SUMÁRIO

FUNDAMENTOS DE PROGRAMAÇÃO

1. FUNDAMENTOS DE PROGRAMAÇÃO

Programar é o ato de **ensinar uma máquina ou computador na execução de tarefas e na tomada de decisões**, por meio do processo de **criação de conjuntos de instruções** que formam um **algoritmo**. Essas instruções são formadas por **regras definidas para executar algo ou gerar um resultado**.

Programar também pode ser considerado o **processo de escrita, teste e manutenção de um programa de computador ou algoritmo**. Diferentes partes de um programa podem ser escritas em diferentes linguagens. Mas afinal, **o que é um algoritmo?**

Algoritmo é uma "**receita**" que é utilizada para **executar um conjunto predeterminado de tarefas ou resolver algum problema com um número finito de passos**.

Conforme essa definição de **algoritmo**, percebemos que, de forma inconsciente, executamos diversos algoritmos em nosso dia a dia. Observe alguns exemplos abaixo:

- **Algoritmo 1 – Somar 3 (três) números:**
  - **Instrução 1**: receba os três números a serem somados.
  - **Instrução 2**: some os três números.
  - **Instrução 3**: informe qual é o resultado da soma.
- **Algoritmo 2 – Ir para a escola:**
  - **Instrução 1**: acordar cedo.
  - **Instrução 2**: ir ao banheiro para tomar banho.
  - **Instrução 3**: ir ao guarda-roupa.
  - **Instrução 4**: escolher a roupa.
  - **Instrução 5**: vestir a roupa.
  - **Instrução 6**: preparar o café.
  - **Instrução 7**: pegar uma condução.
  - **Instrução 8**: descer próximo à escola.
- **Algoritmo 3 – Receita de bolo:**
  - **Instrução 1**: bata em uma batedeira a manteiga e o açúcar.
  - **Instrução 2**: junte as gemas uma a uma até obter o creme homogêneo.

- **Instrução 3**: adicione o leito aos poucos.
- **Instrução 4**: desligue a batedeira e adicione farinha de trigo, o chocolate em pó e o fermento.
- **Instrução 5**: bata as claras em neve e junte-as à massa de chocolate misturando delicadamente.
- **Instrução 6**: junte uma forma retangular pequena com manteiga e farinha.
- **Instrução 7**: leve para assar em forno médio pré-aquecido por aproximadamente 30 minutos.

Perceba que, em cada algoritmo acima, tentamos "**solucionar um problema**", mas, para isso, torna-se necessário **seguir os passos sequencialmente e com um fim estabelecido e finito**. É com base na representação dos algoritmos que um software de computador, os aplicativos e jogos funcionam.

Obviamente, para que os computadores sigam as instruções do algoritmo, eles precisam de uma linguagem própria que chamamos de **Linguagem de Programação**. Essa linguagem é uma forma de **abstrair a comunicação dos seres humanos e computadores**, fazendo o uso de **códigos e comandos que possibilitem a máquina entender o que é necessário para ser executado**.

Existem diversas linguagens de programação utilizadas no mundo, tais como Java, PHP, Python, C# etc. Cada uma delas possui **regras, requisitos mínimos e sintaxe limitada para sua utilização**, pois **possibilita uma padronização e organização na execução dos passos a serem seguidos nas instruções dos algoritmos**. Observe abaixo, um exemplo de código utilizando a linguagem de programação Java e C++:

```
class MeuPrimeiroProgramaJava {

    public static void main() {

        System.out.print("Olá, mundo");

    }

     }

// Meu primeiro programa em C++

#include <iostream.h>

main()
{
    cout << "Olá, Mundo!" << endl;
    return 0;
}
```

Como cada **linguagem de programação** acima possui suas próprias regras e sintaxes específicas, utilizaremos aqui um **tipo de pseudocódigo,** chamado **português estruturado**, que nada mais é que uma linguagem de programação cujos **comandos estão em português**, o que facilita a compreensão da execução do nosso programa.

Pseudocódigo é uma **técnica textual de representação de um algoritmo**. Nele, os **verbos** (ações) disponíveis para utilização são **limitados e empregados no imperativo**, deve-se **evitar as expressões excessivamente longas**, com o objetivo de **eliminar a possibilidade de ambiguidade**.

A técnica é baseada em uma **PDL** (**Program Design Language**), que é uma **linguagem genérica** na qual é possível **representar um algoritmo de forma semelhante à das linguagens de programação**.

A estrutura de um algoritmo em **pseudocódigo** pode variar um pouco de acordo com o autor ou com base na linguagem de programação que será utilizada posteriormente, mas essas **variações ocorrem apenas na sintaxe**, pois a **semântica deve ser exatamente a mesma**.

A estrutura empregada para a **construção de nossos pseudocódigos** será a seguinte:

> ***Algoritmo*** *<nome do algoritmo>*
> //tem o objetivo de identificar o algoritmo, deve-se utilizar um //nome o mais significativo possível, para facilitar a //identificação.
> ***Var***
> *<declaração de variáveis>*
> //aqui são informadas quais variáveis e seus respectivos tipos //serão utilizadas no algoritmo.
> ***Início***
> *<instrução 1>*
> *<instrução 2>*
> ...
> *<instrução n>*
> // aqui será escrito a sequência de comandos que deve ser //executada para solucionar o programa em questão.
> ***Se*** *<condição>* ***então***
> ***início***
> *<demais instruções>*
> ***fim***
> ***Senão***
> ***início***
> *<demais instruções>*
> ***fim***
> ***Fim-Se***
> ***Fim*** *<marca o fim do algoritmo>*

Observe um exemplo abaixo de **pseudocódigo** que recebe um valor inteiro, fornecido pelo usuário, e retorna esse valor inteiro no monitor:

```
algoritmo "exemplo 1"
var x: inteiro
inicio
leia(x)
escreva(x)
fim
```

Agora, vamos pegar esse primeiro exemplo e adicionar uma nova instrução, vamos adicionar duas unidades (somar com 2) ao número digitado. Observe:

```
algoritmo "exemplo 2"
var numero, resposta: inteiro
inicio
escreva("Digite um número inteiro: ")
leia(numero)
resposta ← numero + 2
escreva("Resultado (numero + 2): ", resposta)
fim
```

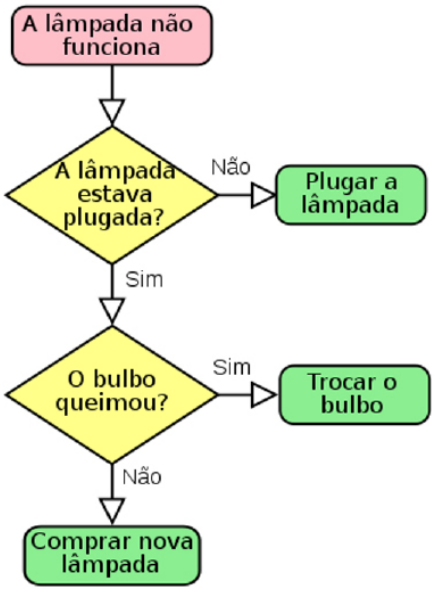
Além do **português estruturado**, é possível representar um algoritmo por meio de **Fluxograma**, que é uma espécie de **diagrama utilizado para documentar processos**, **facilitando a visualização** e **possibilitando encontrar falhas ou problemas na eficiência**.

A utilização de **fluxogramas** para explicar um algoritmo **facilita o entendimento de uma maneira visual**. Enquanto o **algoritmo** realiza uma **análise passo a passo de um processo**, o **fluxograma** ajuda a **explicar os passos de um programa de maneira gráfica**.

Fluxograma é um **tipo de diagrama**, e pode ser entendido como uma **representação esquemática de um processo ou algoritmo**, muitas vezes feito por meio de **gráficos** que ilustram de forma descomplicada a **transição de informações entre os elementos que o compõem**, ou seja, é a **sequência operacional do desenvolvimento de um processo**, o qual caracteriza: o trabalho que está sendo realizado, o tempo necessário para sua realização, a distância percorrida pelos documentos, quem está realizando o trabalho e como ele flui entre os participantes deste processo.

Portanto **um fluxograma representa graficamente um algoritmo** com ajuda de diferentes **símbolos, formas e setas** para **demonstrar um processo ou programa**. O objetivo principal de usar um **fluxograma** é **analisar diferentes métodos**.

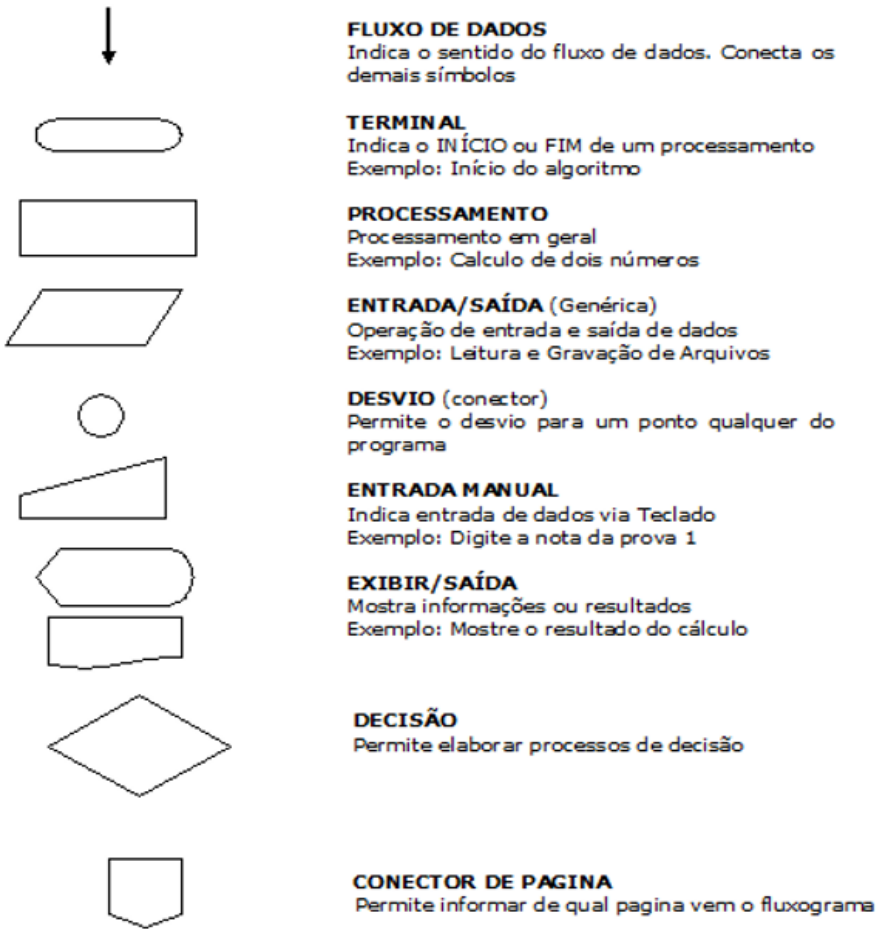
Observe um exemplo abaixo, com um **fluxograma** simples mostrando como lidar com uma lâmpada que não funciona:



Os **fluxogramas** são muito utilizados em projetos de software para **representar a lógica interna dos programas**, mas podem também ser usados para **desenhar processos de negócio e o workflow que envolve diversos atores corporativos no exercício de suas atribuições**.

O **fluxograma** pode ser definido também como o **gráfico em que se representa o percurso ou caminho percorrido por certo elemento** (por exemplo, um determinado documento), por meio dos vários departamentos da organização, bem como o tratamento que cada um vai lhe dando.

Os principais símbolos de um **fluxograma** são:



Outro detalhe importante é que, diferente do **português estruturado**, a **maioria das linguagens de programação são baseadas na língua inglesa**, ou seja, para a programação, esse é o idioma utilizado para a **sintaxe das linguagens e para a maioria das referências**.

Note que, na representação acima do **fluxograma**, há uma **estrutura e sintaxe específicas que precisam ser seguidas** para melhor entendimento de **qual será o resultado do algoritmo**.

Toda linguagem de programação tem uma **sintaxe** a ser seguida, que é um **conjunto de palavras e regras que definem o formato das sentenças válidas**. Algumas **palavras são reservadas**, pois fazem parte da **sintaxe da linguagem** e **não podem ser usadas para outro propósito em um algoritmo** que não seja aquele previsto nas regras de sintaxe. Tais palavras, também são chamadas de **palavras chaves**. Nos exemplos acima que trabalhamos, **as palavras chaves estão em negrito**.

As palavras chaves têm um fim específico no programa e não devem ser usadas para qualquer outra coisa além daquilo que elas se propõem a fazer.

Todo programa possui uma **estrutura básica** que **estabelece como os comandos e as instruções devem ser incluídos em um algoritmo**, estabelecendo um **começo**, um **meio** e um **fim**. Quando escrevemos um programa, ele deve **conter uma sequência que será seguida**, conforme exemplificado abaixo:

```
Algoritmo "MeuPrimeiroAlgoritmo"
início
imprima("Olá, mundo!")
fim
```

Observe no código acima que todo algoritmo/programa deve começar com a palavra reservada "**algoritmo**" seguida de "**início**", uma ou mais funções e finalizar com a palavra "**fim**". Observem que acabamos de ver duas regras: uma para criar um programa e outra para chamar uma instrução, e a isso chamamos de **sintaxe**.

As **palavras reservadas** são **componentes da própria linguagem ou algoritmo** e **não podem ser redefinidas**, ou seja, denominar elementos criados pelo desenvolvedor. Por exemplo, a palavra "**início**", palavra reservada que **indica o início de um bloco de código**, não pode ser nome de rótulos, constantes, tipos ou quaisquer outros identificadores no programa.

Há situações em que erramos algo no nosso algoritmo/programa que **fere as regras de sintaxe da linguagem** que estamos usando (**português estruturado**, no nosso caso). Esses erros são chamados de **erros de sintaxe**, pois **impedem a execução de um programa**. Por exemplo, temos que colocar "**apóstrofo**" ao invés de "**aspas**" ao **indicar a mensagem que você quer que o computador exiba na tela**, impedindo o correto funcionamento do programa.

Outra prática bastante recomendada e utilizada nos algoritmos e nas linguagens de programação são os **comentários** em partes que **necessitam de explicação e que são ignorados pelas linguagens**, pois eles servem apenas para **detalhar determinadas instruções**.

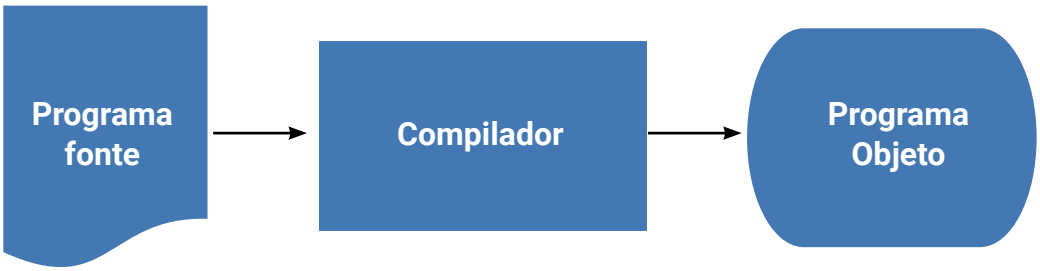
Na maioria das linguagens, para **criação de um comentário em uma linha**, basta colocar os caracteres "**//**" seguido do texto que é o seu **comentário** ou, caso você precise escrever textos muito longos, utilizamos o **comentário multilinhas** que é feito com a sintaxe: "**/***" seguido do texto que pode estar em várias linhas seguido de "***/**". Observe o exemplo a seguir:

```
/*
* Esse é o meu primeiro algoritmo.
* O resultado dele é a exibição na tela do texto "Olá, mundo!"
*/
algoritmo "OláMundo"
inicio // início do programa
imprima("Olá, mundo!")
// a função utilizada escreve uma mensagem na tela
fim /* fim do programa */
```

Os programas de computador escritos em alguma linguagem de programação precisam ser **compilados ou interpretados** a fim de que sejam **executados por um** "**computador**". As instruções escritas em determinada linguagem **são convertidas para linguagem de máquina,** e esse processo chamamos de **compilação**.

O obtivo de se utilizar **compilador** é fazer com que **um programa escrito em uma linguagem de programação seja compreendido pelo computador** e **possa executar todas as instruções escritas**.

Vale salientar que não trataremos aqui como os **compiladores** e os **interpretadores** funcionam, mas precisamos **conhecer a sintaxe da linguagem de programação** a fim de que o **compilador possa traduzir para a linguagem que a máquina possa executar**.



A imagem acima representa uma ideia de um **compilador**, em que há um **programa fonte escrito em determinada linguagem de programação de alto nível**, por exemplo, a linguagem de programação Java, **o compilador que analisa o programa fonte e traduz para uma linguagem de máquina que será executado pelo computador**.

Quando há um **erro de sintaxe**, o **compilador informa ao usuário**. A maioria dos compiladores são amigáveis e possuem um tratamento de erros que informa ao usuário qual é o **tipo de erro** e **onde** (qual linha do programa, por exemplo) **o erro acontece**.

Vale acrescentar que, quando dizemos que determinado programa é portável (**portabilidade**), significa que ele **pode ser executado em diferentes plataformas**, tais como Windows, Linux, Mobile etc., ou seja, podemos dizer que o **compilador consegue traduzir uma linguagem fonte para diferentes linguagens de máquina**.

Dentre as principais propriedades de um **compilador,** vale destacar:

- **Geração do código objeto correto e independentemente do tamanho do programa fonte, desde que a quantidade disponível de memória permita.**
- **Deve exibir mensagens de erro de forma clara, informando o local exato do erro.**

Vala salientar que, mesmo que um programa seja compilado corretamente, isso não garante que o resultado sairá de forma correta, pois alguns erros podem acontecer e envolvem a semântica do programa.

Destaco também a **utilização de compiladores em editores de texto**, quando digitamos um texto e/ou uma palavra em um editor e ele sublinha, significa que o processador utiliza um compilador que, na medida em que o usuário está digitando, o **compilador está verificando erros ortográficos e gramaticais**.

Por fim, para exemplificar, dentre alguns compiladores temos: GCC, pacote de programas responsáveis por fazer a compilação do seu código em C, C++, Objective-C, Fortran, Ada, Go, and D. Além disso, o NetBeans é uma **Ambiente de Desenvolvimento Integrado** (**IDE**) mas também **compilador de códigos em diversas linguagens**, como C, C++ e PHP. Em resumo, **muitas IDEs possuem compiladores na sua estrutura de ferramentas e que provavelmente são utilizados**.

Outro meio de **traduzir um código de alto nível para um código que o computador compreenda**, ou seja, linguagem de máquina, é por meio da utilização de um **interpretador**. Para traduzir um programa escrito em uma linguagem de alto nível, o interpretador gera um novo código, chamado de código objeto.

O **interpretador** trabalha **interpretando o código linha por linha à medida em que ele vai sendo executado** e, dessa forma, dizemos que **o programa é interpretado**. Vejamos a imagem abaixo:

Na imagem acima, é **representado o funcionamento do interpretador**, em que **as entradas são os programas fontes e as entradas do usuário**, sendo que **o interpretador produz a saída correspondente**.

O interpretador não traduz, mas **realiza a interpretação do código fonte e de sua respectiva entrada**. Dessa forma, a **interpretação** é um processo mais lento e frequentemente requer mais espaço em memória.

Por exemplo, a linguagem Java, que ficou muito popular, tinha como principal crítica a lentidão dos programas gerados. A linguagem Java não era apenas interpretada, era chamada de híbrida pois gerava um código intermediário, chamado de **byte code,** que **facilitava o processo de interpretação**.

Quando um programa é interpretado, **em vez de traduzir de uma vez todo o programa fonte para então executar**, **são traduzidas apenas pequenas unidades básicas do programa para imediatamente serem executadas**. Este é um dos principais objetivos da **interpretação**, por isso dizemos que **não há tradução**, pois o interpretador refere-se não só ao programa que simula a execução do código intermediário, mas a todo o processo.

1.1. TIPOS DE DADOS PRIMITIVOS

Os programas de computador processam entradas e retornam saídas em seu resultado. **As entradas e saídas são dados** que podem ser: **numéricos**, **letras**, **texto**, **imagens** etc. Em um programa, os **dados podem ser representados em duas formas**:

- **Constantes: representam dados que não são modificados durante sua execução. Por exemplo, ao definirmos uma constante matemática, chamada PI com valor aproximado de 3,14, a cada execução do programa a referida constante terá o mesmo valor.**
- **Variáveis: representam dados que são manipulados e/ou possuem seu valor alterado durante a execução. Por exemplo, ao definirmos uma variável chamada média, que representa a média aritmética entre dois números, poderá assumir diferentes valores dependendo da quantidade de valores de entrada no programa.**

Os **tipos de dados primitivos** são utilizados para otimização da memória ao definirmos uma variável. São **tipos básicos que devem ser implementados por todas as linguagens de programação**, tais como: **números reais**, **inteiros**, **booleanos** (lógicos), **caracteres** e **strings**. Por exemplo, podemos ter uma variável nome, que armazena textos, já a variável idade pode armazenar apenas números inteiros, na variável sexo podemos armazenar apenas um caractere (“M” ou “F”).

Devemos **especificar em nossos algoritmos o tipo de cada variável**.

Existem **duas categorias** na qual podemos classificar: **tipos de dados primitivos** e **os tipos de dados customizados**.

Existem **quatro tipos de dados primitivos**, e, de acordo com a capacidade de memória necessária para cada variável, as linguagens subdividem estes tipos de dados, mas, no geral, os **tipos primitivos** são:

- **Inteiro: para valores numéricos negativos ou positivos, sem casas decimais. Por exemplo, uma variável que irá armazenar a idade de algum usuário.**
- **Real: para valores numéricos negativos ou positivos, com inclusão de casas decimais. Por exemplo, uma variável que irá armazenar informações de peso ou de altura.**
- **Lógico (ou booleano): assume somente dois valores, verdadeiro ou falso. Reserva um bit na memória, em que o valor 1 representa “Verdadeiro” e o valor 0 representa “Falso”. Por exemplo, uma variável que representa o estado de uma lâmpada, entre acesa ou apagada.**
- **Texto: para variáveis que armazenam textos. Por exemplo, uma variável que irá armazenar o nome de uma pessoa. O tipo de dado texto consiste em um único símbolo ou de**

uma concatenação (conexão) de símbolos do alfabeto utilizado pelo português estruturado. Este alfabeto inclui todas as letras do alfabeto romano, os dígitos, "0, 1,..., 9", e os caracteres de pontuação, tais como "?", entre muitos outros símbolos. Os elementos do conjunto de valores do tipo de dado texto devem ser escritos, nos algoritmos, entre aspas duplas, como, "Jose da Silva".

Vale acrescentar que, particularmente em algumas linguagens de programação, há uma **divisão dos tipos primitivos** de acordo com o espaço necessário para os valores daquela variável.

Na linguagem Java, por exemplo, o **tipo de dados inteiro é dividido em 4 tipos primitivos**: **byte**, **short**, **int** e **long**. A capacidade de armazenamento de cada um deles é diferente, conforme segue abaixo:

- **byte: é capaz de armazenar valores entre -128 até 127.**
- **short: é capaz de armazenar valores entre – 32768 até 32767.**
- **int: é capaz de armazenar valores entre –2147483648 até 2147483647.**
- **long: é capaz de armazenar valores entre –9223372036854775808 até 9223372036854775807.**

O principal objetivo dessa divisão é **otimizar a utilização da memória**, pois em algumas linguagens **não é necessário especificar o tipo de dados da variável**, o que pode ocorrer dinamicamente. Porém **é necessário informar o tipo de dados de cada variável e/ou constante em algoritmos**.

VOCÊ SABIA?

Constantes são dados que simplesmente **não variam com o tempo**, ou seja, elas **possuem sempre um valor fixo**. **Variáveis** representam dados que **são manipulados e/ou possuem seu valor alterado durante a execução**. **Variáveis e constantes necessitam de espaços em memória do computador a serem reservados para guardar informações ou dados**. **Variáveis** estão **associadas a posições na memória RAM**, **armazenando diversos tipos de dados**. **Variáveis e constantes possuem um nome identificador que abstrai o nome do endereço na memória que ela ocupa**.

1.2. DECLARAÇÃO E INICIALIZAÇÃO DE VARIÁVEIS

As linguagens de programação utilizam diferentes formas para **declarar e inicializar variáveis**, por isso, como não trataremos uma linguagem específica, será exemplificado utilizando-se o **português estruturado** para facilitar o entendimento.

O **conteúdo de uma variável** pode ser **alterado, consultado ou apagado diversas vezes durante a execução de um programa**, porém **o valor apagado é perdido**. Já **as constantes são fixas e não podem ser alteradas**.

Vale destacar que, para **atribuir valor a uma variável ou constante,** é necessária uma notação específica para **associar um determinado valor**, ou seja, para **guardar o conteúdo no endereço de memória específico**. Vale lembrar que cada linguagem de programação adota uma maneira de representá-la.

Como forma de demonstrar a **inicialização de variáveis**, iremos utilizar a notação de atribuição **seta** (←), muito utilizada no **Português Estruturado**. Observe um exemplo abaixo:

constante real pi ← 3,14;
constante inteiro meses ← 12;
variavel inteiro idade ← 18;
variavel real peso ← 23.14;
logico repetente;
texto nome ← "Antonio", sobrenome ← "Sousa";

As **variáveis** são o **elemento básico de processamento**. A sua declaração permite **definir que tipo de informação irá conter**. A **declaração de variáveis** segue as seguintes regras:

- **O nome tem de começar por uma letra ou pelo caractere sublinhado (_) e não pode ser uma palavra reservada da linguagem.**
- **O valor de inicialização tem de ser compatível com tipo de variável definido.**
- **Se o valor de inicialização for omitido, a variável é inicializada com os valores por defeito.**

Tipo	Descrição	Valores	Valor por defeito
Inteiro	valores ordinais definidos com quatro bits	-2 147 483 648 2 147 483 647	0
Real	Valores com parte decimal definidos com 64 bits	-1.7 E 308 1.7 E 308	0.0
Lógico	Valore lógicos - 1 bit	verdadeiro falso	falso
Carácter	Caracteres da Tabela ASCII	ASCII(0) ASCII(255)	" " (espaço)
Texto	Conjuntos de caracteres	"Sequências de caracteres" "entre aspas"	"" (vazio)

- **É possível definir mais que uma variável utilizando o caractere vírgula (,).**
- **É possível omitir a palavra variável.**

As **constantes seguem as mesmas regras que a declaração de variáveis**, exceto que **não é possível omitir o valor de inicialização**.

Na **declaração de variáveis**, podemos utilizar a palavra reservada "**var**", de acordo com a instrução abaixo:

> ***var***
> *<identificador 1>, < identificador 2>,..., <identificador n>: <tipo das variáveis>;*

Em que <**identificador 1**> é o nome (**identificador**) **de uma variável** e <**tipo das variáveis**> determina o **tipo de valor que as variáveis poderão receber**. Os **identificadores das variáveis** são usados para **referenciar as variáveis dentro do algoritmo**. Devem ser claros e precisos, dando uma ideia do "**papel**" da variável no algoritmo.

Existem **regras básicas para formação dos identificadores**, são elas:

- **Os caracteres aceitos são números, letras maiúsculas, letras minúsculas e caractere sublinhado (_).**
- **O primeiro caractere deve ser sempre letra ou sublinhado.**
- **Não é permitido espaços em branco ou caracteres especiais, tais como: @, +,$,%.**
- **Não podemos utilizar palavras reservadas como identificadores.**

Exemplo de **identificadores válidos**:

- **A**
- **A**
- **Nota**
- **NOTA**
- **X36**
- **IDADE**
- **_peso**
- **_nome**

Exemplo de **identificadores inválidos**:

- **3a – não pode iniciar com número.**
- **If – não pode utilizar palavra reservada.**
- **Nota(3) – não pode utilizar parênteses.**
- **Nota 1 – não pode utilizar espaço em branco.**

Para identificar uma variável ou uma constante, devemos utilizar um **nome simbólico**, porém é importante lembrar que **palavras-chaves não devem ser utilizadas** para esse fim, ou seja, palavras associadas a comandos e definições da linguagem (tais como if, for e int) são **reservadas**, **não podem ser utilizadas para o nome de variáveis e/ou constantes**.

Utilizando o **português estruturado**, observe um exemplo de declaração de variáveis e constantes:

idade, numeroApartamento: ***inteiro****;*

nomePessoa, endereço: ***texto****;*

mediaSalario, altura: ***real****;*

Os nomes de variáveis podem ser de qualquer tamanho, sendo que usualmente nomes significativos devem ser utilizados.

VOCÊ SABIA?

Algumas linguagens de programação fazem **distinção entre caracteres maiúsculos e caracteres minúsculos**, de forma que a variável Salariomedio é diferente de SalarioMedio, a constante Pi é diferente de PI. A essas diferenças chamamos **case sensitive,** que significa que **caracteres em caixa alta e em caixa baixa são tratados de modo diferentes**.

Conforme observado, diversas **variáveis de um mesmo tipo podem ser declaradas em um mesmo comando**, sendo que o **nome de cada variável deve ser separado por vírgulas**. Além disso, **variáveis podem ser também inicializadas enquanto declaradas**, observe o exemplo abaixo:

constante
inteiro NotaMaxima ← 100;
var
inteiro idade ← 0;
real altura ← 1,79;
logico statusLampada ← 0; (onde o valor 0 é apagada e 1 é acesa)
Texto nomePessoa ← "Jose da Silva";

Uma variável cujo valor não será alterado pelo programa pode ser qualificada como **constante**. Vale ressaltar que a constante NotaMaxima **não poderá ter seu valor alterado**. Evidentemente, constantes deste **tipo devem ser inicializadas no momento de sua declaração**.

1.3. UTILIZAÇÃO DE LITERAIS E STRINGS

Existem **tipos de dados que podem ser decompostos**, é o caso das **literais** ou **strings**. Elas são utilizadas para **armazenar caracteres agrupados**, ou seja, caso seja necessário representar o nome de uma pessoa, como “JOAO”, é possível **decompor em caracteres**: “J”, “O”, “A”, “O”.

Alguns detalhes importantes sobre **manipulação de strings**:

- **Sempre devemos utilizar aspas (“”):**

  texto ← “O nome é:”;

- **Para representar o valor vazio para uma string, devemos seguir o exemplo abaixo:**

  texto ← “”;

- **Quando comparamos duas strings, as diferenças entre maiúsculas e minúsculas são desconsideradas, então:**

  se (“CASA” = “casa”) entao = Verdadeiro;

Muitas linguagens de programação, tais como Java, C#, PHP etc., possuem funções que permitem a **manipulação de strings** (caracteres/literais) e isto facilita bastante a sua utilização. Abaixo, segue um exemplo de um **algoritmo com strings**:

```
/*
* Esse é o meu primeiro algoritmo utilizando strings
* O resultado dele é a exibição na tela do texto “Olá, mundo!”
*/
Algoritmo “MeuExemploComStrings”
var
nome1: texto;
nome2: texto;
meutexto: texto;
inicio
meutexto ← “Resultado: “;
escreva(“Digite o nome 1:”);
leia(nome1);
escreva(“Digite o nome 2:”);
leia(nome2);
se (nome1 = nome2) entao
escreva(texto, “Iguais”);
senao
escreval(texto, “Diferentes”);
fimse
fim
```

1.4. CATEGORIAS DE OPERADORES E PRECEDÊNCIA

Para solucionar alguns problemas, os **algoritmos utilizam expressões matemáticas**, como operadores de adição, subtração, multiplicação etc. Além disso, existem outros mais complexas, como raízes, exponenciais, logaritmos.

Uma expressão que utiliza a **operação de adição**, representa uma **soma entre dois números que ao final produz um resultado**. É nesse momento que entram os operadores nos algoritmos, pois eles possuem o objetivo de **relacionar valores para resultar um outro valor**. Existem três tipos de operadores: **aritméticos**, **lógicos** e **relacionais**.

A representação de alguns operadores pode mudar de acordo com cada linguagem de programação, mas geralmente todas utilizam o mesmo símbolo.

OPERADORES ARITMÉTICOS

Os **operadores aritméticos** definem as **operações que podem ser realizadas sobre os números inteiros e reais**. Para os **inteiros**, as **operações aritméticas são a adição, subtração, multiplicação e resto da divisão**. Já para os números **reais**, as **operações aritméticas são a adição, subtração, multiplicação e divisão**.

Na linguagem do **português estruturado**, os **operadores aritméticos correspondentes às operações definidas sobre os inteiros** são:

- **+ (adição).**
- **- (subtração ou menos unário).**
- ∗ **(multiplicação).**
- **\ (divisão inteira).**
- **% (resto, aplicado apenas aos valores inteiros).**

Com base nos operadores acima, podemos **escrever expressões aritméticas** que podem **envolver constantes e/ou variáveis inteiras**. Como exemplo, suponha que idade1, idade2 e idade3 sejam variáveis do tipo inteiro. Então, temos que:

```
nota1 + nota2 + nota3;
nota1 – Nota2 ∗ Nota3%2;
- 7 + 4 ∗ 8 \ 2;
```

Todas as expressões aritméticas acima são **válidas na linguagem português estruturado**.

A **precedência de operadores** determina a **ordem em que os operadores são processados**. Aqueles com **maior precedência** são **processados primeiro**. Na expressão “5*3+2”, o **operador**

de multiplicação (*) tem **maior precedência que o operador de adição** (+). Podemos dizer que o valor da expressão é igual a 17, se avaliarmos 5 * 3 primeiro e, depois, a expressão 15 + 2.

Da mesma forma que na matemática os **operadores de multiplicação e divisão têm precedência de execução em relação aos operadores de soma e subtração**. Vale destacar que, se tiver **parênteses** na expressão, estes têm **precedência ainda maior**. Os **operadores de mesma prioridade são interpretados da esquerda para a direita**.

Abaixo, listamos a procedência de **operadores aritméticos**:

Operador	Símbolo	Prioridade
Multiplicação	*	2º
Divisão	/	2º
Adição	+	3º
Subtração	-	3º
Exponenciação	^	1º

A seguir, demonstraremos um algoritmo utilizando operadores aritméticos:

```
/*
* Esse algoritmo tem o objetivo de somar dois números inteiros;
*/
Algoritmo "Quadrado da soma de 2 inteiros"
var
a, b, quadrado: inteiro;
inicio
escreva("Entre com o primeiro inteiro:");
leia(a);
escreva("Entre com o segundo inteiro:");
leia(b);
quadrado ← (a+b)*(a+b);
escreva("O quadrado da soma dos inteiros lidos e:", quadrado );
fim
```

OPERADORES RELACIONAIS

São utilizados para **comparar números e literais**, **retornando valores lógicos**. Não existe um tipo específico para a representação de valores lógicos (**verdadeiro** e **falso**). Entretanto **qualquer valor pode ser interpretado como um valor lógico**.

Podemos definir que 0 (**zero**) **representa falso** e **qualquer outro valor representa verdade**. Também é usual **representar verdade** com o valor 1 (**um**). Para produzir um valor lógico (**verdadeiro** ou **falso**), usamos os **operadores relacionais**.

Dessa forma, podemos **comparar dois valores** e **temos como resultado da avaliação de um operador relacional 0, se a comparação é falsa,** e **1, se a comparação é verdadeira**.

Abaixo, listamos os **operadores relacionais**:

Operador	Resultado
x=y	Verdade se x for igual a y
x!=y ou x<>y	Verdade se for diferente de y
x<y	Verdade se x for menor de y
x>y	Verdade se x for maior que y
x<=y	Verdade se x for menor ou igual a y
x>=y	Verdade se x for maior ou igual a y

OPERADORES LÓGICOS

Os **operadores lógicos** são "**and**", "**or**" e "**not**", que significam "**e**", "**ou**" e "**não**", sendo utilizados respectivamente para **conjunção**, **disjunção** e **negação**. Algumas linguagens oferecem **operadores lógicos** que funcionam da mesma forma que os **operadores da lógica matemática**. A tabela abaixo apresenta os **operadores lógicos**, considerando a linguagem do **português estruturado**:

Operador
E/And
Ou/Or
Não/Not

Os **operadores lógicos** utilizam **dados lógicos ou booleanos**, pois o objetivo continua o mesmo: **relacionar valores para resultar em um outro valor**. Isso quer dizer que os **operadores lógicos** podem relacionar os seguintes valores: verdadeiro/falso, 1/0, aceso/apagado, ligado/desligado, true/false, e o **resultado de uma expressão lógica também é outro valor lógico**.

Abaixo, listamos as **tabelas verdades** dos **operadores lógicos**:

Tabela Operador "E"	Tabela Operador "OU"	Tabela Operador "NÃO"
V – e V = V	V – ou V = V	Não V = F
V – e F = F	V – ou F = V	Não F = V
F e V = F	F ou V = V	
F e F = F	F ou F = F	

Uma **tabela verdade** é um dispositivo utilizado no estudo da lógica matemática. Com o uso dessa tabela, é possível **definir o valor lógico de uma proposição**, isto é, **saber quando uma sentença é verdadeira ou falsa**.

Os **operadores aritméticos, lógicos e relacionais** devem seguir regras essenciais para a correta **avaliação de expressões**:

- **Observar a prioridade dos operadores, ou seja, operadores de maior prioridade devem ser avaliados primeiro. Se houver empate com relação à precedência, então a avaliação se faz da esquerda para a direita.**
- **As avaliações que estejam dentro de parênteses em expressões possuem o poder de maior prioridade dos demais operadores, forçando a avaliação da subexpressão em seu interior primeiro que as avaliações que estejam fora dos parênteses.**

VOCÊ SABIA?

Entre a utilização desses três operadores (**aritméticos**, **relacionais** e **lógicos**), existe uma certa prioridade de avaliação: **os aritméticos devem ser avaliados primeiro**; em seguida, são **avaliadas as subexpressões com operadores relacionais** e, **por último, os operadores lógicos**.

Operador de Concatenação – quando precisamos **juntar dois valores ou variáveis do tipo texto**, utilizamos esse operador. É muito utilizado para **compor mensagens** ao longo do programa. Na maioria das linguagens de programação, o símbolo usado para a **concatenação** é o mesmo da **adição**. Assim, a expressão "João da"+"Silva" teria como resultado o texto"João da Silva". A utilização do **operador de concatenação** é muito comum em **expressões envolvendo variáveis do tipo texto**, caso contrário será necessário algum tipo de conversão a fim de torná-lo um texto.

Operador de Atribuição – este operador é usado para **definição de valor de uma variável**, ou seja, **é guardado na memória apontada pela variável com o valor posicionado do lado direito do operador de atribuição**. Em algoritmo, é representado pelo símbolo: "←" (seta). Em algumas linguagens de programação é utilizado o símbolo ":" (dois pontos). A sintaxe do comando de atribuição é: NomedaVariavel ← expressão. Por exemplo, media ← 6.

1.5. CONTROLE DE FLUXO DE PROGRAMAS E REPETIÇÃO

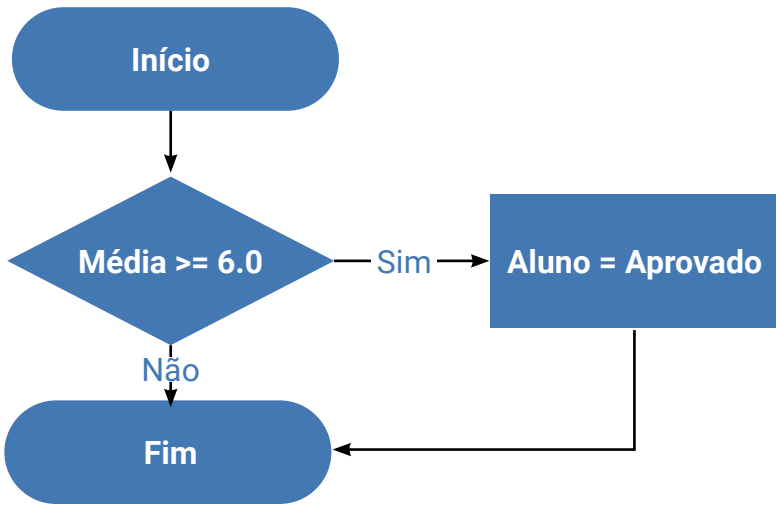
Este tópico permite **interferir na sequência de instruções executadas dependendo de uma condição de teste**, ou seja, o algoritmo terá **caminhos diferentes para decisões diferentes**.

Os algoritmos possuem um **fluxo de execução único**, ou seja, as instruções que são executadas são as mesmas, independentemente dos valores de entrada, e irão **produzir resultados diferentes dependendo desses valores**. Porém, na realidade, algumas vezes executamos determinados procedimentos **dependendo de uma série de situações ou condições**.

Quando construímos um algoritmo, necessitamos **determinar ações diferentes dependendo da avaliação de certas condições**. O uso de **condições** ou **comandos de decisão** muda o fluxo das instruções de um algoritmo ou programa, **permitindo que diferentes instruções ou comandos sejam executadas de acordo com a entrada do programa**. Por exemplo:

se *(media >= 6.0)* ***entao***
aluno = aprovado

Observe esse pequeno código representado por meio do uso de um **fluxograma**:



Em um programa que apresente a média escolar de um aluno, é necessário, além de calcular a média de um aluno, apresentar se ele está aprovado ou reprovado segundo a condição estabelecida para o valor da variável média, em nosso caso, se a média do aluno for igual ou maior que 6.

O controle de fluxo de um algoritmo e/ou programa pode ser classificado de quatro formas diferentes, sendo essa classificação baseada na organização lógica existente em cada situação. Essa classificação **diferencia uma estrutura de decisão** como: **decisão simples**, **decisão composta**, **decisão encadeada** e **decisão de múltipla escolha**.

CONTROLE DE FLUXO SIMPLES OU ESTRUTURA CONDICIONAL SIMPLES

O comando **Se...Fim_Se** – nesse tipo de controle, **uma instrução/comando ou um conjunto de instruções é executado somente se a condição especificada retornar o valor verdadeiro**. Caso o resultado seja falso, nenhuma das instruções constantes na estrutura da seleção será executada, e a execução das instruções será desviada para a instrução seguinte à estrutura de seleção. Observe a estrutura meio do **português estruturado**:

se *(<condição>)* ***então***
<instruções para a condição verdadeira>
fim_se

<instruções para a condição falsa ou após ser verdadeira>
Por exemplo:
Algoritmo *media_de_numeros;*
Var
media: ***real****;*
A: ***inteiro****;*
B: ***inteiro****;*
Inicio
leia(A);
leia(B);
media ← (A+B)/2;
se *(media > 6.0)* ***então***
escreva("Aluno aprovado");
fim_se
escreva("Fim do Algoritmo")
Fim

Note que, no algoritmo acima, **após a palavra reservada var**, há a **definição dos tipos de variáveis**. Em seguida, é **solicitada a leitura dos valores** para as variáveis A e B e, por fim, **esses valores são atribuídos à variável média,** que **realiza a operação desejada**.

De acordo com o resultado calculado na variável média, há uma **condição** que permitirá imprimir o resultado "aluno aprovado" caso esta seja maior que 6.0, e, não sendo, o programa apenas imprime a mensagem "fim do algoritmo", encerrando o programa sem apresentar a média calculada, pois a condição estabelecida resultou em falso.

ESTRUTURA CONDICIONAL COMPOSTA

O comando **Se...Senão...Fim_Se** – na **estrutura de decisão composta**, **uma instrução ou um conjunto de instruções é executado caso o teste condicional especificado retornar o valor verdadeiro** e, **caso o resultado do teste seja falso**, **outra instrução ou um conjunto de instruções é executado**.

Assim, a **condição composta** permite **desviar o fluxo para dois caminhos distintos**. Observe a representação desta estrutura por meio do **português estruturado**:

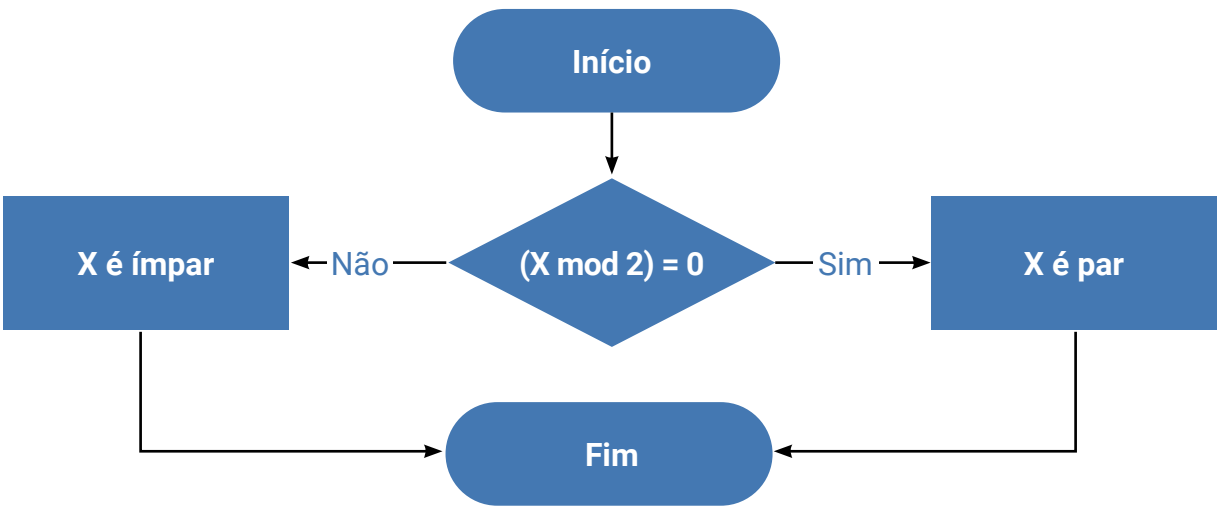
se *(<condição>)* ***então***
<instruções para a condição verdadeira>
senão
<instruções para a condição falsa>
fim_se
<continuação das instruções do algoritmo>

Por exemplo:

```
Algoritmo par_impar;
Var
X: inteiro;
Inicio
leia(X);
se (X mod 2) = 0 então
escreva(X, 'é par');
senão
escreva(X, 'é ímpar');
fim_se
escreva('Fim do Algoritmo');
Fim
```

Note que, **após a palavra reservada var**, em que há uma **definição do tipo da variável**, é **solicitada a leitura do valor** para a variável X. Em seguida é **verificado o resto da divisão inteira** do valor de X por 2, ou seja, é realizada uma verificação se X é divisível por 2. Nessa condição, dependendo do resultado da avaliação da expressão "X mod 2=0", é mostrado o resultado "X é par", caso seja verdadeiro ou "X é ímpar" caso seja falso.

Observe esse pequeno código por meio do uso de um **fluxograma**:



ESTRUTURAS DE DECISÃO ENCADEADAS

Uma instrução de condição pode ser inserida dentro de outra, formando uma estrutura de condição, chamada de **estrutura de decisão encadeada**. O **encadeamento** dessas instruções também é conhecido como **aninhamento de instruções de condição**.

O **encadeamento de instruções** permite tornar uma solução algorítmica ainda mais eficiente, uma vez que **diminui testes condicionais desnecessários**.

Observe o exemplo a seguir:

```
/*
* Esse algoritmo tem o objetivo de calcular a média de duas notas.
*/
Algoritmo media_de_notas;
Var
media: real;
Nota1: inteiro;
Nota2: inteiro;
Inicio
leia(Nota1);
leia(Nota2);
media ← (Nota1 + Nota2)/2;
se (Média >= 5.0) entao
se (Média >= 7.0) entao
escreva("Aluno = Aprovado");
senão
escreva("Aluno = Recuperação");
senão
escreva("Aluno = Reprovado");
fim_se
escreva("Fim do algoritmo");
Fim
```

ESTRUTURAS DE DECISÃO DE MÚLTIPLA ESCOLHA

Há uma estrutura de decisão utilizada em avaliações de valores para uma variável, é a **decisão de múltipla escolha**. Diferentemente das estruturas condicionais anteriores, nesse caso o **teste condicional não retorna um valor lógico**, **mas sim um valor inteiro, real ou caractere**.

Outra diferença está no fato de que essa estrutura **permite desviar o fluxo de execução para caminhos variados**, ou seja, **é verificada a igualdade da expressão de decisão no teste condicional com as opções definidas**, resultando na execução de um bloco, ou de uma única instrução específica, para cada opção.

O comando **Caso** – é adequado quando **precisamos escolher uma entre várias alternativas previamente definidas**, por exemplo, em um menu ou paleta de cores. Para cada "**caso**", é **verificada a constante cujo valor é igual à expressão**, depois são executados todos os comandos seguintes. Caso **não haja o "caso"**, então **é executado o comando associado ao caso padrão/contrário**. Vejamos abaixo sua representação no **português estruturado**:

```
Caso <valor> seja:
<primeiro valor>: <bloco de instruções>
<segundo valor>: <bloco de instruções>
```

<terceiro valor>: <bloco de instruções>
<quarto valor>: <bloco de instruções>
...
<N valor>: <bloco de instruções>
<caso contrário/padrão (nenhuma das opções acima)>: <bloco de instruções>
Fim_Caso

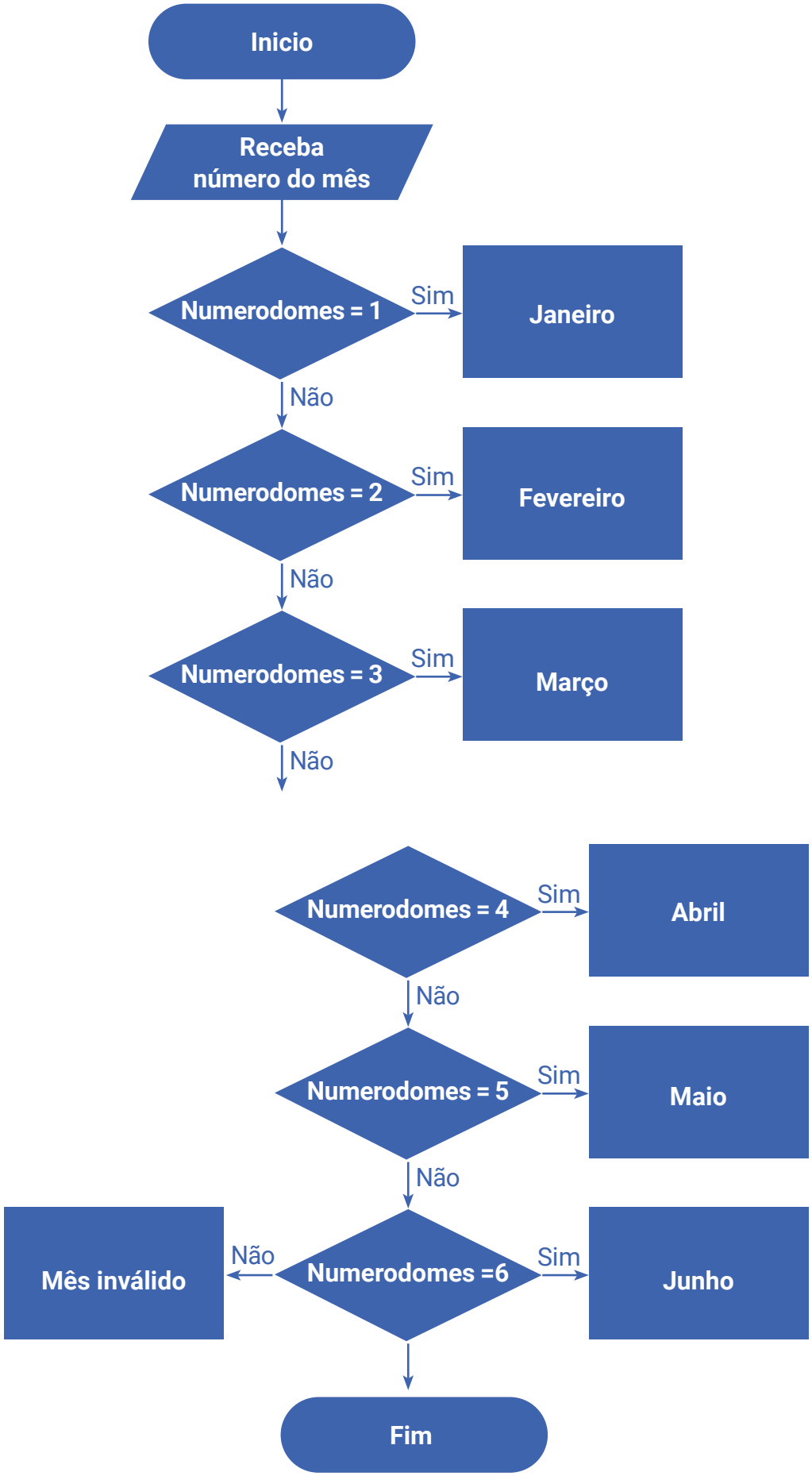
Observe um exemplo utilizando essa estrutura:

***Caso** (Número):*
Caso 1: Cor Vermelha;
Caso 2: Cor Verde;
Caso 3: Cor Azul;
Caso 4: Cor Amarela;
Caso 5: Cor Rosa;
Caso 6: Cor Preta;
Caso Outro: Cor inexistente;
Fim_Caso

Vejamos um exemplo com a utilização da linguagem português estruturado.

***Algoritmo** escreve_mes;*
Var
NumerodoMes: inteiro;
Inicio
Leia (numerodomes);
***Caso** (numerodomes) **seja**:*
1: escreva(“Janeiro”);
2: escreva(“Fevereiro”);
3: escreva(“Março”);
4: escreva(“Abril”);
5: escreva(“Maio”);
6: escreva(“Junho”);
Caso Contrário:
escreva(“Mês inválido”);
Fim_Caso
Fim

Observe esse código por meio do uso de um **fluxograma**:



Vale destacar que a maioria das linguagens de programação de programação utiliza comandos parecidos quanto às estruturas de decisão até aqui estudadas. O comando **Se-Então-Senão** nas linguagens de programação utiliza **If-Then-Else**, no inglês.

ESTRUTURAS DE REPETIÇÃO

Diariamente, realizamos atividades na qual a **repetição de procedimentos é necessária** para executar as tarefas do dia a dia, e, em alguns momentos, essas tarefas são **repetidas várias vezes** e se **encerram quando atingimos o objetivo pretendido**.

Em algumas de nossas tarefas diárias, quando tentamos fechar a tampa de uma garrafa pet, giramos a tampa várias vezes até que a garrafa feche completamente e fique apertado o suficiente para não vazar líquido. Note que, durante esse processo, é verificado a cada volta, se a tampa está bem apartada.

As **estruturas de repetição** são extremamente úteis para **repetir uma série de operações semelhantes que são executadas para todos os elementos de uma lista ou de uma tabela de dados**, ou simplesmente para **repetir um mesmo processamento até que uma certa condição seja satisfeita**.

De forma análoga, podemos perceber diversas atividades da nossa rotina são **repetitivas**. Durante uma fila de banco, por exemplo, o caixa chama o próximo da filha enquanto não chegar ao último da fila. **Outras repetições podem ser qualificadas com antecedência**.

Em algoritmos, essas estruturas tratam uma forma de **executar blocos de comandos somente sob determinadas condições**, mas com a opção de **repetir o mesmo bloco quantas vezes for necessário**.

As **estruturas de repetição** são um modo que temos para **executar blocos de comandos obedecendo determinadas condições**, contando também com a opção de **repetir este bloco quantas vezes precisarmos**.

Observe que as **repetições** têm uma característica em comum que se refere ao fato de que **existe a verificação de condição de repetição,** que pode ser representada por um **valor lógico**, **determinando se a repetição** (comando) **deve ou não prosseguir**. Isso serve de base para a implementação dos comandos de repetição em algoritmos. Ao invés de escrever a mesma instrução várias vezes, ou então, utilizar uma estrutura de condição simples (**IF_THEN_ELSE**) várias vezes, podemos utilizar uma estrutura que indique que uma **instrução será executada quantas vezes forem necessárias**. Observe o exemplo abaixo:

```
Algoritmo imprimir_de_1_ate_5;
Inicio
escreva (1);
escreva (2);
escreva (3);
```

```
escreva (4);
escreva (5);
Fim
```

No exemplo acima, utilizamos a instrução “escreva” repetidas vezes. Para o algoritmo acima, a quantidade de cinco instruções que utilizamos é bem viável e não nos dá tanto trabalho para codificação, porém pense em um algoritmo que precise imprimir milhares de vezes uma sequência de números, ou seja, teremos um trabalho tremendo repetindo comandos, além da necessidade de escrever as sequências de números corretamente.

Em virtude dos problemas apresentados, as linguagens de programação possuem soluções para o problema apresentado acima. A maioria das linguagens de programação utiliza estruturas de repetição para evitar esses problemas: **Para** (**For**), **Enquanto** (**While**), **Faça-Enquanto** (**do-While**).

O comando **Para** (**For**) – define uma estrutura (instruções) que será repetida por um número determinado de vezes. Para que seja possível determinar quantas vezes precisamos executar as instruções, precisamos sempre definir um contador para o comando, isto é, podemos implementar um loop pré-definido. Vejamos a sintaxe abaixo:

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

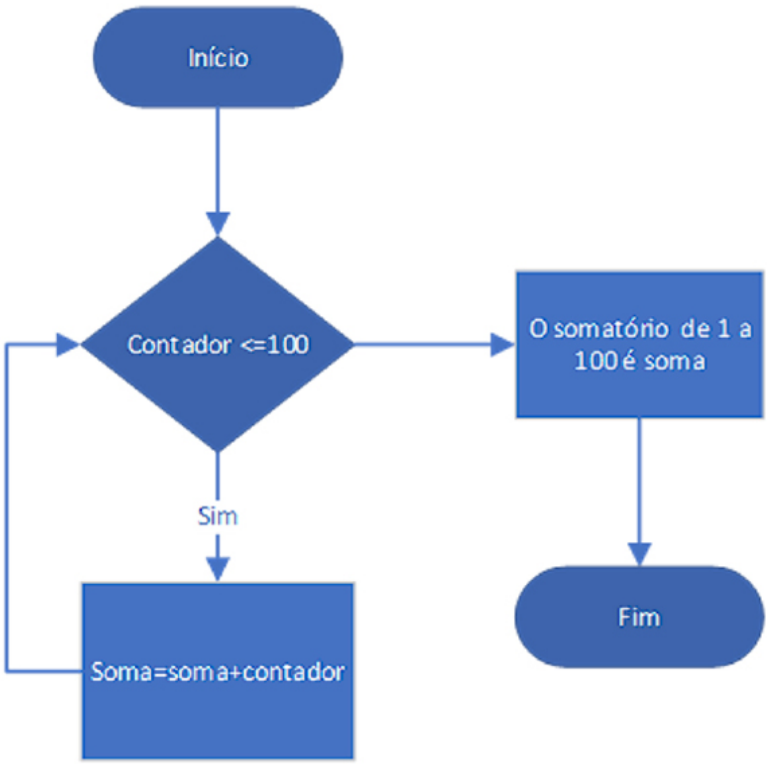
Observe que temos a **separação do comando em três cláusulas**:

- **Preparação: <variável de controle> de <inicio>.**
- **Condição: <variável de controle> até <fim>.**
- **Incremento: passo <incremento>.**

Vejamos os exemplos abaixo:

```
Algoritmo “UtilizandoComando_Para”;
Var
contador, soma: inteiro;
Inicio
soma:= 0;
Para contador de 1 até 100 faça:
soma:= soma + contador;
Fim_Para
escreva(“O somatório de 1 a 100 é: “, soma);
Fim
```

Vejamos uma representação de um algoritmo fazendo uso da estrutura de repetição com o comando **Para** (**For**), utilizando **fluxograma**:



Observe outro exemplo:

Algoritmo *Soma_Impares;*
Var
num,soma: ***inteiro****;*
Inicio
soma ← 0;
Para *(num* ***de*** *1* ***até*** *100)* ***faça****:*
Se *(num mod 2 = 1)* ***então****:*
soma ← soma+num;
Fim_Se
Fim_Para
escreva("A soma dos números impares é:"soma);
Fim

O comando **Enquanto** (**While**) – esse comando de repetição se caracteriza por uma **verificação de encerramento de atividades antes de se iniciar a execução de seu bloco de instruções**. Essa instrução é útil quando não sabemos ao certo a quantidade de repetições necessárias para execução das instruções. Esta estrutura **repete uma sequência de comandos enquanto uma determinada condição** (expressão lógica) **for satisfeita**. A sintaxe do comando segue abaixo:

Enquanto *<expressão-lógica>* ***faça****:*

<instruções>
Fim_Enquanto

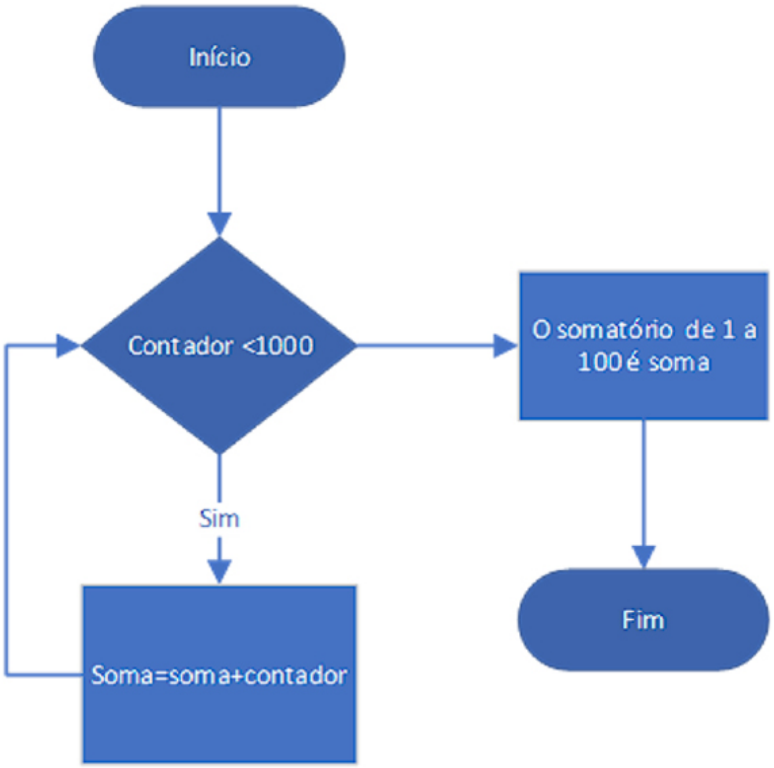
A **expressão-lógica** é **avaliada antes de cada repetição** e, **enquanto seu resultado for verdadeiro**, as **instruções são executadas**.

O comando **Fim_Enquanto** indica o **fim das instruções que serão repetidas**. Cada vez que a execução atinge este ponto, **retorna-se ao início do laço para que a expressão-lógica seja avaliada novamente** e assim sucessivamente, **até que a condição seja falsa** e o **algoritmo seja finalizado**.

Observe um exemplo do uso desse comando de repetição:

Algoritmo *"UtilizandoComando_Enquanto";*
Var
contador, soma: ***inteiro****;*
Inicio
contador:= 1;
soma:= 0;
Enquanto *contador < 1000* ***faça****:*
soma:= soma + contador;
contador:= contador + 1;
Fim_Enquanto
escreva("O somatório de 1 a 100 é:", soma);
Fim

Observe o algoritmo acima utilizando **fluxograma**:



O comando **Faça-Enquanto** (**Do-While**) – a última estrutura de repetição que apresentaremos é o comando **Faça-Enquanto**. O comando funciona de forma similar ao comando **Enquanto**, exceto pelo fato de que **a condição de controle só é testada após a execução do bloco de instruções, e não antes**, como é o caso do comando **Enquanto**. Outra diferença se refere aos comandos que são **repetidos pelo menos uma vez**, já que **a condição de parada se encontra ao final**. Vejamos a sintaxe abaixo:

Faça
<bloco de instruções>
Enquanto *<condição booleana>*

O comando **Faça-Enquanto** é utilizado sempre que tivermos certeza de que **o bloco de instruções será executado ao menos uma vez**, **sem a necessidade do teste na entrada do bloco**. Observe o exemplo abaixo.

Algoritmo *Soma_Media;*
Var
contador: ***inteiro****;*
valor, soma, media: ***real****;*
Inicio
contador ← 0;
soma ← 0;
Faça*:*
escreva("Entre com um valor:");

leia(valor);
soma ← soma + valor;
contador ← contador + 1;
***Enquanto** (contador >= 100);*
media ← soma/contador;
escreva("Soma:", soma);
escreva("Média:", media);
Fim

1.6. SUB-ROTINAS

A utilização de **sub-rotinas** em algoritmos/programas, conhecidas também por **módulos** ou **subprogramas**, tem se tornado muito comum e bastante útil para organização de um algoritmo.

Uma **rotina** é um **conjunto de instruções** (algoritmo), já a **sub-rotina**, **subprograma** ou **módulo** é uma **parte menor desse conjunto de instruções** que, em geral, será **utilizado repetidamente em diferentes locais do sistema** e que **resolve um problema mais específico**, parte do problema maior.

Com o aumento da complexidade dos problemas a serem solucionados, os algoritmos estão cada vez mais densos, e, como solução, surgiu a necessidade de **dividir esses problemas maiores em menores**, com o objetivo de simplificar os algoritmos.

Uma **sub-rotina** é um algoritmo que, geralmente, encerra em si próprio uma parte da solução de um problema maior, que costuma ser o algoritmo a que essa **sub-rotina** está subordinada. As **sub-rotinas** são importantes para:

- **Subdividir algoritmos complexos e facilitar o entendimento.**
- **Melhor estruturação de algoritmos, facilitando a identificação de erros e documentação.**
- **Modularização, facilitando a manutenção e reutilização de sub-rotinas já implementadas.**
- **Redução de código duplicado.**
- **Decomposição de problemas grandes em pequenas partes.**

Uma característica importante está na **reutilização**, pois tem sido adotada em soluções por muitos grupos de desenvolvimento, pois **diminui o retrabalho e gera economia de tempo**.

Ao desenvolver de novo um sistema, devemos tentar ao máximo **utilizar sub-rotinas já existentes**, como **bibliotecas**, de modo que a quantidade de software realmente novo que deve ser desenvolvido seja **minimizada**.

Na maioria das vezes, as **sub-rotinas** são úteis para **encerrar em si uma certa sequência de comandos**, pois, em algumas situações, esse **subprograma** é **chamado várias vezes em um algoritmo** e, nestes casos, **proporcionam uma redução do tamanho de algoritmos maiores**.

Todo o código do algoritmo é **dividido em algoritmo principal e diversos subprogramas**. O **algoritmo principal** é aquele por **onde a execução do algoritmo sempre se inicia**, sendo que este pode **eventualmente invocar as demais sub-rotinas**.

Sub-rotinas são de **instruções que realizam tarefas específicas**, por isso segue algumas características importantes:

- **Carregado uma vez e pode ser executado quantas vezes for necessário.**
- **Podem ser usadas para economizar espaço e tempo de programação, já que podem ser usadas várias vezes em um mesmo programa.**

Observe a **estrutura de uma sub-rotina**:

Algoritmo *<nome do algoritmo>*
Var
<definição das variáveis globais>
<definições das sub-rotinas>
Inicio
<corpo do algoritmo principal>
Fim

Durante a execução do algoritmo principal, ao **encontrar uma chamada de uma sub-rotina**, a **execução desse algoritmo é interrompida** e, a partir daí, **a execução é passada para os comandos definidos no corpo dessa sub-rotina**, que, **ao término**, **retorna ao último ponto de execução do algoritmo principal**, local onde houve a interrupção e a invocação da sub-rotina, prosseguindo para instrução imediatamente seguinte.

Vale destacar que as **sub-rotinas** podem ser uma **função** ou um **procedimento**, sendo que a grande diferença é que no primeiro retorna-se um valor e no segundo, não.

FUNÇÕES

As **funções** são **definidas como entidades separadas do algoritmo principal**. Uma **função retorna um só valor**, **a partir de um determinado conjunto de argumentos**.

São um **tipo especial de procedimento** em que, **depois que a chamada é executada**, **o valor calculado é retornado no nome da função** que passa a ser uma **variável da expressão**.

Observe a **sintaxe e estrutura de criação de uma função**:

Funcao *<nome> (<parâmetros>) <tipo_de_dado>*

Var
<variáveis locais>
Inicio
<instruções>
Fim

Temos que:

- **<nome> nome simbólico pelo qual a função é invocada.**
- **<parâmetros> lista de parâmetros da função.**
- **<tipo de dado> tipo de dado da informação retornado pela função ao algoritmo chamador.**
- **<variáveis locais> definição das variáveis locais que são utilizadas somente pela função.**
- **<instruções> demais instruções do corpo da função.**

No corpo do algoritmo principal, o comando de invocação de um subprograma do tipo **função** sempre aparece **dentro de uma expressão do mesmo tipo de dado que o valor retornado pela função**, que é chamada por meio do simples aparecimento do nome da função, **seguido pelos respectivos parâmetros entre parênteses**.

Após o **término da execução da função**, o trecho do comando que realizou a chamada é **substituído pelo valor retornado pela função chamada** dentro da expressão em que se encontra.

Além disso, **no corpo da função**, há o comando "retorne <expressão>", que é usado para **retornar o valor calculado**. Ao **encontrar o comando retorne**, a **expressão entre parênteses é avaliada**, a **execução da função é encerrada** e o **valor da expressão é retornado ao algoritmo principal**.

Vale lembrar que uma expressão pode ser uma simples constante, uma variável ou uma combinação das duas por meio de operadores. Essa expressão **deve ser do mesmo tipo que o valor retornado pela função**.

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

Algoritmo *Exemplo_de_funcao;*
Var
X, Y, Z: ***real;***
funcao *media(n1, n2: real):* ***real***
Var
med: ***real;***
Inicio
med ← (n1+n2)/2;
retorne *med;*

Fim
Inicio
escreva("Digite a nota 1");
escreva("Digite a nota 2");
leia(X);
leia(Y);
Z ← Media(X,Y);
escreva ("A sua média é", Z);
Fim

PROCEDIMENTOS

O **procedimento** é um **algoritmo que é executado**, enquanto a **função** é um **algoritmo que será executado e irá produzir um resultado**, ou seja, **retornar um valor**, que poderá ser usado no algoritmo onde a função é chamada.

Vale ressaltar que **a chamada de um procedimento nunca surge no meio de expressões**, como no caso de funções. **A chamada de procedimentos só é feita em comandos isolados dentro de um algoritmo**, como as **instruções de entrada** (leia) **e saída** (escreva) **de dados**.

Além disso, são **estruturas que agrupam um conjunto de comandos**, que **são executados quando o procedimento é chamado**.

Observe a sintaxe de um **procedimento**:

Procedimento *<nome> (<parâmetros>)*
Var
<variáveis locais>
Inicio
<demais instruções>
Fim

Em que:

- **<nome> nome simbólico pelo qual o procedimento é chamado.**
- **<parâmetros> parâmetros do procedimento.**
- **<variáveis locais> definições das variáveis locais, análoga ao das variáveis em algoritmos.**
- **<instruções> conjunto de instruções do corpo do procedimento.**

Observe um exemplo:

Algoritmo *Soma_Dois_Numeros;*
Procedimento *soma (a, b: inteiro)*

Inicio
result ← a + b;
Fim
// Programa principal
Var
x,y, result: ***inteiro***
Inicio
escreva(“entre com dois números:”);
soma(x,y);
escreva(“Resultado:”, result)
Fim

1.7. MECANISMOS DE PASSAGEM DE PARÂMETROS EM SUB-ROTINAS

As **sub-rotinas** utilizam **parâmetros que são repassados durante a invocação desse sub-programa**. Quando o módulo principal chama uma **função** ou **procedimento**, ele **passa alguns valores chamados argumentos de entrada**.

Esses argumentos de entrada podem se dar segundo dois mecanismos distintos: **passagem por valor** (ou por cópia) ou **passagem por referência**

Passagem de Parâmetros por Valor (ou **Por Cópia**) – o **parâmetro é calculado** e **uma cópia de seu valor é fornecida ao parâmetro formal**, **no ato da invocação da sub-rotina**. A execução da **sub-rotina** prossegue normalmente e todas as modificações feitas no parâmetro formal **não afetam o parâmetro real**, pois **trabalha-se apenas com uma cópia dele**. Observe um exemplo de **passagem de parâmetro por valor**:

inteiro *funcao Soma(inteiro: a, inteiro: b)*

Os parâmetros **a** e **b** são **passados por valores do tipo inteiro**.

A função “soma” também pode ser escrita da seguinte maneira:

inteiro *funcao Soma(inteiro: a,b)*

Quando **mais de um valor é do mesmo tipo,** eles **podem ser separados por vírgulas**.

Passagem de Parâmetros por Referência – é realizada quando o **espaço de memória ocupado pelos parâmetros é compartilhado**. Nessa situação, as **eventuais modificações feitas afetam os parâmetros**. Uma mesma **sub-rotina** pode utilizar diferentes mecanismos de passagem de parâmetros e para diferenciar uns dos outros, **convencionou-se colocar a palavra reservada “var” antes da definição dos parâmetros passados por referência**. Observe um exemplo de **passagem de parâmetro por referência**:

inteiro *funcao Soma2(inteiro: var a, inteiro: b)*

Observe que **a** é um **parâmetro passado por referência** e **b** é um **parâmetro passado por valor**.

Significa que a **passagem por referência** irá **receber ou armazenar um endereço de memória relacionada a uma variável,** e **não o valor de uma variável**.

1.8. VARIÁVEIS GLOBAIS E LOCAIS

Com a utilização das sub-rotinas, surgiu o conceito de **variáveis globais** e **variáveis locais**. As **variáveis globais** são aquelas **declaradas no algoritmo principal e visíveis**, isto é, **podem ser usadas no algoritmo principal e por todas as demais sub-rotinas**. Já as **variáveis locais** são **definidas dentro de uma sub-rotina** e **somente visíveis nessa sub-rotina**, ou seja, sem acesso para demais partes do algoritmo.

Vale salientar que o **escopo das variáveis globais e locais** são a **região do programa onde a variável é reconhecida e pode ser utilizada**. As **variáveis locais** são aquelas que **só têm validade dentro da função/procedimento na qual foram declaradas**, já as **variáveis globais** têm **validades para todas as funções/procedimentos do programa principal**.

As **variáveis locais** são **passadas como parâmetros ou declaradas no início de cada função** e são **conhecidas como parâmetros formais**, pois **recebem cópias dos valores passados para a função,** e as **alterações no valor dos parâmetros não serão percebidas pelo programa que chamou a função**.

As **variáveis globais** são **declaradas antes de todas as funções do programa**.

VOCÊ SABIA?

As **variáveis globais** podem ser **alteradas por todas as funções do programa**. **Se uma função tem uma variável local com o mesmo nome de uma variável global**, **para aquela função a variável é local**. **Evitar ao máximo o uso de variáveis globais,** pois **elas ocupam a memória por mais tempo** e **tornam o programa mais difícil de ser compreendido**.

Observe a aplicação de variáveis locais e variáveis globais no exemplo a seguir:

```
Algoritmo exemplo_variáveis_locais_e_globais;
Var
X: real //variável global;
I: inteiro //variável global;
```

*funcao func(): **real**;*
Var
*X: **vetor** [1..5] **de inteiro** // **variável local***
*Y: **caractere**[10] // **variável local***
Inicio
<demais instruções>
Fim
Inicio
<demais instruções>
Fim

1.9. RECURSIVIDADE (RECORRÊNCIA)

É uma maneira de solucionar um problema por meio da **instanciação de instâncias menores do mesmo problema**, ou seja, **recursão** é uma **técnica em que uma função chama a si mesma**.

Também podemos dizer que **recursividade** é o mecanismo da programação no qual uma **definição de função refere-se a ela mesma**, assim **função recursiva** é uma **função que é definida em termos de si mesma**.

Recursividade é o mecanismo básico para **repetições nas linguagens de programação**.

As **funções recursivas** são em sua maioria soluções mais simples, se comparadas a funções tradicionais, já que **executam tarefas repetitivas sem utilizar nenhuma estrutura de repetição**, como for ou while.

Porém, vale ressaltar que essa simplicidade têm um preço que requer muita atenção em sua implementação. Trazendo a **recursividade** para o nosso cotidiano, um ótimo exemplo está na ação de contar um saco de moedas, em que cada ato de retirar uma moeda do saco precisaria de uma função “contar dinheiro” que corresponde a identificar qual é o valor da moeda e somar à quantia que ainda está no saco.

Para identificar a quantia que ainda está no saco basta chamar a mesma função “contar dinheiro” novamente, porém dessa vez deve ser considerado que a moeda tirada anteriormente não está mais lá. Este processo de retirar uma moeda, identificar seu valor e somar com o restante do saco se repete até que o saco esteja vazio. Quando o **ponto de parada for atingido** e a **função retornar o valor zero**, indica que **não há mais moedas a serem contados no saco**.

Existem vantagens e desvantagens quanto ao uso de **funções recursivas**, pois um programa recursivo é mais elegante e menor que a sua versão iterativa, além de exibir com maior clareza o processo utilizado, desde que o problema ou os dados sejam naturalmente definidos por meio de **recorrência**.

Por outro lado, **um programa recursivo exige mais espaço de memória** e é, na grande maioria dos casos, **mais lento do que a versão iterativa**.

RESUMO

Na nossa primeira aula, vimos alguns conceitos importantes de **Fundamentos de Programação**, dentre eles tratamos dos **algoritmos**, que são **sequências de passos finitos para executar determinada tarefa**. Os **algoritmos** possuem **duas principais formas de representação**, o **pseudocódigo,** que **representamos por meio do português estruturado,** e o **fluxograma,** que **utiliza uma forma visual para representar a sequência de execução**.

Vimos também que as **palavras reservadas** são **componentes de uma linguagem de programação** e que **não podem ser utilizadas como identificadores de variáveis e constantes**. Além disso, apresentamos os conceitos entre **compiladores,** que têm a **função de analisar todo o código a fim de traduzi-lo de uma vez**, e o **interpretador,** que tem a **função de conversão/análise do código fonte aos poucos**, ou seja, **sempre que uma instrução é executada**.

Quanto aos **tipos primitivos de dados**, aprendemos os **tipos número real e inteiro, lógico e texto**. Vimos a importância dessas estruturas como **entrada e saída para produção de um resultado específico**. Mostramos a forma de **declaração e inicialização** e o quanto isso é importante no mundo da linguagem de programação.

Vale ressaltar que as **variáveis e constantes** possuem regras específicas e que **não podemos nomeá-las de forma inadequada para evitar problemas na compilação ou interpretação**, a depender da linguagem de programação utilizada. Tratamos também de uma dica importantíssima, que é a **distinção entre caracteres maiúsculos e caracteres minúsculos,** conhecida por muitas linguagens de **case sensitive**.

Quanto aos **operadores**, aprendemos os **aritméticos,** que **definem as operações que podem ser realizadas sobre os números inteiros e reais**; os **relacionais,** que **são utilizados para comparar números e literais**, **retornando valores lógicos**; os **lógicos,** que **funcionam da mesma forma que os operadores da lógica matemática e utilizam dados lógicos**; os de **concatenação,** que **são utilizados para juntar dois valores ou variáveis do tipo texto;** e os de **atribuição,** que **são usados para definição de valor de uma variável**.

Tratamos também sobre o **controle de fluxo de programas e repetição** tão importantes para conhecermos um pouco mais sobre comandos que **podem interferir na sequência de instruções de um programa**. Aprendemos os comandos **Se...Fim_Se,** que é uma **estrutura condicional simples**, em que o que **é executado somente se a condição especificada retornar o valor verdadeiro** e, **caso seja falso**, **nenhuma das instruções constantes na estrutura da seleção será executada**.

Temos a **estrutura condicional composta** representada pelo comando **Se..Senão..Fim_Se,** que **trata de situações em que uma instrução ou conjunto de instruções é executado caso o teste condicional especificado retorne o valor verdadeiro** e, **caso o resultado do teste seja**

falso, **outra instrução ou um conjunto de instruções é executado**. Além disso, temos as **estruturas de decisão de múltipla escolha,** utilizando o comando **Caso,** que **utiliza avaliação de valores possíveis em uma variável**.

Dando continuidade, vimos as **estruturas de repetição** que proporcionam ao nosso algoritmo a **execução das tarefas repetidas de um determinado programa**. Essas estruturas tratam uma forma de **executar blocos de comandos somente sob determinadas condições**, mas com a opção de **repetir o mesmo bloco quantas vezes for necessário**. Conhecemos as estruturas **Para** (For), **Enquanto** (While) e **Faça-Enquanto** (do-While).

Vimos que o comando **Para** (for) **define uma estrutura** (instruções) **que será repetida por um número determinado de vezes**. A estrutura **Enquanto** (While), que se **caracteriza por uma verificação de encerramento de atividades antes de se iniciar a execução de seu bloco de instruções** e o comando **Faça-Enquanto** (Do-While), que funciona de forma similar ao comando enquanto, exceto pelo fato de que a **condição de controle só é testada após a execução do bloco de instruções e não antes**.

Entramos no universo das **sub-rotinas/módulos/subprogramas**, que se subdividem em **funções** que **retorna um só valor**, **a partir de um determinado conjunto de argumentos ou parâmetros** e os **procedimentos** que são **blocos de instruções que realizam tarefas específicas sem retorno de valor**.

Por fim, vimos o conceito e a aplicação de **recursividade** (**recorrência**) como mecanismo da programação, no qual **uma definição de função se refere a ela mesma**, como uma forma básica para **repetições nas linguagens de programação**. As **funções recursivas** são em sua maioria soluções mais simples, se comparadas a **funções** tradicionais, já que **executam tarefas repetitivas sem utilizar nenhuma estrutura de repetição**.