books, 1992.

Salveti, D.D.; Barbosa, L.M. ***Algoritmos***, Makron-Books, 1998 .

Tremblay, J-P.; Bunt, R.B. ***Ciência dos Computadores: Uma Abordagem Algorítmica***. São Paulo: McGraw-Hill do Brasil, 1983.

Venancio C.F. ***Desenvolvimento de Algoritmos, Uma Nova Abordagem***. Ed. Érica, 1998 .

Wirth, N. **Algoritmos *e estruturas de dados***. Rio de Janeiro: Prentice-Hall do Brasil, 1989.

ANEXOS

Respostas das atividades:

Unidade V

(I) 0	(R) –0.001	(R) –0.0
(I) 1	(R) +0.05	(B) .V.
(R) 0.0	(I) +3257	(N) V
(R) 0.	(L) "a"	(L) 'abc'
(I) –1	(L) "+3257"	(N) F
(I) –32	(L) '+3257.'	(N) .F
(I) +36	(L) "–0.0"	(L) "V"
(R) +32.	(L) '.F.'	(B) .F.

Unidade VII

1)

(V) abc	(N) 3abc	(V) a
(N) 123a	(V) _a	(V) acd1
(N) _	(V) Aa	(N) 1
(V) A123	(V) _1	(V) A0123
(V) a123	(V) _a123	(V) b312
(N) AB CDE	(N) guarda-chuva	(N) etc.

2)

Nome simbólico	Posição inicial	Tipo de dado
X	0	**INTEIRO**
Y	2	**INTEIRO**
NOME	4	**LITERAL[20]**
PROFISSÃO	24	**LITERAL[20]**
RUA	44	**LITERAL[30]**
NUMERO	74	**INTEIRO**
RENDA	76	**REAL**

Unidade VIII

Expressão	Resultado
A .OU. B	V
A .E. B	F
.NÃO. A	F
X = Y	F
X = (Y/2)	V
R = S	F
S = T	F
R <> S	V
R > S	V
S > T	F
((A .OU. B) .OU. (X = Y) .OU. (S = T))	((V .OU. F) .OU. (2.5 = 5.0) .OU. (S = T)) (V .OU. F .OU. F) V

Unidade X

Seção 2 - Saída é Valores de a,b,c são: 222

Seção 3

```
Algoritmo Exemplo_repeticao_7;

 Var
   num : inteiro;

 Início
   num:=0;

   Repita
     Escreva(num);
     num:=num+1;
   até que (num > 100);

 Fim.
```

```
Program Exemplo_repeticao_7;

 var
   num : integer;

 Begin
   num:=0;

   Repeat
     write (' ',  num);
     num:=num+1;
   until (num > 100);

   readkey;
 End.
```

```
Algoritmo Exemplo_repeticao_8;

 Var
    num, soma : inteiro;

 Início
    num := 1;
    soma := 0;

    Escreva(' A soma de: ');

    Repita
       Escreva(num);
       soma := soma + num;
       num := num + 1;
    até que (num > 10);

 Escreva(' = ',  soma);

 Fim.
```

```
Program Exemplo_repeticao_8;

 var
   num, soma : integer;

 Begin
   num := 1;
   soma := 0;

   write (' A soma de: ');

   repeat
     write (' ',  num);
     soma := soma + num;
     num := num + 1;
   until (num > 10);

   write (' = ',  soma);

  readkey;

 End.
```

NOTAS SOBRE O AUTOR

Ivo Mario Mathias

Doutor em Agronomia pela Universidade Estadual Paulista - Júlio de Mesquita Filho - UNESP. Mestre em Informática pela Universidade Federal do Paraná. Graduado em Administração pela Universidade Estadual de Ponta Grossa. Graduado em Ciências Contábeis pela Universidade Estadual do Ponta Grossa. Professor Associado da Universidade Estadual de Ponta Grossa, desde 1987, lotado junto ao Departamento de Informática. Atuei como Professor permanente no Mestrado em Computação Aplicada. Trabalhei como pesquisador e orientador de iniciação científica em vários grupos de pesquisa, dentre eles fui Líder do grupo de pesquisa em informática aplicada a agricultura INFOAGRO-UEPG registrado junto ao CNPq. Ministrei disciplinas de Algoritmos e Programação de Computadores, Estruturas de Programação, Lógica Computacional, Fluxos em Redes, Organização de Computadores, Inteligência Artificial e Redes Neurais Aplicadas a Agricultura.

Ministério
da Educação

UEPG
UNIVERSIDADE ESTADUAL DE PONTA GROSSA

nute@d
Núcleo de Tecnologia e Educação Aberta e a Distância