UNIP
UNIVERSIDADE PAULISTA

Interativa

Linguagem de Programação Aplicada

Autores: Prof. Nilson Magalhães Bueno
Prof. Rogério da Costa Gião
Colaboradores: Prof. Ataíde Pereira Cardoso Júnior
Prof. José Carlos Morilla

Professores conteudistas: Nilson Magalhães Bueno / Rogério da Costa Gião

Nilson Magalhães Bueno

Mestre em Ciências na área de Sistemas de Potência no Programa de Engenharia Elétrica da Escola Politécnica da Universidade de São Paulo (USP), em junho de 2012. Pós-graduado em Especialização em Automação Industrial no Programa de Educação Continuada em Engenharia Elétrica (Pece) da Escola Politécnica da Universidade de São Paulo (USP) em 2007. Pós-graduado em Formação em Educação à Distância – EaD pela Universidade Paulista (UNIP). Bacharel em Análise de Sistemas pela UNIP em 2002. Técnico em Instrumentação Industrial pela Escola Fortec – SV/SP em 1987. Experiência profissional de 29 anos na indústria siderúrgica (Cosipa/Usiminas) junto à gerência de automação industrial. Professor e coordenador dos cursos superiores de tecnologias de Automação Industrial, Análise e Desenvolvimento de Sistemas, Gestão da Tecnologia da Informação e Redes de Computadores na UNIP *campi* Santos e Chácara Santo Antônio. Docente das disciplinas relacionadas à Tecnologia da Informação nos cursos superiores de Gestão de Logística, Comércio Exterior, Recursos Humanos, Portuária e Segurança Privada. Coordenador da pós-graduação em Engenharia de Redes e Sistemas de Telecomunicações na UNIP. Professor titular mestre nos cursos de Engenharia de Produção, Análise e Desenvolvimento de Sistemas e Pós-Graduação na Unimonte, campus Santos/SP.

Rogério da Costa Gião

Professor nas áreas de Tecnologia da Informação e Engenharia Elétrica, cursando mestrado em Engenharia Mecânica na Universidade Santa Cecília de Santos-SP, pós-graduado em Gerenciamento de Projetos pela Fundação Getúlio Vargas de São Paulo, pós-graduado em Automação Industrial pela Universidade Católica de Santos e graduado em Engenharia Eletrônica pela Fundação Armando Álvares Penteado (Faap). Possui experiência de vinte anos na área de engenharia de projetos de sistemas voltados para automação industrial, adquiridos na empresa Usiminas, no estado de São Paulo. Atua como professor nas áreas de Tecnologia da Informação, Sistemas de Informação, Ciências da Computação e Engenharia Elétrica/Eletrônica.

Dados Internacionais de Catalogação na Publicação (CIP)

B928L Bueno, Nilson Magalhães.

Linguagem de Programação Aplicada. / Nilson Magalhães Bueno, Rogério da Costa Gião. – São Paulo: Editora Sol, 2020

176 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. Linguagem de programação aplicada. 2. Administração de redes. 3. Variáveis em VBScript. I. Gião, Rogério da Costa. II. Título.

CDU 681.3

U508.57 – 20

Sumário

Linguagem de Programação Aplicada

Unidade IV

APRESENTAÇÃO

Neste curso, o aluno aprende a dimensionar, configurar equipamentos, a trabalhar com sistemas de gerenciamento e segurança de redes e simulações de ambientes virtuais e cloud computing com alto desempenho. Além dos objetivos gerais citados anteriormente, o curso tem os seguintes objetivos específicos:

- formar profissionais capazes de definir parâmetros de utilização de redes de computadores;
- formar profissionais capazes de manterem-se atualizados (aprender a aprender);
- capacitar profissionais para definir e avaliar arquitetura de redes de computadores;
- despertar e incentivar o aluno na pesquisa científica como ferramenta no desenvolvimento de projetos experimentais.

Para que o egresso alcance o perfil esperado, o curso superior de Tecnologia em Redes de Computadores proporcionará meios para o desenvolvimento das seguintes competências:

- conhecer e analisar seu campo de atuação profissional e seus desafios contemporâneos;
- desenvolver visão e raciocínio estratégico para a definição e implementação dos princípios básicos de redes de computadores;
- conduzir projetos e liderar equipes relacionadas às redes de computadores, com o uso de metodologias e processos avançados;
- elaborar e executar planos de aplicação das normas de instalações de redes de computadores;
- avaliar e propor novas tecnologias e identificar oportunidades para soluções inovadoras;
- conduzir projetos, programas e atividades de aplicação de redes de computadores com qualidade e segurança;
- comunicar-se de forma eficaz e adequada às diversas situações, tanto de forma oral quanto escrita;
- gerenciar recursos e coordenar projetos de redes de computadores;
- identificar, analisar, avaliar e resolver problemas em redes de computadores, empregando bases científicas, tecnológicas, senso crítico e criatividade;
- gerenciar e manter os projetos lógicos e físicos de redes de computadores;
- implantar e documentar rotinas;

- implantar a conectividade entre sistemas heterogêneos;
- propor soluções de problemas relacionados à comunicação de dados;
- transformar as ideias e projetos em produtos e serviços na área de redes de computadores com visão empreendedora.

Este livro-texto tem como objetivo abordar, de forma clara e abrangente, as principais linguagens de script, todas muito utilizadas na administração de redes de computadores, enfatizando VBScript e Bash Script. A pretensão não é apenas apresentar como utilizar essas linguagens, mas também analisar como problemas reais podem ser sanados com a utilização de scripts com exemplos práticos e estudos de casos. Para tanto, alguns dos problemas e necessidades comuns de redes Linux e Windows são considerados.

O funcionamento efetivo e livre de falhas é a parte mais importante das redes de computadores. Verifica-se que a falha total ou parcial de redes públicas e/ou privadas acarreta prejuízos muitas vezes difíceis de calcular. As respostas da administração de redes devem ser rápidas e eficientes. Entre as ferramentas de apoio a essa área, para os novos e já conhecidos desafios da administração de redes, as linguagens de script se apresentam como uma opção poderosa, que vem cada vez mais sendo adotadas. Com a utilização de scripts, é possível automatizar tarefas, combinar ações, armazenar, processar e interpretar arquivos de sistema, além de uma infinidade de outras aplicações.

Não se esgota aqui todo o aprendizado necessário ao aluno de redes de computadores, em que o complemento aos seus estudos ocorrerá através da especialização a este amplo campo de novas tecnologias existentes ao profissional moderno em contato com as melhores práticas de gestão das redes de dados e telecomunicações.

INTRODUÇÃO

O objetivo deste livro-texto é levar o aluno ao entendimento de como um dispositivo microprocessador, por exemplo, um computador (PC), um dispositivo de rede de dados ou de telecomunicações, deve ser programado através de linguagens de mais alto nível, para que possam executar as tarefas de acordo com as necessidades dos seus usuários. Tais dispositivos podem desempenhar diversas tarefas, sendo necessário, para isso, que sejam detalhadas passo a passo, de forma compreensível, pelo processador da máquina, utilizando aquilo que se chama de programa. Nesse sentido, um programa de um dispositivo que possua um microprocessador nada mais é que um algoritmo escrito de forma compreensível por sua unidade de processamento. Não podemos entender um algoritmo como sendo uma única solução para um determinado problema, vamos entendê-lo como um caminho que deve ser seguido para atingirmos um determinado objetivo.

Compreender a evolução dos sistemas computacionais, desde sua criação até os tempos atuais, em que várias necessidades de seu entendimento foram aparecendo, tanto no âmbito do hardware como no do software, também faz parte do estudo. O conceito de redes de computadores, bem como a sua gestão, são tópicos presentes, fazendo com que o aluno tenha contato com o monitoramento e controle, associados às suas respectivas ferramentas e à sintaxe de alguns comandos. O objetivo não

é trazer toda a elucidação que envolve o assunto, mas sim mostrar a importância básica de alguns comandos ao profissional que lida com os diversos recursos das redes de computadores.

As respostas da administração de redes devem ser rápidas e eficientes. Entre as ferramentas de apoio a essa área, as linguagens de script vêm se apresentando como uma opção poderosa. Com scripts, podemos automatizar tarefas, combinar ações, armazenar, processar e interpretar arquivos de sistema, além de uma infinidade de outras aplicações, visando armazenar e transportar os pacotes que se deseja transmitir. Para os novos e já conhecidos desafios da administração de redes, os scripts são uma ferramenta que deve ser cada vez mais adotada. Para tanto, o aluno entrará em contato básico com a linguagem de programação VBScript.

Por fim, a pretensão não é apenas apresentar como a linguagem de programação VBScript elucida problemas e necessidades comuns de redes presentes nos sistemas operacionais Windows e Linux, mas também auxiliar os profissionais dessa área a realizar um gerenciamento eficiente das redes de computadores, com o intuito de evitar problemas que possam parar as comunicações, e implementar ferramentas que melhorem a performance das redes. Um breve histórico do sistema operacional Linux, bem como alguns comandos de inicialização desse sistema operacional, faz-se presente. A introdução a alguns scripts Shell, que possuem o objetivo de manipular variáveis e alguns comandos de repetição, completam o entendimento da matéria.

Unidade I

1 ALGORITMOS

1.1 Conceito e definições de algoritmos

O processo de transferir para máquinas as tarefas que normalmente são executadas pelo homem é o conceito que conhecemos por automatização de processos. Os tipos de máquinas são os mais diversos possível, desde máquinas mecânicas até eletrônicas, todas elas com os seus graus de desenvolvimento e tecnologias embarcadas. Dentre as máquinas eletrônicas, podemos considerar computadores, telefones celulares, tablets, componentes específicos para determinadas atividades, como switches de rede. Para que qualquer uma dessas máquinas possa executar as tarefas adequadamente, dentro do que se espera, em termos de rapidez e qualidade, é necessário que seja realizado um determinado tipo de programação dentro dela. Para que essa programação possa ser eficiente, é necessário o detalhamento em sequência das atividades que a máquina deve realizar, a fim de atingir o esperado objetivo.

A sequência correta de atividades a ser executada por uma máquina, o que irá garantir a execução correta do processo, é o que podemos chamar de algoritmo. Definimos algoritmo como uma sequência de passos, muito bem definidos, que devem ser seguidos, conforme planejado, para atingir um determinado objetivo. É importante frisar que o número de passos irá depender da complexidade do problema, mas deve ser finito, e o algoritmo deve ter início e fim bem definidos.

Como resultado de um algoritmo, devemos chegar a uma saída que deve representar o resultado esperado para todo o raciocínio que foi utilizado no seu desenvolvimento. Um bom algoritmo é aquele que consegue chegar a um determinado resultado da forma mais eficiente possível.

Sempre que um algoritmo é elaborado, deve-se sempre questioná-lo, ou seja, avaliá-lo de forma a afirmar se ele realmente é a melhor solução para um determinado problema; se não for, devemos modificá-lo. Tudo isso para que se torne a solução com o menor número de passos, conseguindo mesmo assim chegar ao resultado esperado.

Observação

A elaboração de um bom algoritmo depende da aplicação correta do raciocínio lógico que originou a sequência de atividades para a resolução do problema. O raciocínio lógico é uma particularidade de cada indivíduo. Por esse motivo, para um mesmo problema, podemos encontrar diferentes algoritmos.

Algumas regras são importantes para que possamos conseguir um algoritmo eficiente, podemos citar:

- o número de atividades, ou passos, deve ser finito;
- um passo deve ser preciso, ou seja, não deve de forma alguma gerar dúvidas em relação ao que está sendo feito;
- deve possuir, uma ou mais de uma, saídas para determinadas situações;
- deve ter um ponto que determina o seu fim, que deve ser sempre atingido, independentemente do caminho que tenha se tomado.

Saiba mais

Para um melhor desenvolvimento da aplicação de lógicas em linguagens de programação, seguem duas referências de livros que demonstram as técnicas e melhores práticas para resolução de problemas:

HEGENBERG, L. *Lógica*: o cálculo sentencial, cálculo de predicados, cálculo com igualdade. 3. ed. São Paulo: EPU, 2012.

BARBIERI FILHO, P.; HETEM JR., A. *Lógica para computação*. São Paulo: LTC, 2012.

1.1.1 Formas de representação de algoritmos

Ao longo dos tempos, foram desenvolvidas várias formas de representação para algoritmos. Existem representações em nível lógico, em nível de linguagem usual das pessoas e em nível de figuras representativas. Não podemos afirmar se uma é melhor ou pior do que as outras, o que existe é aquela que mais se adequa a uma determinada situação, e também da preferência pessoal de quem a está desenvolvendo. Para essa escolha, não devemos esquecer a qual nível de detalhes se deseja alcançar, já que esse é um parâmetro que muda de um tipo para outro, podendo interferir diretamente no entendimento de um certo algoritmo por quem o irá analisar.

Serão abordadas quatro formas de representação de algoritmos, são elas:

- descrição narrativa;
- fluxograma tradicional;
- diagrama de Chapin;
- pseudocódigo ou portugol.

1.1.2 Descrição narrativa

A descrição narrativa utiliza a mesma forma utilizada pelas pessoas para explicar as situações em linguagem falada. Nesse caso, deve-se considerar a língua utilizada para sua descrição. Podemos tomar um exemplo para esse tipo de representação, para isso, a fim de assar um bolo de fubá, devemos executar as seguintes atividades:

I – separar os ingredientes: 200 g de fubá, 200 g de açúcar, três xícaras de manteiga, 200 g de farinha de trigo, um copo de leite, duas colheres de fermento químico;

II – misturar todos os ingredientes em um recipiente;

III – colocar a mistura em uma assadeira untada;

IV – levar ao forno pré-aquecido;

V – manter a mistura no forno por mais ou menos 35 minutos;

VI – retirar o bolo do forno e aguardar esfriar para poder consumir.

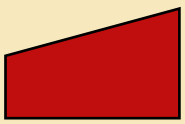
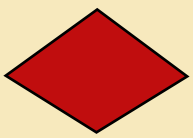
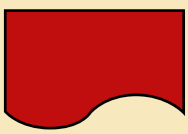
A descrição narrativa, assim, não é a forma mais adequada para utilização, devido ao fato de utilizar a linguagem falada como base, que pode levar a condições imprecisas ou que gerem várias interpretações. No exemplo anterior, podemos verificar várias situações de dúvida, em que diferentes pessoas podem entender a atividade de forma diferente. Peguemos o passo I; nele, a definição de três xícaras de manteiga e um copo de leite podem gerar dúvidas já que não se define o tipo de xícara e o tamanho do copo. O passo V declara o tempo como mais ou menos 35 minutos, ou seja, não há uma precisão no que se deve fazer. Esses exemplos sempre devem ser evitados em algoritmos.

1.1.3 Fluxograma convencional

A representação de fluxograma convencional é a forma que faz uso de formas geométricas para indicar quais são as ações que devem ser executadas em cada um dos passos do algoritmo. Essa forma de representação costuma ser muito utilizada, pois a forma geométrica não deixa dúvidas em relação ao que se deve fazer naquele passo. Além disso, a utilização de figuras facilita o entendimento pelas pessoas que irão analisar o algoritmo.

Nesse tipo de representação, cada tipo de atividade é associado a uma determinada figura, o quadro seguinte traz as possíveis ações a serem executadas e suas respectivas representações geométricas.

Quadro 1 – Principais símbolos de um fluxograma

Símbolo	Função
	Início e fim do algoritmo
	Indica cálculo e atribuições de valores
	Indica a entrada de dados
	Indica uma decisão com possibilidades de desvios
	Indica saída de dados
	Indica o fluxo de dados. Serve também para conectar os blocos ou símbolos existentes

Uma representação do tipo fluxograma deve sempre começar indicando seu início, através da utilização do símbolo de início; em seguida, cada passo deve ser indicado pelo símbolo correspondente a ele; e, no final, o símbolo de fim, que pode ser mais de um, também deve ser inserido.

A sequência dos passos que deve ser seguida é representada pelo símbolo da seta, demonstrando a sequência lógica que o algoritmo deve seguir. Vários caminhos ao longo do algoritmo podem ser seguidos, e até mesmo ações podem ser executadas mais de uma vez, o que vai determinar essa sequência é o resultado que cada passo irá gerar e o caminho lógico que depende do passo executado. Sempre irá existir uma saída de um determinado passo, seguindo a lógica, até chegar ao final do algoritmo, chegando, dessa forma, ao resultado esperado.

Essa forma de representação é uma das mais utilizadas no ambiente da programação, pois permite construir algoritmos com diversos níveis de detalhamento, desde os mais simples aos mais complexos. A sequência estabelecida pelo fluxograma é a representação lógica para um determinado problema e servirá de base para a elaboração de um programa de computador.

Veja, a seguir, a representação de um algoritmo em fluxograma para cálculo da média final de um aluno, através do cálculo da média aritmética entre duas notas.

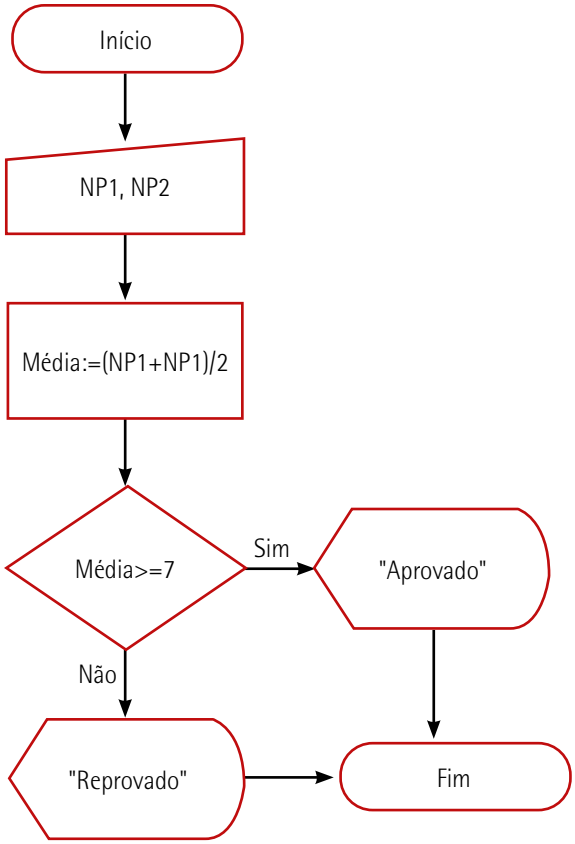


Figura 1 – Exemplo de fluxograma convencional

Podemos citar, como principal vantagem da utilização desse tipo de algoritmo, o fácil entendimento do raciocínio lógico que foi desenvolvido. Porém existem duas desvantagens que devem ser citadas, uma referente à necessidade do conhecimento dos símbolos que definem as ações e outra referente à dificuldade que existe para realizar qualquer alteração no algoritmo.

1.1.4 Diagrama de Chapin

Esse tipo de representação foi desenvolvida por Ned Chapin, que teve a intenção de criar um algoritmo que apresentasse uma visão de hierarquia estruturada de uma determinada lógica para resolução de um problema.

A maior vantagem da utilização desse tipo de algoritmo é a possibilidade de demonstrar, de forma bem transparente, o ponto de entrada e o ponto de saída, além das estruturas de sequência, seleção e divisão do fluxo lógico, ficando fácil a visualização de ações incluídas e recursivas.

A figura a seguir apresenta um exemplo de algoritmo na forma de diagrama de Chapin, para o cálculo da média de um aluno mediante o cálculo aritmético entre duas notas.

Observação

Ned Chapin foi um líder no campo da ciência da computação, particularmente na disciplina de manutenção de software. Faleceu em 27 de dezembro de 2014, aos 61 anos. Nasceu na Península Olímpica e cresceu vivendo em vários locais em Washington, Oregon e Califórnia. Aos 60 anos, ele assumiu o cargo de professor e, posteriormente, professor emérito de sistemas de informação na Universidade Estadual da Califórnia.

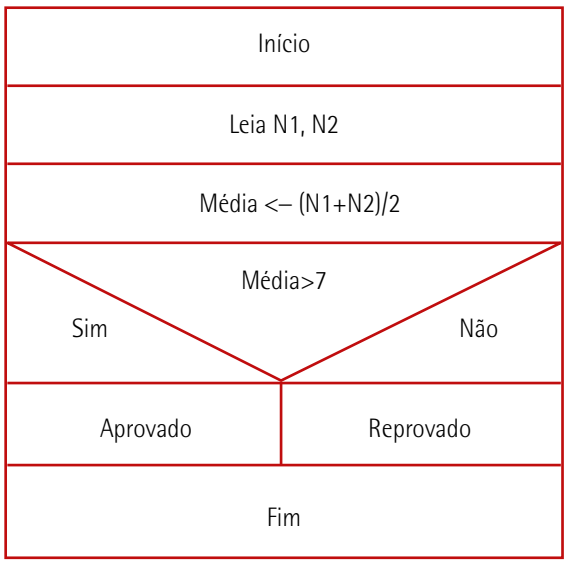


Figura 2 – Diagrama de Chapin

Observação

Um diagrama de Nassi-Shneiderman (NSD), na programação de computador, é uma representação gráfica do projeto para a programação estruturada. Esse tipo de diagrama foi desenvolvido em 1972, por Isaac Nassi e Ben Shneiderman, que eram ambos estudantes de pós-graduação em SUNY-Stony Brook. Esses diagramas também são chamados de estrutogramas, como mostram as estruturas de um programa.

1.2 Pseudocódigo

Pseudocódigo, ou também conhecido como português estruturado, é uma forma de representação de algoritmos que tem como característica principal a semelhança com as estruturas de linguagens de programação. Trata-se de uma representação muito aceita pelos desenvolvedores, e possibilita a criação de algoritmos muito ricos em detalhes. Devido a essas características, essa forma de representação permite uma migração muito simples para a linguagem de programação que será utilizada.

1.2.1 Formatação do algoritmo em pseudocódigo

Um algoritmo em pseudocódigo deve respeitar uma estrutura básica, com os campos mínimos para sua confecção. Essa estrutura é obrigatória e não deve ser substituída por nenhuma outra identificação nas suas áreas. Veja a estrutura de um algoritmo desse tipo a seguir:

```
Algoritmo <nome_do_algoritmo>
          <declaracao_de_variaveis>
          <subalgoritmos>
Inicio
          <corpo_do_algoritmo>
Fim
```

Sendo:

- **Algoritmo** é o indicador que apresenta um novo algoritmo.
- <**nome_do_algoritmo**> representa a identificação simbólica do algoritmo, visando sua identificação perante demais algoritmos.
- <**declaracao_de_variaveis**> é a área, dentro da estrutura do algoritmo, destinada à declaração das variáveis que serão utilizadas ao longo do algoritmo. As variáveis podem ser globais ou locais; a explicação de cada uma será apresentada mais adiante.
- <**subalgoritmos**> é a área destinada à declaração de subalgoritmos que auxiliam o desenvolvimento do principal. Esse assunto também será abordado adiante.
- **Inicio** e **Fim** são os parâmetros que definem os pontos de início e fim do corpo principal do algoritmo. É nessa área em que todas as instruções são definidas.

Podemos ver, a seguir, um exemplo de um algoritmo em pseudocódigo, vamos utilizar o mesmo cálculo da média de um aluno.

```
Algoritmo Media
Var NP1, NP2, Media_Final
Inicio
    Leia NP1, NP2
    Media_Final := (NP1+NP2) / 2
    Se Media_Final >= 7 Entao
          Escreva " Aluno Aprovado "
  Senao
          Escreva " Aluno Reprovado "
Fim
```

Podemos citar, como principais vantagens dessa representação, o fácil entendimento do algoritmo por parte de quem o estiver analisando e a facilidade de migração para qualquer linguagem de programação. Podemos citar, como desvantagem, a necessidade de conhecimento da estrutura e da sintaxe das instruções utilizadas.

1.3 Orientações para elaboração de um algoritmo

O desenvolvimento de um algoritmo depende de alguns pontos que devemos levar em conta, para que ele consiga resolver determinado problema da melhor forma. O tipo do algoritmo que se adota simplesmente indica a forma como será apresentado, sendo mais importante o raciocínio lógico que foi aplicado para a resolução. Quando apresentamos um problema para diversas pessoas, geralmente obtemos delas soluções diferentes, mesmo que o resultado final seja alcançado. Cada um pensa de uma forma, o que gera raciocínios lógicos diferentes para um mesmo problema. Sendo assim, obteremos algoritmos diferentes para a resolução de um determinado problema, e todos podem estar certos se analisarmos o resultado esperado. O que define um bom algoritmo é a forma como se chegou ao resultado final, ou seja, se ele conseguiu obter o resultado com o menor número de instruções possível. Um algoritmo mais simples irá gerar um programa de computador mais otimizado, diminuindo o tempo de processamento e, consequentemente, chegando mais rapidamente ao resultado final.

Existem algumas recomendações para a elaboração de um bom algoritmo, porém somente a prática trará resultados melhores para os desenvolvedores. A principal atividade no desenvolvimento de um sistema não está na codificação do programa de computador, mas sim na forma como foi pensada a resolução do problema e, consequentemente, na elaboração do algoritmo. Veremos, adiante, que a elaboração do programa se resume à transformação do algoritmo no código-fonte do programa, que simplesmente é a aplicação da sintaxe da linguagem ao que foi pensado.

Para a elaboração de um bom algoritmo, podemos citar algumas orientações que podem auxiliar na elaboração:

I – Faça uma análise do problema, de forma a levantar os principais pontos que devem ser considerados para sua resolução. Esta etapa pode ser refeita quantas vezes forem necessárias.

II – Após a análise, verifique se todos os pontos levantados foram entendidos, de forma que não fique nenhuma dúvida sobre cada um deles. Muitas vezes é necessário realizar novas análises, para esclarecimentos.

III – Faça o levantamento de todas as saídas que devem ser obtidas com a execução do algoritmo.

IV – Faça um levantamento das entradas que serão necessárias para que o fluxo do algoritmo funcione conforme levantado. Essas entradas devem ser bem definidas, pois elas são as informações iniciais para o andamento do algoritmo.

V – Defina qual será a linha principal de sequência do algoritmo, aquele que será o raciocínio base para a resolução do problema.

VI – Com a linha base definida, verifique quais informações adicionais serão necessárias ao longo da execução do algoritmo. Muitas vezes esta etapa vai se esclarecendo ao longo do desenvolvimento do algoritmo.

VII – A validação das etapas do algoritmo precisa ser feita com muito cuidado, realizando testes com dados de entrada e verificando as saídas de cada etapa.

VIII – Após as etapas anteriores terem sido concluídas, reveja o algoritmo com o intuito de formatá-lo dentro das boas práticas de desenvolvimento. Algumas dessas práticas serão abordadas ao longo deste material.

Mesmo com todas as recomendações anteriores, há algumas que devemos seguir, com o intuito de simplificar a implementação de um algoritmo:

I – Objetividade é algo que sempre se deve almejar na descrição das etapas. A utilização de um verbo por frase auxilia muito nesse objetivo.

II – A elaboração deve ser simples, de forma a implementar um algoritmo que seja de fácil entendimento a qualquer pessoa.

III – O uso de frases simples, com palavras objetivas, é muito importante para um bom entendimento.

IV – Nunca utilize palavras e frases que possam levar a mais de uma interpretação, evitando dúvidas no entendimento.

Exemplo de aplicação

Desde que os celulares e os smartphones estão no mercado, os telefones públicos estão quase esquecidos e nem sempre as pessoas sabem mais como utilizá-los. Às vezes ocorrem emergências, principalmente quando você percebe que se esqueceu de carregar o celular, está longe de casa e precisa fazer uma ligação urgente. Pensando nessa situação, faça um algoritmo não computacional, cujo objetivo é usar um telefone público.

- Tirar o fone do gancho.
- Ouvir o sinal de linha.
- Introduzir o cartão.
- Teclar o número desejado.
- Se der o sinal de chamar:
  - conversar;

– desligar;

– retirar o cartão.

- Se não der o sinal de chamar:

– colocar o fone no gancho;

– repetir.

1.4 Tipos de dados

Um computador é um dispositivo eletrônico que tem a função de manipular informações através de instruções que são inseridas em programas de computador. As informações que são manipuladas por ele devem estar classificadas em tipos conhecidos pelo computador, pois, caso contrário, ele não conseguirá entender a informação e, consequentemente, não conseguirá manipulá-la conforme desejado. Podemos classificar as informações em dois grupos:

- a primeira é a instrução; instruções definem como os dados serão manipulados pelo computador;
- a segunda são os dados na sua essência, eles quantificam as informações que devem ser processadas pelo sistema.

Em relação aos dados, podemos classificá-los em alguns tipos, que definem como o computador vai tratar cada um deles através das instruções correspondentes à manipulação dos dados. A classificação citada não segue os tipos utilizados nas linguagens de programação, mas sim uma forma de representar os dados de forma que sejam aplicados a qualquer uma das linguagens existentes.

- **Inteiros**: classificamos como dados inteiros aqueles que são numéricos, tanto positivos como negativos. Neste grupo não se enquadram os dados fracionários. Exemplos: 0, 10, -635.
- **Reais**: classificamos como dados reais os dados numéricos, tanto positivos como negativos, incluindo os fracionários. Exemplos: 10, 0, 8,6, -635.
- **Caracteres**: classificamos como dados caracteres qualquer sequência que contenha números, letras e caracteres especiais. Essas sequências devem ser utilizadas sempre entre aspas, delimitando ao sistema seu início e seu fim. Algumas denominações são atribuídas a esses tipos de sequência, tais como alfanumérico, literal ou string. Exemplo: algoritmo, escola de programação, telefone (1234-5678).
- **Lógicos**: classificamos como dados lógicos aqueles aos quais são atribuídos valores de verdadeiro ou falso, não podendo receber qualquer outro valor. Esses valores geralmente são originados de operações lógicas, e também são chamados de tipo booleano. Essa denominação surgiu graças ao matemático George Boole, que desenvolveu a área da matemática lógica.

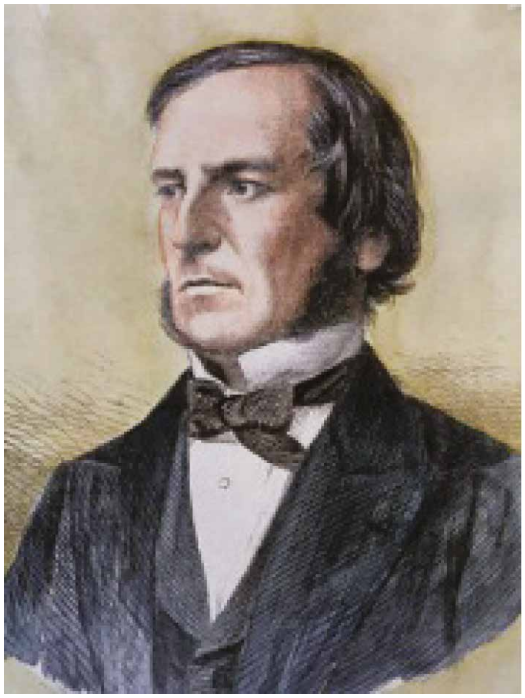


Figura 3 – George Boole

Observação

George Boole nasceu em 1815 e faleceu em 1864. Foi um matemático e filósofo britânico, criador da álgebra booleana, fundamental para o desenvolvimento da computação moderna. Em 1838, publicou o pequeno livro *A Análise Matemática da Lógica*, sua primeira contribuição para o vasto assunto, que o tornaria famoso pela ousadia e perspicácia de sua visão.

1.5 Variáveis

1.5.1 Manipulação de dados

Um computador manipula dados que ficam armazenados em sua memória, sendo que cada dado fica alocado em um espaço determinado, que vai depender do tipo de cada um, ou seja, cada tipo de dado necessita de um espaço diferente na memória do computador. Não existe possibilidade de dois dados compartilharem um mesmo espaço de memória.

Como a quantidade de dados manipulada pelo computador é muito grande, ele precisa identificar o local da memória em que se encontra cada um deles. A forma utilizada para identificar uma determinada área de memória é lhe atribuindo um nome, permitindo que o conteúdo de um espaço de memória possa ser acessado a qualquer momento.

1.5.2 Conceito e função de variáveis

O conceito de variáveis estipula tudo que pode sofrer alterações, que é instável ou mutável.

Os computadores trabalham acessando os dados na memória. Para que isso seja possível, eles precisam reconhecer os dados, manipulá-los e saber onde começa e onde termina a área da memória ocupada pelo dado. Esse é um processo muito complexo. Com o intuito de simplificar esse acesso aos dados, criou-se o recurso de utilização de variáveis, que tem como objetivo armazenar o conteúdo relativo a uma determinada informação.

Na definição de variáveis através de uma denominação, que estará responsável pelo armazenamento de um dado, que, por sua vez, pode sofrer alterações ao longo de sua existência, sempre ocupará o mesmo espaço na memória. Por meio do nome, o computador sabe exatamente onde está um determinado dado, executa a manipulação necessária devolvendo no mesmo endereço o novo conteúdo.

Uma variável possui três parâmetros fundamentais para que o computador possa acessá-la corretamente, são eles:

- nome;
- tipo de dado ao qual corresponde o seu conteúdo;
- conteúdo que representa a informação nela contida.

Um dos parâmetros das variáveis é o seu nome. Por meio dele, uma variável se diferencia da outra, não podendo existir duas com o mesmo nome em um determinado programa computacional.

Em algoritmos, devemos seguir algumas orientações para a criação de nomes de variáveis. Essas orientações não são obrigatórias, pois a criação de nomes é livre para quem estiver criando o algoritmo; elas servem para sua padronização, de forma a tornar o entendimento mais simples por quem o estiver analisando. Lembre-se que os nomes criados para variáveis nos algoritmos normalmente são levados para os programas, considerando aí as regras estabelecidas nas linguagens de programação. As orientações são:

- é importante que o nome de uma variável comece sempre por uma letra;
- o nome não deve conter nenhum caractere especial com exceção da sublinha;
- o nome não deve conter nenhum espaço em branco;
- não se pode utilizar o nome de uma instrução de programa para atribuir a uma variável, pois o computador não saberá identificar o que é instrução ou dado a ser manipulado.

Exemplos: SALARIO, ANO_NASCIMENTO, nome_funcionario, data_1.

Para criação de variáveis, é importante a escolha de nomes que representem a função que ela irá ter no programa. Dessa forma, o entendimento da variável no contexto do algoritmo fica mais simples.

Devemos tomar bastante cuidado com o tipo de dado que vamos atribuir a uma variável, pois, como vimos, cada tipo ocupa um espaço diferente na memória. Se classificada de forma errada, pode-se solicitar mais espaço que o necessário ao sistema ou, em um caso pior, pode-se solicitar menos espeço do que o necessário, correndo o risco de perder informações importantes.

Sempre que definimos uma variável para um determinado sistema, solicitamos ao computador uma alocação de memória necessária para o seu armazenamento. Essa área fica alocada durante a execução do programa, sem poder ser alterada. Porém o seu conteúdo, ou seja, a informação nela contida, pode ser manipulado e ter o seu conteúdo alterado quantas vezes for solicitado pelo programa que a manipula. A área de memória fica alocada até que o programa seja encerrado, liberando assim o espaço para outra utilidade.

Observação

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

1.5.3 Definição de variáveis em algoritmos

Como vimos anteriormente, para que um espaço de memória possa ser alocado, é necessário que realizemos as definições de variáveis, dessa forma, o computador saberá onde está cada conteúdo referenciado ao longo do algoritmo. É importante que façamos a definição delas antes que sejam manipuladas, pois, caso contrário, estaremos solicitando a manipulação de uma área que ainda não foi alocada pelo sistema.

Algumas linguagens de programação permitem que as variáveis não sejam declaradas, criando-as conforme são solicitadas ao longo do corpo do programa, porém, em algoritmos, adotaremos a filosofia de declará-las, para que eles possam ser transcritos para qualquer linguagem de programação, mesmo os que não possuem esse recurso.

Quadro 2

Algumas da principais linguagens de programação	
Linguagens históricas	ALGOL • APL • Assembly • AWK • B • **BASIC** • BCPL • COBOL • CPL • Forth • **Fortran** • Lisp • Logo • Simula
Linguagens acadêmicas	Gödel • Haskell • Icon • Lisp • Logo • Lua • **Pascal** • Prolog • Python • Scala • Scheme • Scratch • Simula
Linguagens proprietárias	ABAP • ActionScript • COBOL • Delphi • MATLAB • PL/SQL • RPG • Scratch • Transact-SQL • Visual Basic
Linguagens não proprietárias	Ada • Assembly • **C** • C++ • C♯• Icon • Lisp • Logo • Object Pascal • Objective-C • Pascal • Scheme
Linguagens livres	Boo • CoffeeScript • D • Dart • Erlang • Haskell • Java • JavaScript • Perl • PHP • Python • Ruby • Rust • Scala

Em algoritmos, as variáveis devem ser definidas no início, na área reservada para essa finalidade, sempre através da instrução relativa a criação. Em seguida, veja duas formas de declaração de variáveis em algoritmos do tipo pseudocódigo:

VAR <nome_da_variável> : <tipo_da_variável>

Ou

VAR <lista_de_variáveis> : <tipo_das_variáveis>

- a instrução VAR é a responsável pela criação de variáveis e deve ser utilizada uma vez para definição de uma ou de um conjunto de variáveis;
- podemos definir variáveis de mesmo tipo na mesma linha separadas por vírgula;
- quando as variáveis são de tipos diferentes devem ser declaradas em linhas diferentes.

Exemplo para definição de variáveis:

```
VAR nome_funcionario: caractere[15]
    idade_funcionario: inteiro
    salario_funcionario: real
    dependentes: lógico
```

No exemplo, declaramos quatro variáveis:

- **nome_funcionario**, que contém dados do tipo caractere com tamanho máximo de 15;
- **idade_funcionario**, que contém dados do tipo inteiro;
- **salario_funcionario**, que contém dados do tipo real;
- **dependentes**, que contém dados do tipo lógico.

Lembrete

Você somente conseguirá atribuir um valor a uma variável se ela estiver criada na seção de declaração.

Saiba mais

Assista ao filme:

UMA MENTE brilhante. Direção: Ron Howard. EUA: Universal Pictures, 2001. 135 minutos.

Nesse filme, a utilização de algoritmos para diversas situações é realizada pelo ator principal, professor John Nash. O filme ajuda a compreender o relacionamento existente de situações do nosso dia a dia com as teorias de algoritmos.

1.6 Conceito e utilidade de constantes

Podemos considerar como constante algo que não sofre alterações, sendo estável ou fixo. Esse é um conceito muito utilizado na área de programação pelos recursos por ela proporcionados, que abordaremos a seguir.

Normalmente, devemos declarar as constantes que serão utilizadas no algoritmo, bem como nos programas de computador, no início, junto à área de definição de variáveis. Isso permite facilitar o entendimento da função das constantes e, principalmente, para facilitar a alteração dos seus valores, quando necessário. Alterando, desse modo, esse valor na área de definição, todas as instruções que fazem uso dessa constante já assumem seu novo valor, assim que o programa entrar em execução novamente.

1.6.1 Definição de constantes em algoritmos

Definimos as constantes através da utilização da instrução demonstrada a seguir, respeitando a sintaxe:

CONST <nome_da_constante> = <valor>

Exemplos:

CONST pi = 3.14159
nome_da_empresa = "Doces Maria"

Nos exemplos anteriores, podemos verificar a criação de duas constantes, uma com valor numérico real e outra do tipo string ou conjunto de caracteres, neste caso o conteúdo obrigatoriamente deve ficar entre aspas.

1.7 Operadores

Podemos definir operadores como componentes de uma expressão matemática que atuam sobre dois operandos, gerando um determinado resultado. Exemplo: 100 – 20 (operador de subtração atua sobre os dois operandos).

Os operadores se classificam em duas categorias, de acordo com o número de operandos sobre os quais eles atuam:

- **Tipo binário**: operadores agem sobre dois operandos. Podemos citar, entre eles, os aritméticos, tais como soma e subtração.

- **Tipo unário**: operadores que atuam sobre um único operando. O operador mais comum deste tipo é o operador de negação (-), que tem a função de inversão do sinal de um operando.

Também existe uma classificação de operadores que considera o tipo de dado dos operandos e o tipo do valor obtido como resultado. Nessa classificação, os operandos se enquadram em aritméticos, lógicos e literais. Essa classificação tem relação com o tipo da expressão nos quais os operadores aparecem.

Operadores de atribuição

Definimos operador de atribuição ao que tem a função de atribuir um determinado valor a uma variável.

A sua representação é feita pelo símbolo: **:=**

Representamos a sintaxe desse comando como sendo:

Nome_da_Variavel := expressão

De acordo com a sintaxe, o operador atribui ao componente da esquerda – variável – o resultado da expressão à direita. Essa atribuição só é concluída após a validação da expressão e a obtenção de um resultado por ela.

Operadores aritméticos

Definimos como operadores aritméticos aqueles que representam as operações aritméticas básicas, assim como na matemática tradicional. O quadro a seguir relaciona os operadores aritméticos utilizados em algoritmos.

Quadro 3 – Operadores aritméticos

Operador	Tipo	Operação	Prioridade
+	Binário	Adição	4
-	Binário	Subtração	4
*	Binário	Multiplicação	3
/	Binário	Divisão	3
MOD	Binário	Resto da divisão	3
DIV	Binário	Divisão inteira	3
**	Binário	Exponenciação	2

A coluna prioridade define a ordem em que eles são executados em uma expressão.

Operadores relacionais

Definimos como operadores relacionais aqueles que retornam como resultado da operação, sendo valores lógicos verdadeiros ou falsos. Esses operadores são classificados como binários. O quadro a seguir relaciona os operadores lógicos utilizados em algoritmos.

Quadro 4 – Operadores relacionais

Operador	Comparação
>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual
=	Igual
<>	Diferente

Essas comparações lógicas só podem ser usadas entre variáveis de mesmo tipo, e seu resultado sempre serão valores lógicos.

Tendo uma variável do tipo inteira denominada valor, contendo o valor 10, podemos verificar que a primeira das expressões a seguir retornará como resultado da comparação lógica um valor falso, e a segunda um valor verdadeiro:

valor <= 5

valor > 8

Quando trabalhamos com variáveis do tipo string, os operadores lógicos fazem uso da tabela ASCII (do inglês American Standard Code for Information Interchange – Código Padrão Americano para o Intercâmbio de Informação), que relaciona um determinado caractere a um valor, para realizar a comparação lógica. Uma variável string terá seu valor menor do que uma outra, se ela tiver os números menores dentro da tabela ASCII.

Exemplo:

"linguagem" > "LINGUAGEM"

"ABC" < "EFG"

"Basic" < "Basic compiler"

Os **códigos ASCII** para letras minúsculas possuem valores maiores que as letras maiúsculas. Podemos observar que o comprimento da string também é fundamental para determinar o valor, mesmo quando temos caracteres iguais. Nesse caso, strings maiores terão valores maiores como resultado.

O ASCII é um código binário (cadeias de bits 0s e 1s) que codifica um conjunto de 128 sinais: 95 sinais gráficos (letras do alfabeto latino, sinais de pontuação e sinais matemáticos) e 33 sinais de controle, utilizando, portanto, apenas sete bits para representar todos os seus símbolos.

Tabela 1 – Exemplo de alguns sinais x código ASC II

Binário	Octal	Decimal	Hexadecimal	Sinal
0110 0001	141	97	61	a
0110 0010	142	98	62	b
0110 0011	143	99	63	c
0110 0100	144	100	64	d
0110 0101	145	101	65	e
0110 0110	146	102	66	f

Operadores lógicos

Definimos como operadores lógicos aqueles que têm a função de combinar expressões do tipo relacionais. Esses operadores também têm como resultado valores lógicos, verdadeiros ou falsos. O quadro a seguir relaciona os operadores lógicos utilizados em algoritmos.

Quadro 5 – Operadores lógicos

Operador	Tipo	Operação	Prioridade
OU	Binário	Disjunção	3
E	Binário	Conjunção	2
NÃO	Unário	Negação	1

Analisemos agora as opções lógicas através de duas expressões, EXP1 e EXP2, e os resultados obtidos:

- EXP1 **E** EXP2 somente será verdadeiro se ambas as expressões retornarem valores verdadeiros. Se qualquer uma delas, ou as duas juntas, retornar valor falso, o resultado lógico também será falso.

1) EXP1 **OU** EXP2 será verdadeiro se qualquer uma das duas expressões, ou as duas, retornar valores verdadeiros. Somente assumirá valor falso caso as duas expressões retornarem valores falsos.

2) **NÃO**, também conhecido como operador de negação, tem a função de inverter o valor lógico obtido da expressão, ou seja, retornará verdadeiro caso o resultado de uma expressão for falso e vice-versa.

Operadores literais

Definimos como operadores literais aqueles que atuam sobre caracteres ou conjunto de caracteres. Existe uma grande variação desse tipo de operador de uma linguagem para outra, mas o operador mais utilizado dentro dessa classificação é o operador de concatenação, que tem a função de juntar duas strings. Veja um exemplo que demonstra essa junção, sempre pegando o início da segunda string, concatenando ao final da primeira.

O símbolo para esse operador é o +.

"LINGUAGEM DE " + "PROGRAMAÇÃO"

Executando a operação anterior, obtemos o seguinte resultado:

"LINGUAGEM DE PROGRAMAÇÃO"

Lembrete

O operador aritmético atua entre dois números, resultando em outro número. Por sua vez, o operador relacional atua entre duas grandezas, resultando num valor lógico. Já o operador lógico atua apenas em valores lógicos, resultando em um valor lógico.

Expressões

Trabalhamos com expressões dentro das linguagens de programação da mesma forma como na matemática, em que variáveis e constantes numéricas se relacionam através de operadores, formando expressões matemáticas que, após a sua execução, resultarão em valores correspondentes ao tipo dos dados que fazem parte da expressão. O que muda nas linguagens de programação em relação à matemática clássica são algumas simbologias para determinados operadores.

Expressões aritméticas

Definimos como expressões aritméticas as que apresentam como resultado valores numéricos inteiros ou reais, fazendo exclusivamente uso de operadores aritméticos, variáveis de tipos numéricos e de parênteses para alteração da sequência de sua execução.

Expressões lógicas

Definimos como expressões lógicas as que apresentam como resultado um determinado valor lógico, sendo eles verdadeiros ou falsos. Nesse tipo de expressão, os operadores lógicos e relacionais são utilizados de forma que podem ser combinados a expressões aritméticas. A utilização de parênteses se faz necessária quando se fizer a comparação de duas ou mais expressões desse tipo, indicando a ordem de execução das comparações.

Expressões literais

Definimos como expressões literais as que apresentam como resultado valores literais, ou seja, caracteres. O operador utilizado nesse tipo de expressão é o operador de concatenação (+).

Avaliação das expressões

Quando expressões possuem somente um operador, elas podem ser consideradas diretamente, porém quando elas se tornam mais elaboradas, com mais de um operando na expressão, há a necessidade de considerar a sua execução por trechos. A ordem de execução dos trechos é determinada de acordo com o formato da expressão e se considerando a prioridade dos operadores que dela fazem parte. Quando se faz uso de parênteses, a prioridade é alterada para que se resolva o que se encontra dentro deles.

Algumas regras são muito importantes para que possamos fazer uma análise correta das expressões:

- A primeira coisa que devemos avaliar é a prioridade dos operadores que fazem parte da expressão, de acordo com os valores apresentados nas tabelas dos operadores. Quanto maior for a prioridade de um operador, significa que ele deve ser executado primeiro. No caso de igualdade de prioridade, deve-se executar a expressão da esquerda para a direita.

- Parênteses utilizados em expressões têm a capacidade de alterar a ordem de prioridade de execução dos operadores. Deve-se sempre executar o que está entre parêntesis antes dos demais.

- Entre os próprios operadores existe um certo grau de prioridade para a análise de uma expressão. Primeiramente, analisam-se os aritméticos e literais, em seguida são analisadas as subexpressões que contenham operadores relacionais e, por último, os operadores lógicos.

1.8 Instruções e comandos

Definimos como instruções primitivas os comandos essenciais para que as tarefas possam ser executadas pelos sistemas computacionais. Podemos citar as atividades de entrada e saída de dados, que correspondem à forma de comunicação com o usuário, e as atividades de manipulação dos dados na memória. Todas as linguagens de programação possuem instruções para essas atividades.

Podemos definir alguns pontos que são fundamentais nos sistemas computacionais:

- Dispositivo de entrada é a forma com que os dados são inseridos pelo usuário ou pelo próprio sistema, fazendo a transferência de dados na memória do computador. Os dados também podem prover de outros dispositivos e até mesmo de outros computadores que possuam meios de comunicação. Os principais componentes que exercem essa função são: teclado, mouse e leitora de código de barras.

- Dispositivo de saída á a forma pela qual os dados ou informações são mostrados ao usuário. Geralmente são os resultados do processamento, executado nos programas de computador, mas

também podem ser a transferência de dados internamente na memória. Os principais componentes que exercem essa função são: o monitor, impressoras e displays de informação.

- Sintaxe nada mais é do que a forma correta que instruções e comandos devem ser descritos, permitindo que os compiladores possam entender o que se deseja e realizar a tradução para linguagem de máquina sem nenhum problema.

- Semântica é o que definimos como a sequência de atividades que serão executadas pelo computador para um determinado comando.

Essas regras são obrigatórias para uma correta execução dos programas de computadores, e são adotadas na elaboração dos algoritmos.

1.8.1 Comandos de atribuição

A função do comando de atribuição é armazenar informação ou dados em uma variável.

A sintaxe utilizada para esse comando é a seguinte:

<nome_da_variável> := <expressão>

Exemplos:

nome := "Maria"

valor := 20.95

quantidade := 10

valor_total : valor * quantidade

tarifas := total * 17 / 100

A forma como esse comando funciona, ou seja, sua semântica, é a seguinte:

- a expressão é avaliada;

- o valor resultante é armazenado na variável que aparece à esquerda do comando.

A seguir, um exemplo de algoritmo em pseudocódigo que faz uso do comando de atribuição.

```
Algoritmo Aplicacao_Comando_de_Atribuicao
Var preco_unitario, preco_final : real
    quantidade : inteiro
Inicio
    preco_unitario := 10.0
```

```
    quantidade := 5.0
    preco_final := preco_unitario * quantidade
Fim
```

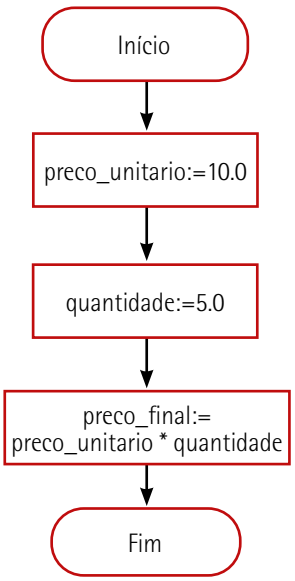


Figura 4 – Representação em fluxograma

Lembrete

A utilização da simbologia := para o operador de atribuição é necessária para diferenciá-lo do operador de igualdade, impedindo, dessa forma, uma interpretação errada do raciocínio lógico que foi elaborado para resolução de um determinado problema.

1.8.2 Comandos de saída de dados

Em algoritmos, fazemos uso de comandos específicos para saída de dados. É através deles que as informações manipuladas pelo computador e disponíveis na memória são transferidas para os dispositivos de saída, sendo apresentadas para os usuários.

Na forma representativa de fluxograma, os comandos de saída são representados por:

Figura 5

Em pseudocódigo podemos utilizar quatro sintaxes para esse comando:

ESCREVA <variavel>
Exemplo: **ESCREVA** W

ESCREVA <lista_de_variaveis>
Exemplo: **ESCREVA** nome, endereço, município

ESCREVA <literal>
Exemplo: **ESCREVA** linguagem de programação

ESCREVA <literal>, <variavel>, ... ,<literal>, <variavel>
Exemplo: **ESCREVA** Meu nome é: , nome, e meu endereço é: , endereço

A palavra **escreva** fica sendo reservada à lista de comandos, não podendo ser utilizada como nome de variável, pois, caso isso aconteça, o algoritmo não tem como identificar um comando de saída de dados.

O funcionamento do comando de saída é relativamente simples: os argumentos ou as variáveis são enviados para o dispositivo de saída. Quando se trata de uma lista de variáveis, o conteúdo de cada uma é buscado na memória e enviado para o dispositivo. Quando estamos lidando com strings, todo o conteúdo é enviado para o dispositivo de uma única vez.

O comando permite ainda a mistura de variáveis com dados literais, trazendo um grande recurso para a montagem de mensagens mais claras para o usuário. Nesse caso, a lista de envio é lida da esquerda para a direita, quando o comando encontra uma variável no meio, ele busca o conteúdo e o integra na mensagem.

Veja um exemplo de algoritmo em pseudocódigo que utiliza o comando de saída de dados.

```
Algoritmo Comando_Saida_de_Dados
Var preco_unitario, preco_final : real
    quantidade : inteiro

Inicio
    preco_unitario := 10.0
    quantidade := 5
    preco_final := preco_unitario * quantidade
    Escreva preco_final
Fim
```

Observação

Algumas literaturas fazem uso das instruções escrever ou imprimir no lugar do comando escreva. Todas as possibilidades estão corretas, sendo que mais importante é o entendimento da utilização independentemente da sintaxe. Essa é a grande vantagem de utilizar o português estruturado: ele não fica limitado a uma regra restrita de linguagem de programação.

1.8.3 Comandos de entrada de dados

Em algoritmos, os comandos de entrada de dados definem a forma como as informações, necessárias ao sistema, são inseridas pelo usuário e transferidas para a memória do computador.

Na representação de fluxograma, o comando de entrada de dados é representado por:

Figura 6 – Comando de entrada de dados

Em pseudocódigo, existem duas sintaxes possíveis para utilização do comando.

LEIA <variavel>
Exemplo: **LEIA** W

LEIA <lista_de_variaveis>
Exemplo: **LEIA** nome, endereço, cidade

Assim como acontece com a palavra **escreva**, a palavra **leia** também passa a ser utilizada somente como comando, não podendo ser utilizada como nome de variável.

O funcionamento do comando é bem semelhante ao anterior, só que no caminho contrário: as informações que são inseridas pelo usuário, através dos dispositivos de entrada, são transferidas para as áreas de memória correspondentes às variáveis definidas no sistema, sendo elas variáveis únicas ou pertencentes a uma lista de variáveis. Veja um algoritmo que faz uso do comando de entrada de dados.

```
Algoritmo Comando_Entrada_de_Dados
Var preco_unitario, preco_final : real
    quantidade : inteiro
```

```
Inicio
    Leia preco_unitario, quantidade
    quantidade := 5
    preco_final := preco_unitario * quantidade
    Escreva preco_final
Fim
```

As boas práticas de programação pregam que sempre devem ser desenvolvidos sistemas que sejam amigáveis com o usuário, ou seja, deve-se sempre elaborar os programas de forma que a interação com o usuário seja o mais simples possível, de modo a não deixar em dúvida o que o usuário deve fazer em determinada situação. Nesse sentido, a elaboração de uma interface amigável é fundamental. Existem algumas orientações que devem ser seguidas para atingirmos esse objetivo:

- sempre que houver a necessidade de uma entrada de dados por parte do usuário, o programa deve emitir uma mensagem informativa solicitando claramente a informação que precisa ser inserida;

- sempre que o sistema for mostrar ao usuário uma saída de informação, ele deve emitir uma mensagem contextualizando a informação, sem deixar dúvidas sobre o que está aparecendo para ele naquele instante.

Seguindo essas orientações, tem-se um sistema amigável e de fácil utilização pelo usuário. Veja um exemplo de algoritmo em pseudocódigo que segue essas orientações.

```
Algoritmo Interface_Amigavel
Var preco_unitario, preco_final : real
    quantidade : inteiro
Inicio
    Escreva "Digite o preço unitário do produto: "
    Leia preco_unitario
    Escreva "Digite a quantidade de produtos: "
    Leia quantidade
    preco_final := preco_unitario * quantidade
    Escreva "O preco final é: " , preco_final
Fim
```

2 FUNÇÕES MATEMÁTICAS

Em programação, existem funções que executam atividades específicas, com o objetivo de facilitar o desenvolvimento de sistemas. Podemos considerar uma função como um trecho de código desenvolvido por alguém, que executa uma atividade específica, comum a vários programas. Nesse caso, essas funções são encapsuladas em um bloco de programa que pode ser chamado, em qualquer ponto de um programa, pelo nome recebido. Qualquer programador pode fazer uso dessas funções, passando para elas os valores necessários para executar suas atividades e devolver o resultado da função.

Pense em uma calculadora, com diversas teclas que executam sequências matemáticas; sem que o usuário saiba como o cálculo é feito, um resultado retorna. Para calcularmos a raiz quadrada de um número, selecionamo-lo e apertamos a tecla de raiz quadrada, assim, a calculadora retorna o resultado sem precisarmos saber como o mesmo foi feito.

Uma classe de funções muito utilizadas e disponibilizadas por todas as linguagens de programação é de funções matemáticas. Em algoritmos também existem funções prontas para cálculos matemáticos. Seguem alguns deles:

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

2.1 Estruturas de controle do fluxo de execução

Em programação, existem situações em que a sequência de execução das instruções ocorre uma após a outra, é o que chamamos de programas sequenciais. Neles, elas são sempre executadas na mesma sequência. Esse tipo de programação é o que foi apresentado nos exemplos anteriores.

Na vida real, nem sempre conseguimos elaborar os programas dessa forma. Muitas vezes vão ocorrer situações em que a sequência de execução das instruções necessita mudar, ou seja, diante de certas circunstâncias, o programa deve executar uma ou outra sequência de instruções a fim de atingir os objetivos. Também há situações em que o programa precisa executar uma determinada sequência de instruções mais de uma vez, de forma determinada.

Nessas situações, dizemos que existe a necessidade de se controlar o fluxo de execução de um programa, e esse controle é implementado com funções específicas, que dependem da lógica implementada para tratamento dos dados.

Em algoritmos, há três tipos de estruturas para controle do fluxo de execução das instruções, são elas:

- estruturas sequenciais;
- estruturas de decisão;
- estruturas de repetição.

Para compreendermos melhor as estruturas, é importante entender o conceito de **comando composto**. Podemos definir um comando composto como sendo um conjunto de instruções ou comandos simples agrupados em um trecho de código, visando atingir um determinado objetivo. Como instruções simples podemos citar comandos de atribuição, comandos de entrada e saída.

2.1.1 Estrutura sequencial

Definimos como uma estrutura sequencial aquela na qual as instruções de um algoritmo são executadas uma de cada vez, sempre em sequência, ou seja, uma instrução só é executada após o término completo da instrução anterior, e cada uma é executada somente uma vez. Em algoritmo, as estruturas sequenciais são delimitadas pelas instruções de **início** e **fim**.

2.1.2 Estruturas de decisão

Definimos, como estrutura de decisão, aquela que determina a sequência de execução das instruções em um algoritmo de acordo com o resultado de uma operação de uma ou mais comparações. Os resultados dessas comparações são respostas lógicas, por se tratar de uma expressão lógica de comparação.

As estruturas de decisão se classificam de acordo com o número de comparações a serem executadas, definindo, dessa forma, o caminho do fluxo de execução a ser seguido. Apresentamos os três tipos possíveis.

- Estrutura de decisão simples (**se ... então**).
- Estrutura de decisão composta (**se ... então ... senão**).
- Estrutura de decisão de múltipla escolha (**escolha ... caso ... senão**).

As estruturas sequenciais conseguem resolver grande parte das necessidades dos programadores, porém há situações em que se depende da análise de comparação dos dados ao longo do código para chegar ao resultado esperado, fazendo com que o algoritmo tenha que executar uma sequência de comandos diferente, mediante os resultados obtidos nas comparações. Para essas situações, aplicam-se as estruturas de decisão.

Estruturas de decisão simples (se ... então)

Em uma estrutura de decisão simples, é executada uma única comparação para sua validação lógica e, de acordo com o resultado, que pode ser verdadeiro ou falso, uma determinada sequência de instruções será executada. Veja, na figura a seguir, uma representação de blocos para esta estrutura.

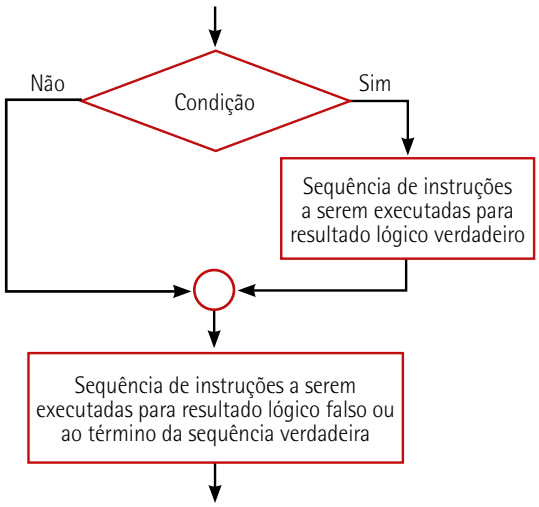


Figura 7

Para os algoritmos em pseudocódigo, há duas formas de representação desse tipo de estrutura. A primeira é:

SE <condição> ENTAO <comando_único>

Exemplo 1

SE **X** > 100 ENTAO Escreva "A variável X é maior que 100"

Já a segunda opção é:

```
SE <condição> ENTAO
INICIO
    <comando_composto>
FIM
```

Exemplo 2

```
SE X > 100 ENTAO
INICIO
    contador := contador + 1
    soma_numeros := soma_numeros + X
    Escreva "X é maior que 100"
FIM
```

A sequência anterior demonstra que, se a condição de comparação lógica tiver um resultado verdadeiro, então um comando simples ou um conjunto de comandos é executado. No caso de um comando composto, ele é delimitado pelas palavras **início** e **fim**. É dessa forma que o algoritmo entende onde começa e onde termina a sequência de instruções a serem executadas dentro dessas estruturas. Assim que a estrutura de decisão é finalizada, o algoritmo segue sua sequência normal, indo para a próxima instrução.

Caso o resultado da comparação lógica seja falso, nenhuma instrução de dentro da estrutura é executada, e o algoritmo segue a sequência normal de execução após o término da estrutura de decisão, delimitada pela palavra **fim**.

Veja um exemplo de algoritmo em pseudocódigo para estrutura de decisão.

```
Algoritmo Estrutura_Decisao_Simples
Var Y : inteiro
Inicio
   Escreva "Digite um valor: "
     Leia Y
     Se Y > 100 Entao
              Escreva "Y é maior que 100"
Fim
```

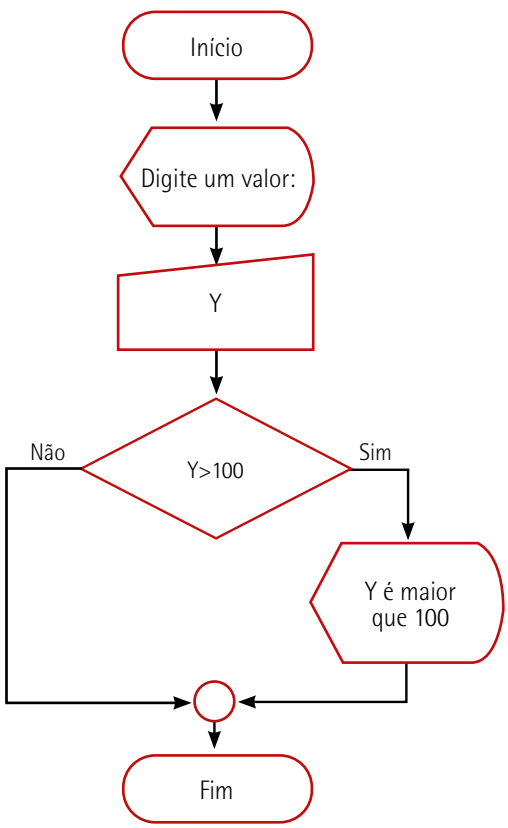


Figura 8

Estruturas de decisão composta (se ... então ... senão)

Em uma estrutura de decisão composta, é executada uma única comparação para sua validação lógica e, se o resultado for verdadeiro, um comando simples ou um comando composto é executado. Se o resultado da comparação lógica for falso, então um outro comando simples ou composto é executado. Veja, na figura a seguir, uma representação de blocos para essa estrutura.

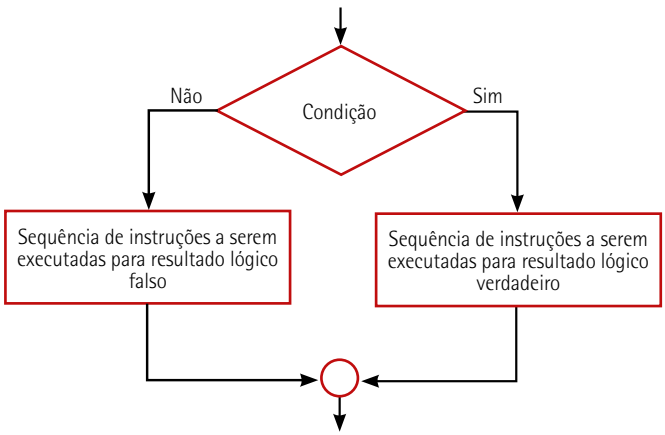


Figura 9

Para os algoritmos em pseudocódigo, há duas formas de representação desse tipo de estrutura. A primeira opção é:

```
SE <condição> ENTAO <comando_único_1>
SENAO <comando_único_2>
```

Exemplo 1

```
SE Y > 1000 ENTAO Escreva "A variável Y é maior que 1000"
SENAO Escreva "A variável Y não é maior que 1000"
```

Já a segunda opção é:

```
SE <condição> ENTAO
   INICIO
     <comando_composto_1>
   FIM
SENAO
   INICIO
     <comando_composto_2>
   FIM
```

Exemplo 2

```
SE Y > 1000 ENTAO
INICIO
   contador_1 := contador_1 + 1
   somatoria_1 := somatoria_1 + Y
   Escreva "Y é maior que 1000"
FIM
 SENAO
INICIO
   contador_2 := contador_2 + 1
   somatoria_2 := somatoria_2 + Y
   Escreva "Y não é maior que 1000"
FIM
```

A sequência anterior demonstra que, se a primeira condição de comparação lógica tiver resultado verdadeiro, então um comando simples ou um conjunto de comandos (**comando_composto_1**) é executado. No caso de comando composto, ele é delimitado pelas palavras **início** e **fim**. Ao término da execução dessas instruções, o algoritmo segue para a próxima instrução, após a estrutura de decisão composta.

Caso o resultado da comparação lógica seja falso, então um comando simples ou um conjunto de comandos (**comando_composto_2**) é executado, referente à condição de exceção (**senão**). No caso de comando composto, ele é delimitado pelas palavras **início** e **fim**. Ao término da execução dessas instruções, o algoritmo segue para a próxima instrução, após a estrutura de decisão composta.

Veja a seguir um exemplo de algoritmo em pseudocódigo para estrutura de decisão.

```
Algoritmo Estrutura_Decisao_Composta
Var Y : inteiro
Inicio
   Leia Y
   Se Y > 100 Entao
            Escreva "Y é maior que 100" Senao
            Escreva "Y não é maior que 100"
Fim
```

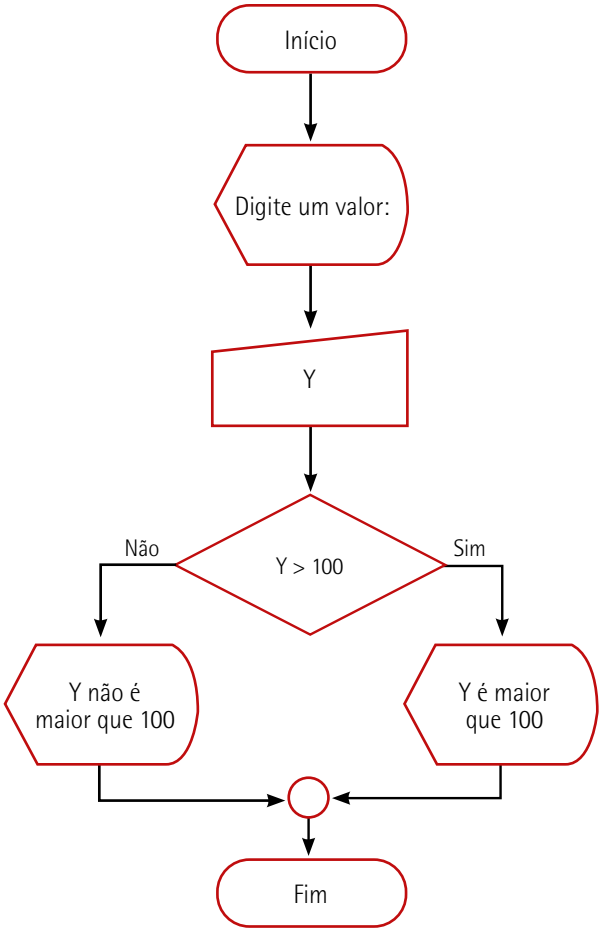


Figura 10

Quando elaboramos as comparações lógicas em algoritmos, devemos tomar o máximo cuidado, pois as decisões sobre o fluxo de execução das instruções dependem dessas comparações. Caso erros aconteçam, os resultados lógicos não ocorreram como planejado, e sequências indesejadas de instruções acontecerão. Quando elaboramos várias comparações lógicas, com certeza termos algoritmos diferentes.

Esse tipo de estrutura permite a concatenação de múltiplas estruturas de decisão, ou seja, podemos utilizar várias estruturas, uma após a outra, e também concatenadas, sendo uma dentro da outra. Como exemplo, podemos mostrar o resultado de um aluno mediante a média obtida das provas. Este algoritmo é um exemplo de estrutura concatenada:

Algoritmo **estruturas_concatenadas**

```
Var NP1, NP2, Media_Final : real
Inicio
Ler NP1, NP2
Media_Final := (NP1+NP2)/2
Se Media_Final = 7 Entao Escreva "Aluno Aprovado"
```

```
Se Media_Final >= 3 E Media_Final < 7 Entao Escreva "Aluno de Exame"
Se Media_Final < 3 Entao Escreva "Aluno Reprovado"
Fim
```

O algoritmo em pseudocódigo a seguir faz uso de **estruturas aninhadas ou encadeadas**. Algoritmo **estruturas_aninhadas**:

```
Var NP1, NP2, Media : real
Inicio
Ler NP1, NP2
Media_Final := (NP1+NP2)/2
Se Media_Final = 7 Entao
   Escreva "Aluno Aprovado"
Senao Se Media_Final >= 3 E Media_Final < 7 Entao
   Escreva "Aluno de Exame"
Senao
   Escreva "Aluno Reprovado"
Fim
```

Podemos observar que a utilização de estruturas concatenadas propicia algoritmos mais simples, facilitando o seu entendimento. Já as estruturas aninhadas ou encadeadas propiciam que a execução dos algoritmos seja mais rápida, pois assim que se encontra uma condição verdadeira, a sequência de instruções é executada, e nenhuma nova comparação é realizada, fazendo com que o algoritmo já passe para o próximo ponto, após a estrutura de decisão; menos pontos são executados.

As estruturas aninhadas ou concatenadas são utilizadas somente quando o seu uso é indispensável, sendo a concatenada a mais utilizada, pela facilidade no entendimento.

Lembrete

A utilização de estruturas de decisão concatenadas facilita a aplicação dos algoritmos, mas não impede que o desenvolvedor utilize somente estruturas simples para as devidas comparações. Na maioria das situações, a utilização de estruturas concatenadas diminui o número de comparações a serem executadas no programa de computador que será gerado.

Estruturas de decisão múltipla caso (caso ... fim_caso ... senão)

Em uma estrutura de decisão de múltipla escolha, podemos ter uma ou mais comparações lógicas a serem testadas e, consequentemente, uma instrução ou conjunto de instruções diferente para cada uma das condições. Esse tipo de estrutura é uma implementação mais simples de comparações lógicas encadeadas. Sua principal vantagem é deixar o código mais simples quando há muitas comparações a serem testadas. Veja, em seguida, uma representação de blocos para essa estrutura.

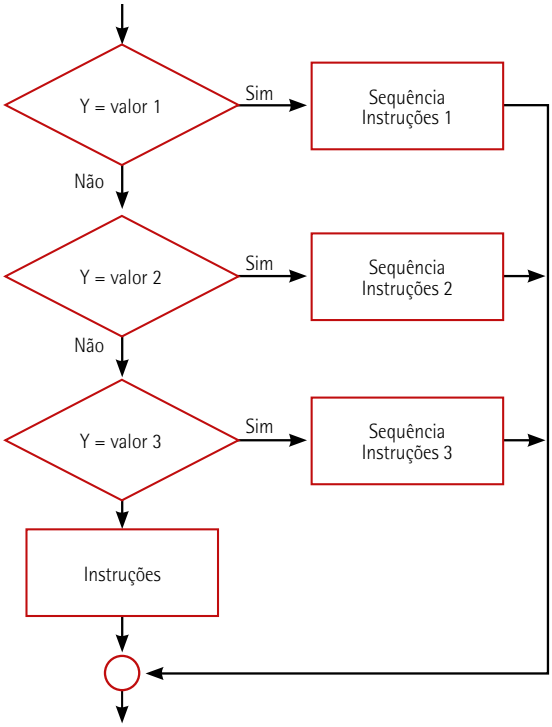


Figura 11

Em pseudocódigo, a estrutura de múltipla decisão é:

```
INICIO
CASO <variavel>
   SEJA <condicao_1> FACA
   <comando_composto_1>
   SEJA <condicao_2> FACA
   <comando_composto_2>
...
   SEJA <condicao_n> FACA
   <comando_composto_n>
   SENAO <comando_composto_s>
FIM
```

Podemos observar que, quando a sequência de execução chega em uma estrutura de múltipla decisão, a primeira comparação lógica é avaliada de acordo com o valor da variável. Se o resultado for verdadeiro, a sequência de instruções é executada e o fluxo de execução segue para a primeira instrução após o término da estrutura de decisão. Caso a primeira comparação lógica seja falsa, a segunda comparação lógica é avaliada: se o resultado for verdadeiro, a sequência de instruções é executada e o fluxo sai da estrutura, indo para a instrução. Assim ocorre, sucessivamente, pelas demais comparações lógicas. Se todas as comparações lógicas apresentarem resultado falso, então o fluxo entra na condição **senão**, executando a sequência de instruções referente a essa situação.

A situação da condição **senão**, na realidade, não avalia nenhuma comparação lógica para a variável, como nas demais; ela entra em uma condição na qual nenhuma das anteriores foi satisfeita. Consideramos

essa uma condição de exceção às demais. Essa cláusula não é obrigatória, porém muito utilizada em situações de informação ao usuário em que todas as demais condições não foram satisfeitas.

Podemos citar um exemplo de aplicação dessa estrutura para o caso de reajuste dos salários de aposentados do INSS, de acordo com a categoria. Reajuste de 20% para aposentados do funcionalismo público, 15% para aposentados do comércio e 10% para os demais aposentados. Na representação de diagrama de blocos, ou fluxograma, o algoritmo fica da seguinte maneira:

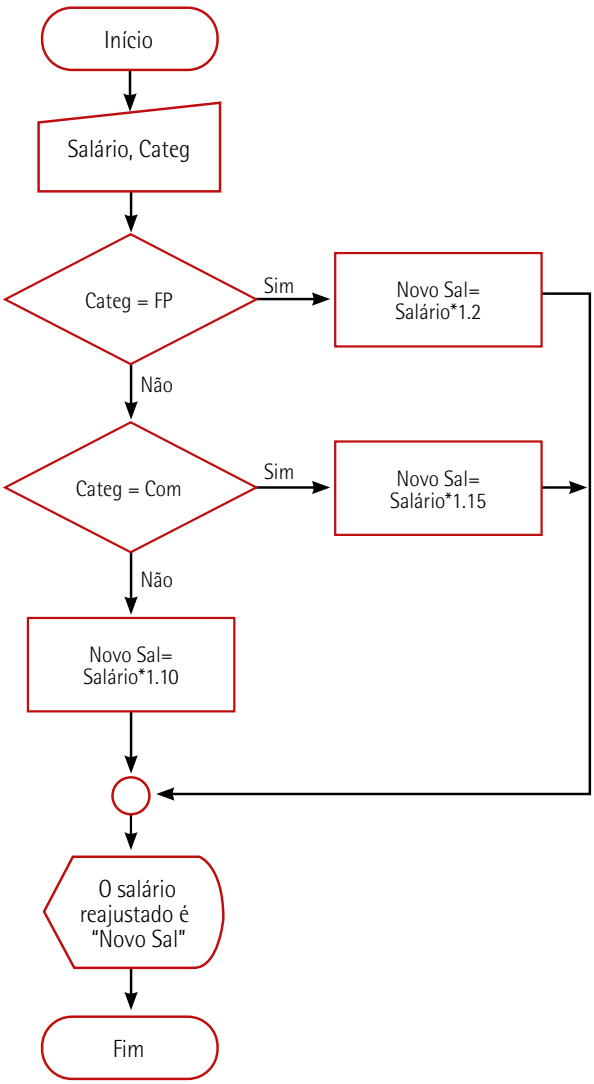


Figura 12

A sintaxe da construção de caso é:

```
Algoritmo Salario
Var Salario, Novo_Sal : real
    Categ: caractere [20]
Inicio
    Leia Salario, Categ
    Caso Categ
```

```
            Seja "FP" faca
                  Novo_Sal := 1.2 * Salario
            Seja "Com" faca
                  Novo_Sal := 1.15 * Salario
            Senão
                  Novo_Sal := 1.10 * Salario
            Fim
            Escreva "O salário reajustado é ", Novo_Sal
Fim
```

2.1.3 Estruturas de repetição

Outro tipo de estrutura que permite a alteração do fluxo de execução das instruções em algoritmos são as chamadas estruturas de repetição. Esse tipo de estrutura é utilizado quando se necessita repetir um certo trecho de instruções por um número determinado de vezes. Há diversas aplicações para esse tipo de estrutura no nosso dia a dia, e o tipo de estrutura a ser utilizada depende da quantidade de vezes que se deseja que tal sequência seja executada. Esse tipo de estrutura possui algumas denominações, sendo as mais comuns laços ou loops.

As estruturas de repetição se classificam em dois grandes grupos, que dependem de o programador conhecer previamente quantas vezes a sequência de instruções deve ser executada. De acordo com essa definição, os grupos podem ser:

- **Laço contado**: esse tipo de estrutura de repetição é utilizado quando o programador conhece exatamente o número de vezes que a sequência de instruções será executada.
- **Laço condicional**: esse tipo de estrutura de repetição é utilizado quando o programador não conhece antecipadamente o número de vezes que a sequência de instruções deve ser executada. Nesse caso, a repetição fica condicionada a uma determinada condição lógica que, em caso de resposta verdadeira, dá permissão para a repetição da sequência de instruções.

Quando fazemos uso de estruturas de repetição em algoritmos, torna-se necessário a utilização de variáveis específicas para o controle do número de repetições – denominadas contadoras e acumuladoras.

Quando utilizamos a variável contadora, geralmente a iniciamos com valor zero antes do início da estrutura de repetição, que deve ser obrigatoriamente incrementada de um determinado valor – geralmente de um – no interior da estrutura de repetição. Veja um exemplo de estrutura de aplicação desse tipo de variável.

```
...
contador := 0
<estrutura_de_repeticao>
...
contador := contador + 1
...
<fim_da_estrutura_de_repeticao>
...
```

Quando fazemos uso de uma variável acumuladora, geralmente atribuímos um valor a ela, sendo zero o mais comum, antes que a estrutura de repetição seja iniciada. Essa variável deve ser incrementada também no interior da estrutura de repetição, porém com valor variável, não um valor fixo. Na maior parte dos casos, utilizamos, como incremento, a própria variável utilizada na estrutura de controle da repetição. Veja um exemplo de estrutura de aplicação desse tipo de variável.

```
...
somatoria := 0
<estrutura_de_repeticao_com_variavel_y>
...
somatoria := somatoria + y
...
<fim_da_estrutura_de_repeticao>
...
```

Laços condicionais

Laço baseado em condições é chamado de laço condicional. As comparações lógicas de determinam as condições podem ficar no início ou no final das estruturas, e as instruções ficam no seu interior, delimitada pelos campos **início** e **fim**.

Um tipo de estrutura condicional é a **Enquanto ... Faça**. Veja uma representação em blocos:

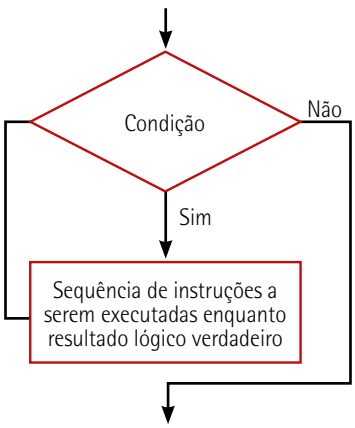


Figura 13

A forma de representação em pseudocódigo é:

```
ENQUANTO <condição> FACA
   INICIO
     <comando_composto>
   FIM
```

O funcionamento dessa estrutura se baseia no teste lógico, realizado no início da instrução **enquanto**. Para um resultado falso, as instruções localizadas no interior da estrutura não são realizadas.

Se o resultado for verdadeiro, então elas são executadas uma vez, e o processamento retorna para realizar um novo teste lógico para a comparação lógica; se a resposta continuar verdadeira, as instruções

continuam sendo executadas. A partir do momento que a resposta lógica se torna falsa, então as instruções deixam de ser executadas. O processamento segue adiante, a partir da próxima instrução do final da estrutura de repetição. A delimitação da estrutura é feita pelos indicadores de **início** e **fim**.

Podemos observar que a execução do código fica dentro da estrutura enquanto a comparação lógica retornar verdadeiro e não sai dessa situação enquanto um resultado falso não ocorrer. Isso faz com que o programador precise tomar um cuidado maior ao criar uma condição, para que, em algum momento, o processamento possa sair da estrutura. Caso isso não ocorra, ele ficará dentro da estrutura indefinidamente, ocasionando uma condição conhecida como loop infinito. Uma forma de contornar essa situação é através de uma variável, que deve ser utilizada na comparação lógica, sendo manipulada dentro da estrutura de repetição.

Apresentamos, a seguir, um exemplo de estrutura que executa uma comparação lógica em que são mostrados números iniciados em 1, enquanto sua soma não ultrapasse 500. No diagrama de blocos, a estrutura para instrução **enquanto** é representada por:

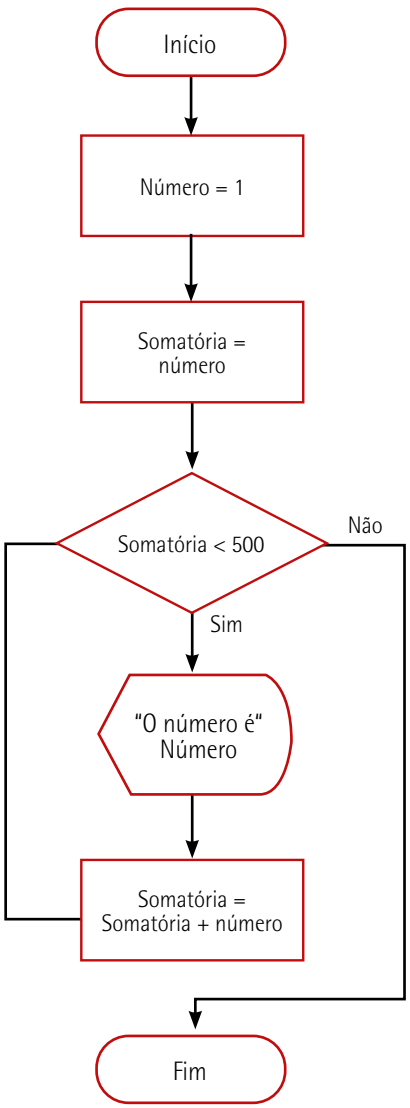


Figura 14

A representação em pseudocódigo é:

```
Algoritmo enquanto
Var somatoria, numero : inteiro
Inicio
numero := 1
soma := numero
   enquanto somatoria < 500 faca
   Inicio
     escreva numero
     numero := numero +1
     somatoria := somatoria + numero
   Fim
Fim
```

Observamos que a estrutura de repetição realiza a comparação lógica com a variável somatória: enquanto essa comparação estiver retornando verdadeira, as instruções delimitadas pelos indicadores **início** e **fim** serão executadas. A partir do momento que a resposta deixar de ser verdadeira, essa execução termina indo em frente no algoritmo. A variável somatória é manipulada dentro do laço e serve para impedir que se crie a condição de execução infinita.

Laços condicional com teste no final (repita ... até que)

Um outro tipo de estrutura de repetição é a que apresenta a comparação lógica no final. A execução das instruções no seu interior fica condicionada também a uma resposta lógica dessa comparação. A representação em pseudocódigo é:

```
REPITA <comando_composto>
ATÉ QUE <condição>
```

Veja a seguir a representação em diagrama de blocos.

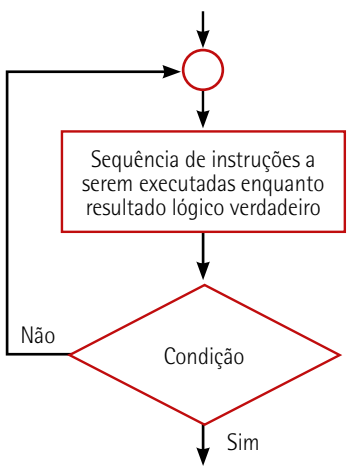


Figura 15

A grande diferença dessa estrutura de repetição em relação à do **enquanto** é que, nesta, a execução da sequência de instruções ocorre pelo menos uma vez. Ao final, a comparação lógica é testada e, se o resultado for falso, a sequência volta a ser executada. A partir do momento em que a resposta passa a ser verdadeira, a execução das instruções dentro da estrutura é interrompida, dando sequência ao restante do algoritmo. Na estrutura de repetição **enquanto**, podemos ter situações em que a sequência de comandos não seja executada nenhuma vez; nesta estrutura, há a garantia de execução ao menos uma vez. Também temos uma grande diferença em relação à manipulação da variável utilizada no laço, que pode ser iniciada dentro do próprio laço.

Veja, a seguir, um exemplo em que números inteiros são digitados até que a soma dos que já foram digitados ultrapasse o valor de 500.

```
Algoritmo repita
Var numero, somatoria : inteiro
Inicio
Repita
     Ler numero
   Escrever "O número digitado foi ", num
   somatoria=somatoria+numero
   até que somatoria > 500
Fim
```

Laços contados (para ... faça)

Uma outra opção de estruturas de repetição é a conhecida como laço contado, bastante utilizada quando conhecemos de antemão o número exato de vezes que as sequências de instruções devem ser executadas. Por esse motivo, são tratadas como estruturas contadoras de execução do laço. Esse tipo de estrutura consegue resolver grande parte dos problemas que necessitam de repetições, mas nada impede a utilização das anteriores para uma mesma situação. Ela é considerada um tipo de estrutura mais simples de ser implementada e entendida. Veja sua representação em diagrama de blocos.

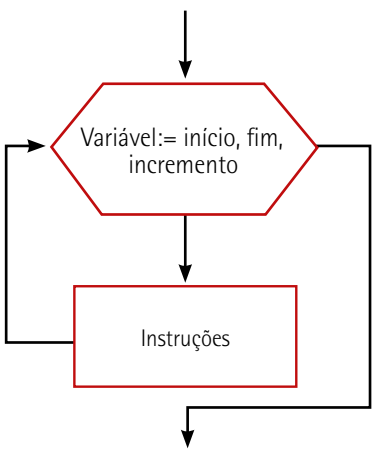


Figura 16

Em pseudocódigo, há duas opções para representação desse tipo de estrutura de repetição. A primeira se dá da seguinte maneira:

```
PARA <variavel> := <inicio> ATE <final> FACA
    <comando_unico>
```

Exemplo 1

```
PARA i := 1 até 100 FACA
ESCREVER i, " + 10 =", i + 10
```

Já a segunda opção é:

```
PARA <variavel> := <inicio> ATE <final> FACA
  INICIO
  <comando_composto>
  FIM
```

Exemplo 2

```
somatoria := 0
temp := 0
PARA i := 1 ATE 100 FACA
  temp := i + 10
  somatoria := somatoria + temp
  ESCREVER "Somatória Total = ", somatoria
FIM
```

A utilização da estrutura de repetição de laço contado inicia uma variável, normalmente em valor zero, e realiza uma comparação com o valor final, que deve ser assumido pelo laço. Se o valor da variável for maior que o parâmetro final, então a sequência de instruções não é executada, levando o algoritmo à sua execução normal após a estrutura. Se o valor da variável for menor ou igual ao parâmetro final, então é habilitada a execução da sequência de instruções e, no seu fim, o valor da variável é incrementado automaticamente, partindo, então, para uma nova comparação.

O processo é repetido até o momento em que a variável assuma um valor maior ou igual ao do parâmetro final de comparação, encerrando a execução do laço.

Uma condição padrão dessa estrutura é o incremento da variável de comparação, que ocorre de um em um, porém podemos determinar sua forma, podendo até termos decrementos dessa variável. A alteração desse parâmetro de incremento é feita através da instrução **passo** seguido do valor numérico que determina o incremento. É importante lembrar que, quando fazemos uso de decremento de passo, temos que prestar a atenção no valor de inicialização da variável, que deve ser compatível, para que seja possível a execução do laço, sem criar a condição de loop infinito. Em pseudocódigo, a representação com passo decrescente de um é feita da seguinte maneira:

```
PARA <variavel> := <inicio> ATE <final> PASSO -1 FACA
  INICIO
    <comando_composto>
  FIM
```

Veja, a seguir, um algoritmo que mostra na tela a sequência de números de 1 até 100:

```
Algoritmo numeros ate 100
Var i : inteiro
Inicio
  Escrever "Números de 1 até 100 "
  Para i := 1 ate 100 faca
  Inicio
    Escrever "Este é o número ", i
  Fim
Fim
```

2.2 Estruturas de controle encadeadas ou aninhadas

Em algoritmos, assim como na programação, podemos utilizar um recurso muito importante para minimizarmos ao máximo a geração de código e para uma melhor organização do algoritmo.

Esse recurso é o alinhamento ou encadeamento de estruturas, no qual podemos utilizar qualquer uma das estruturas utilizadas – que façam parte do comando composto dentro de outra estrutura. Não existe limite para esse recurso em relação à quantidade e tipo de estruturas que se deseja utilizar.

Quando utilizamos esse recurso, é necessário que uma estrutura esteja totalmente encaixada dentro daquela da qual faz parte, ou seja, da estrutura externa. A figura a seguir mostra uma representação de alinhamento de estruturas.

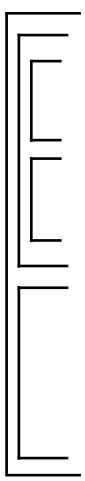


Figura 17 – Exemplos de aninhamentos

2.2.1 Estruturas de dados homogêneas

Assim como na matemática clássica, as linguagens de programação e os algoritmos fazem uso de um conceito muito importante para manipulação de grandes quantidades de dados, desde que sejam todos do mesmo tipo. Esse tipo é denominado estruturas de dados homogêneos, e, em diversas literaturas, encontramos nomes diferentes para elas, tais como: matrizes (o mais comum), vetores, arrays, variáveis indexadas, dentre outras. Em programação, a utilização desse tipo de estrutura se dá principalmente para criação de tabelas de dados. Veremos que, com esse tipo de estrutura, é possível manipular muitas informações, utilizando somente uma variável indexada. As dimensões de uma determinada matriz são definidas por números inteiros e positivos.

2.2.2 Matrizes de uma dimensão ou vetores

O primeiro tipo de estruturas de dados homogêneas que será analisada é denominada matriz de uma dimensão, ou também conhecidas como vetor. A sua definição em um algoritmo é dada pela definição de uma única variável, com um tamanho determinado.

Em algoritmos, uma variável do tipo matriz deve possuir um nome, assim como as demais variáveis, e os parâmetros que a dimensionam em relação à quantidade de colunas e de linhas. Em pseudocódigo, utilizamos a seguinte forma para definição de matrizes de uma dimensão:

```
VAR
<nome_da_variavel> : MATRIZ [ <coluna_inicial>...<coluna_final> ] DE <tipo_de_dado>
```

Exemplo:

```
VAR X :MATRIZ [1 .. 10] DE INTEIRO
```

O exemplo anterior mostra a definição de uma matriz de uma dimensão que possui 10 elementos do tipo inteiro.

2.3 Operações básicas com matrizes do tipo vetor

Podemos executar operações matemáticas com variáveis do tipo matriz da mesma forma como realizamos com variáveis comuns. A diferença é que não podemos executá-las diretamente com o conjunto inteiro de dados, mas com cada elemento de forma individual.

A forma de acessar os elementos individuais de uma matriz é indicando a posição que cada um ocupa nela, através do índice correspondente. É necessário fazer a chamada da variável pelo nome mais o índice do componente desejado. Para acessarmos o primeiro elemento de uma matriz, indicamos da seguinte forma: X[1]; para acessar o décimo elemento, indicamos da seguinte forma: X[10].

As regras e orientações para operações com variáveis aplicadas em algoritmos, até o momento, são aplicadas da mesma forma para matrizes e vetores, sempre de elemento com elemento.

2.3.1 Atribuição de uma matriz do tipo vetor

Para aplicação do operador de atribuição, seguimos a mesma forma utilizada para variáveis comuns.

<nome_da_variável> := <expressão>

Como já explicado, para aplicação desse operador, temos que indicar o elemento desejado da matriz, citando o nome e, entre colchetes, o número correspondente ao índice da posição do elemento. Por esse motivo é que costumamos denominar esse tipo como variável indexada.

No caso de vetores (variáveis indexadas), além do nome da variável, deve-se, necessariamente, fornecer também o índice do componente do vetor em que será armazenado o resultado da avaliação da expressão.

Exemplos:

X[1] := 100

X[4] := 1

X[8] := 1010

X[9] := 22

Saiba mais

Para um melhor entendimento dos conceitos e aplicações de matrizes, leia:

REYNOLDS, J. J.; HARSHBARGER, R. J. *Matemática aplicada*. São Paulo: McGraw Hill, 2006.

MAIO, W. *Fundamentos de matemática*: espaços vetoriais, aplicações lineares e bilineares. São Paulo: LTC, 2007.

Leitura de dados de uma matriz do tipo vetor

A operação de leitura de dados do tipo matriz ou vetor é realizada da mesma forma que para variáveis comuns, considerando apenas que devemos indicar o índice correspondente à posição do elemento na matriz.

Como já explicado, as matrizes ou vetores manipulam grandes quantidades de dados e, para que possamos realizar sua leitura de forma mais simplificada, utilizamos, principalmente, as estruturas de

repetição; entre elas, a do laço **para**. Devemos fazer uso dos laços quando desejamos manipular todos os dados de uma matriz ou vetor; para pequenas quantidades de dados ou dados individualizados, podemos acessá-los individualmente, apesar de não ser muito comum a manipulação individual dos elementos de uma matriz. Veja um exemplo de algoritmo em pseudocódigo que faz uso de um vetor:

```
Algoritmo leitura_vetor
Var numeros : matriz[1, ,10] de inteiro
    i : inteiro
Inicio
  Para i = 1 até 10 faça
  Inicio
  Leia numeros[i]
  Fim
Fim
```

Escrita de dados de uma matriz do tipo vetor

A operação de escrita em matrizes segue as mesmas regras e orientações aplicadas às variáveis comuns, também havendo a necessidade de citar o nome e o índice do elemento que se deseja manipular. Em pseudocódigo, a forma de representação da instrução de escrita de dados se dá da seguinte maneira.

```
ESCREVA <nome_da_variavel> [ <indice> ]
```

Da mesma forma que na leitura de dados, a saída de dados de um vetor também pode ser realizada através da estrutura de repetição **para**. Com ela, o código do algoritmo fica mais simples de ser elaborado, além de facilitar o entendimento para manipulação de grandes quantidades de dados. Veja um exemplo de aplicação das operações de leitura e escrita de um vetor utilizando a estrutura **para**.

```
Algoritmo exemplo_vetor
Var numeros : matriz [1..10] de inteiro
    i, : inteiro
Inicio
  Para i de 1 ate 10 faca
    Inicio
    Leia numeros [ i ]
    Fim
  Para i de 1 ate 10 faca
    Inicio
    Escreva números [ i ]
    Fim
Fim
```

Podemos verificar que as operações de leitura e escrita são semelhantes, indicando a variável e o índice da posição que é dado pela própria variável do laço **para**.

A seguir, segue um novo exemplo, em que um vetor de cem posições é lido e armazenado em um vetor chamado **valores**. Na mesma ação de leitura, é realizada a totalização dos elementos em uma variável. Posteriormente o vetor é escrito bem com a variável acumuladora.

```
Algoritmo soma_elementos
Var numeros : matriz [1..100] de inteiro
   i, : inteiro
   somatoria : inteiro
Inicio
   somatoria := 0
Para i de 1 até 100 faça
     Inicio
     Leia valores [ i ]
     somatoria := somatoria + valores [ i ]
     Fim
Para i de 1 até 10 faça
     Inicio
     Escreva valores [ i ]
     Fim
Escreva "A soma dos valores é ", somatoria
Fim
```

Exemplos de aplicação de vetores

Existem várias utilidades de vetores na área de programação, principalmente quando precisamos manipular grandes quantidades de dados, sendo eles do mesmo tipo. Manipulação que pode ocorrer com estruturas dentro de estruturas, criando, dessa forma, conjuntos de dados em vários graus, aumentando exponencialmente sua quantidade. Também há aplicações que visam elaborar mecanismos de busca ou de ordenação dos dados dentro dessas estruturas de dados homogêneas.

Uma atividade de pesquisa tem a finalidade de verificar se um determinado valor existe dentro de um vetor, executando uma busca em cada um de seus elementos. A atividade de ordenação tem a finalidade de ordenar os componentes do vetor de acordo com uma determinada regra, como, por exemplo, ordenar um vetor de números inteiros em ordem crescente ou decrescente.

2.4 Matrizes com mais de uma dimensão

Existe um outro tipo de estrutura de dados homogêneos, chamadas matrizes de mais de uma dimensão. A principal utilização desse tipo de estrutura é a criação de tabelas de dados em memória do sistema. Também se caracteriza pela definição de uma variável dimensionada através de números inteiros e positivos, que definem seu tamanho.

Nas matrizes de mais de uma dimensão, há a necessidade de especificar seu tamanho e, nesse caso, não somente as linhas, como nas matrizes de uma dimensão, mas também a quantidade de colunas que

comporão a matriz completa. A quantidade de indexadores que devem ser indicados é o que vai definir a dimensão de uma matriz.

Em pseudocódigo, a forma de representação das matrizes de mais de uma dimensão é mostrada a seguir:

Var
<nome_da_variável> : MATRIZ [<linha_inicial> .. <linha_final> , <coluna_inicial> .. <coluna_final>] DE <tipo_de_dado>

Exemplo:

VAR X : MATRIZ [1 .. 10 , 1 .. 100] DE inteiro

Não existe um limite em relação à quantidade de dimensões que podem ser definidas em uma matriz. Veja alguns exemplos de definições de matrizes com várias dimensões:

VAR

A : MATRIZ [1 .. 6] DE INTEIRO

B : MATRIZ [1 .. 20 , 1 .. 6] DE INTEIRO

C : MATRIZ [1 .. 10, 1 .. 20 , 1 .. 6] DE INTEIRO

D : MATRIZ [1 .. 2 , 1 .. 10, 1 .. 20 , 1 .. 6] DE INTEIRO

E : MATRIZ [1 .. 5 , 1 .. 2 , 1 .. 10, 1 .. 20 , 1 .. 6] DE INTEIRO

Como se pode observar com matrizes de mais de uma dimensão, podemos criar subconjuntos de dados dentro de outros, dentro de outros e assim por diante. Cada dimensão irá representar um agrupamento de dados, que deve apresentar algum grau de relação com outro, para que faça sentido o seu armazenamento, e que a manipulação de todos esses dados seja possível.

Com esse tipo de estruturas, manipulamos grandes quantidades de dados, porém é necessário um cuidado especial para que seja possível acessá-los de forma correta, através da indicação dos seus índices.

2.4.1 Operações básicas com matrizes de duas dimensões

As operações matemáticas com matrizes de mais de uma dimensão respeitam as mesmas regras e orientações aplicadas às matrizes de uma dimensão. Não é possível a manipulação do conjunto como um todo, apenas dos seus elementos, de forma individual.

Para que possamos acessá-los de forma individual, é necessário indicar a sua posição dentro da matriz. Isso é feito através da indicação da linha e da coluna em que o elemento se encontra.

A matriz **X**, apresentada no tópico anterior, é capaz de armazenar 100 números inteiros em cada uma das 10 linhas da matriz. Para que consigamos acessar um elemento dessa matriz, é necessário indicar o nome da matriz seguido de sua linha e coluna.

A primeira posição da matriz é dada pela indicação **X[1,1]**, assim como a posição **X[2,10]** acessa o elemento que se localiza na linha 2 e coluna 10.

2.4.2 Atribuição de uma matriz de duas dimensões

A operação de atribuição funciona da mesma forma que nos vetores, atribuindo um valor ou resultado de uma expressão a uma posição específica dentro da matriz. Para isso, é necessária a indicação da linha e coluna correspondentes à posição exata que se deseja. Vale lembrar que, sempre, o primeiro indicador se referirá à linha e o segundo, à coluna. Por exemplo:

X[1,1] := 100

X[1,8] := 12

X[2,4] := 202

X[8,10] := 64

2.4.3 Leitura de dados de uma matriz de duas dimensões

A operação de leitura de uma matriz de mais de uma dimensão também é feita individualmente ou através de uma operação de leitura de todos os elementos, através da utilização da estrutura de repetição para. Em ambos os casos, é necessária a indicação da posição de cada um dos elementos, um de cada vez, mesmo que estejamos utilizando a estrutura de repetição. Devemos lembrar que, sempre, a manipulação é individual, e nunca em sua totalidade. Em pseudocódigo, a forma que devemos utilizar para leitura dos dados de uma matriz com mais de uma dimensão é:

LEIA **<nome_da_variavel> [<indice_1>, ... , < indice_n>]**

Podemos observar, como já foi dito, que é necessário indicar os índices correspondentes à posição do elemento. Não há limite para a quantidade de índices, porém ela dependerá da dimensão que a matriz possui.

Uma forma simples de executar a leitura de todos os elementos da matriz é através das estruturas aninhadas ou encadeadas do laço contado para, isso deixará o código mais simples de ser desenvolvido e entendido. Através desse método, as leituras ocorrerão de forma sucessiva, com a manipulação individual dos elementos da matriz. Devemos utilizar essa técnica quando queremos manipular os dados da matriz com uma mesma operação de manipulação. Veja um exemplo de aplicação de leitura de uma matriz de mais de uma dimensão:

```
Algoritmo Leitura
Var
```

```
valores : matriz [ 1..10, 1..100] de inteiro
i, j : inteiro
Inicio
Para i de 1 Ate 10 Faca
   Inicio
     Para j de 1 Ate 100 Faca
              Inicio
              Leia valores [ i, j ]
              Fim
     Fim
Fim
```

2.4.4 Escrita de dados de uma matriz de duas dimensões

A operação de escrita de matrizes com mais de uma dimensão segue as mesmas regras e orientações aplicadas às operações de leitura e de leitura de variáveis do tipo vetor. Devemos indicar a posição de linha e coluna de cada um dos elementos da matriz.

Em pseudocódigo, a forma como se deve descrever o comando de escrita de matrizes de mais de uma dimensão é:

ESCREVA <nome_da_variavel> [<indice_1>, ... , <indice_n>]

Veja, a seguir, um exemplo de aplicação desse tipo de operação com matrizes de mais de uma dimensão. Observamos que a utilização da estrutura de repetição para, de forma alinhada, diminui em muito a quantidade de código a ser elaborada.

```
Algoritmo Escrita_Matriz
Var
valores : matriz [ 1..10, 1..100] de inteiro
i, j : inteiro
Inicio
   Para i de 1 Ate 10 Faca
     Inicio
              Para i de 1 Ate 100 Faca
              Leia valores [ i, j ]
              Fim
   Fim
   Para i de 1 Ate 10 Faca
     Inicio
              Para j de 1 Ate 100 Faca
              Leia valores [ i, j ]
              Fim
     Fim
Fim
```

Resumo

Nesta unidade, vimos os principais conceitos relativos a algoritmos, que podemos resumir como um conjunto de atividades que devem ser executadas com o intuito de atingirmos um determinado objetivo.

Verificamos os principais tipos de algoritmos utilizados no âmbito da programação e, em especial, o tipo pseudocódigo, que representa uma arquitetura bem semelhante às linguagens de programação mais utilizadas. Aprendemos sua estrutura, tipos de dados, principais funções e aplicações com estruturas específicas para manipulação dos dados. O conceito de variáveis e constantes também foi apresentado, bem como podemos atribuí-las, de acordo com o seu tipo, dentro de uma estrutura, como também fazermos uso delas ao longo de todo o corpo de um algoritmo.

Exercícios

Questão 1. (Funiversa 2009) Um algoritmo pode ser descrito utilizando-se diversas técnicas. A seguir, apresenta-se um exemplo de algoritmo, com a descrição de suas ações:

```
Algoritmo "Maioridade"
// Lê a idade de uma pessoa e indica se é
maior ou menor de 18 anos
//
// Declaração de variáveis
var
idade: inteiro
inicio do algoritmo
    escreva ("Informe a idade: ")
    leia (idade)
    se (idade < 18) então
            escreva ("Menor de 18 anos!")
    senão
            escreva ("Maior de 18 anos!")
fim se fim do algoritmo
```

Assinale a alternativa que apresenta o nome da técnica utilizada para descrição do algoritmo apresentado.

A) Fluxograma.

B) Diagrama de Chapin.

C) Mapa de Karnaugh.

D) Português estruturado.

E) Programação estruturada.

Resposta correta: alternativa D.

Análise das alternativas

A) Alternativa incorreta.

Justificativa: apresenta o algoritmo sob forma gráfica.

B) Alternativa incorreta.

Justificativa: é um tipo diferente de fluxograma.

C) Alternativa incorreta.

Justificativa: é utilizado para resolução de sistemas booleanos.

D) Alternativa correta.

Justificativa: é utilizado para descrever um algoritmo sem a necessidade de uma linguagem de programação formal. Português estruturado, também conhecido como portugol ou pseudocódigo, é uma forma de representação dos algoritmos. Algumas vantagens são: independência física da solução (lógica, apenas); usa português como base, pode-se definir quais e como os dados vão estar estruturados e passagem quase imediata para uma linguagem real.

E) Alternativa incorreta.

Justificativa: estruturação do programa em funções reutilizáveis.

Questão 2. (Cespe 2010) O termo algoritmo é universalmente usado na ciência da computação na descrição de métodos para solução de problemas, adequados à implementação na forma de programas de computador. A esse respeito, assinale a opção correta.

A) As atuais linguagens orientadas a objeto não são adequadas à implementação de algoritmos desenvolvidos para a programação estruturada, visto que muitos algoritmos estruturados não permitem o encapsulamento.

B) Na construção de tipos de dados estruturados, a declaração de classes não corresponde à definição de um novo tipo de dados estruturado.

C) Em linguagens orientadas a objeto, como Java, a declaração de quaisquer tipos de constantes é implementada por meio de macrossubstituição em linha, durante a pré-compilação, isto é, a substituição de referências a constantes pelo valor declarado.

D) Na linguagem Java, a avaliação de uma expressão que constrói uma nova instância, quando bem-sucedida, retorna uma referência ou ponteiro à área de memória na qual a instância foi alocada.

E) A recursividade é técnica desnecessária caso a programação de um algoritmo seja efetuada em uma linguagem orientada a objetos, uma vez que o envio de uma mensagem a um objeto cria um contexto aninhado que corresponde, indiretamente, à técnica de recursão.

Resolução desta questão na plataforma.