



O PODER DO FUTURO

Formação introdutória à tecnologia
para jovens mulheres negras



Introdução à Lógica





PARTE I – INTRODUÇÃO À LÓGICA

Introdução

Nem nos damos conta, mas quase todas as atividades que executamos no dia a dia demandam lógica. Vamos pensar no passo a passo de uma pessoa, do levantar até o café da manhã:

- Ao acordar, você levanta da cama;
- Desce as escadas;
- Entra na cozinha;
- Pega o pó de café no armário;
- Coloca na cafeteira;
- Joga água no compartimento específico;
- Após inserir todos os ingredientes na máquina, aperta o botão de ligar;
- Quando o café está pronto, pega a garrafa térmica;
- Despeja o café dentro de uma caneca;
- Bebe o café.

Esse passo a passo é uma sequência lógica. No dia a dia, não pensamos em nossas atividades dessa forma, mas, no que diz respeito à máquinas, precisamos ensinar detalhadamente por meio de sequências lógicas em códigos programados, já que máquinas não são capazes de definir comportamentos.

Lógica de programação é a forma de organizar uma sequência de instruções passadas para resolução de algum problema, até mesmo para criação de um *software*, uma aplicação ou site.

Em desenvolvimento web, lógica deve ser o primeiro conhecimento que devemos aprender, entender e praticar. É a partir da lógica que começamos a aprofundar em outros conhecimentos, como linguagens, *frameworks* e por aí vai.

Algoritmos

Qualquer tarefa executada por um computador/*smartphone* é baseada em algoritmos. Um algoritmo é uma sequência de instruções bem definidas, normalmente usadas para resolver problemas específicos, executar tarefas, realizar cálculos, equações, raciocínios, instruções e operações com um objetivo.

É necessário que os passos sejam finitos (com começo, meio e fim) e operados sistematicamente. Um algoritmo conta com entrada (*input*) e saída (*output*) de informações/ dados mediadas pelas instruções. Vamos usar uma calculadora estruturada em algoritmo para realizar um cálculo de multiplicação como exemplo:

```
Algoritmo Multiplicação de números positivos
Declaração de variáveis
numero1, numero2, resultado, contador: Inteiro
Início
ler(numero1)
ler(numero2)
resultado <- 0
contador <- 0
Enquanto contador < numero2 Faça
    resultado <- resultado + numero1
    contador <- contador + 1
Fim-Enquanto
escrever(resultado)
Fim
```

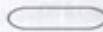


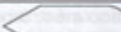




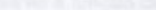
Usamos uma sequência lógica com entrada de dados iniciais (*input*), passos finitos que chegam aos dados finais (*output*) entregues ao usuário.

Diagrama de bloco e simbologia

Depois de definirmos nosso algoritmo, usamos o diagrama de bloco para estruturar a sequência lógica de forma padronizada, para que o algoritmo fique organizado e seja de fácil compreensão.

Quando pensamos em sistemas, o primeiro passo é a definição do algoritmo (análise) e a estruturação através dos diagramas de blocos. Representados por figuras geométricas, eles criam o fluxo que permite demonstrar a linha de raciocínio lógico utilizada pelo programador.

Abaixo, alguns dos símbolos que representam o diagrama de bloco, seus significados e a descrição de cada um:

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

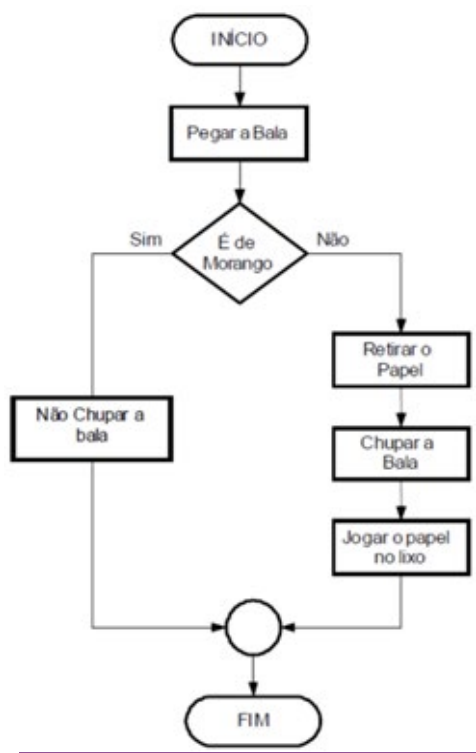
Vamos considerar a sequência lógica para chupar uma bala:

- Pegar a bala;
- Retirar o papel;
- Chupar a bala;
- Jogar fora o papel da bala;

Essa mesma sequência pode ser representada pelo diagrama de blocos, conforme abaixo.



Consideremos agora um diagrama de blocos com tomada de decisão. Nesse caso, a pessoa só vai chupar a bala se ela não for de morango, então usaremos blocos que permitam e direcionem as decisões.



Você deve estar se perguntando se irá usar isso em algum momento como programadora - a resposta é sim. Diagramas são usados no dia-a-dia por desenvolvedoras, feitos para facilitar e agilizar o processo de criação de *software*, uma vez que, com o fluxo desenhado, fica mais fácil passar para o código. Por isso a importância de um dos primeiros passos no desenvolvimento de um código ser a diagramação de blocos.

Variáveis

Variáveis são uma parte relevante e importante em lógica, já que possuem o poder de armazenar e manipular dados e informações inseridas em nossos códigos.

Cada variável é um espaço na memória do computador que trata e altera esses dados durante a execução de um programa.

Algumas formas de declarar uma variável incluem:

var pode ser usada em variável local ou global;

let é usada para declarar uma variável local;

const é uma variável que recebe um valor inalterável constante:

```
var numero = 34;  
  
let numero1 = 23;  
  
const numero2 = 54;
```

Constantes

Em linguagem de programação, constante é um valor que permanece inalterado durante todo o percurso do algoritmo ou processamento. Uma constante é igual em qualquer local ou trecho do código no qual for utilizada.

No cálculo de pi (π), por exemplo, sabemos que seu valor é 3,14, não existindo a possibilidade de ser outro número - trata-se portanto de um número constante. Na lógica de programação, uma constante mantém sempre o mesmo comportamento, e seu valor (independente de ser numérico ou não) é imutável.

Tipos de dados

Em dados primitivos, as variáveis podem receber, armazenar dados na memória e realizar o mapeamento de variáveis na memória. Alguns tipos de dados incluem:

- **Number** representa um conjunto de dados numéricos positivos ou negativos, inteiros, reais, ou decimais;
- **String** é a sequência de caracteres que representa um texto;
- **Boolean** é uma variável que recebe um dado lógico com apenas duas possibilidades: ser verdadeiro ou falso (*true or false*);
- **Null** representa uma variável vazia ou nula que não armazena nada. Variáveis *null* são iniciadas como '0' ou são nulas;
- **Lógicos** formam um grupo de dados para representar dois únicos valores lógicos possíveis: **verdadeiro** ou **falso**.

É comum encontrarmos em outras referências outros grupos de pares de valores lógicos, como **sim/não**, **1/0** ou **true/false**.

Operações lógicas

Operações lógicas são responsáveis por resultados *booleanos* (lógicos) que sempre dirão se a operação é falsa ou verdadeira - neste caso, as mesmas retornarão um valor *booleano*. Abaixo, operadores lógicos:

&& = e
|| = ou
! = não

Em operações com **&&** (e), o resultado só será verdadeiro (**true**) se ambas as condições forem verdadeiras; em operações **||** (ou), apenas uma precisa ser verdadeira para que retorne verdadeiro (*true*); em operações com **!** (não), a condição é negada, ou seja, ao colocarmos uma verdadeira precedida de **!**, ela passa a ser falsa. Se for falsa e precedida por **!**, passa a ser verdadeira.

Abaixo, uma tabela da verdade tanto para **&&** quanto **||** que auxilia no entendimento.

- **&& e** - Se for de noite **e** estiver calor, vou beber um suco (**true**);

A	B	A && B
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	falso
falso	verdadeiro	falso
falso	falso	falso

- **|| ou** - de for de noite **ou** estiver calor, vou beber um suco (**true**);

A	B	A B
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	verdadeiro
falso	verdadeiro	verdadeiro
falso	falso	falso

Estrutura de Decisão

Quando precisamos tomar uma decisão em nosso código baseada em uma condição que retorne um resultado **verdadeiro** ou **falso**, usamos expressões condicionais. Estruturas condicionais são representadas por **if** (se), **else** (se não) ou **else if**.

No exemplo a seguir, declaramos uma variável e lhe atribuímos uma idade. Ao usarmos o condicional **if**, verificamos se a idade é maior que 18; se for, ela retornará que a pessoa é obrigada a votar.

```
var idade = 18;

if(idade > 15 && idade < 18 || idade > 70) {
    console.log('O seu voto é opcional')
}
```

É possível haver mais de uma condição em uma única verificação. No exemplo a seguir, verificaremos se a idade tem alguma das condições usando o operadores lógicos **&&** (e) e **||** (ou) em “menor de 16 anos”, “menor que 18”, ou “maior que 70.”

```
var idade = 18;

if(idade > 15 && idade > 18 || idade > 70) {
    console.log('O seu voto é opcional')
}
```

Acrescentamos o **else if** caso a idade seja menor que a idade permitida para votar.

```
var idade = 18;

if(idade > 15 && idade < 18 || idade > 70) {
    console.log('O seu voto é opcional')
} else if (idade < 16) {
    console.log('Você não pode votar')
}
```

Para finalizar, vamos usar o **else** sozinho, caso nenhuma das condições seja verdadeira.

```
var idade = 18;

if(idade > 15 && idade < 18 || idade > 70) {
  console.log('O seu voto é opcional, mas é importante')
} else if (idade < 16) {
  console.log('Você não pode votar')
} else{
  console.log('Você é obrigado a votar, com consciência')
}
```

Outra estrutura condicional é o **switch case**, de comportamento semelhante ao **if** e **else** porém mais organizado e de fácil compreensão. O **switch case** só recebe valores pré-definidos e não-condições que retornem valores verdadeiros ou falsos.

```
let animal = 'Cachorro';

switch(animal){
  case 'Cachorro':
    console.log('mamifero');
    break;
  case 'Galinha':
    console.log('oviparo');
    break;
  case 'Grilo':
    console.log('onivoro');
    break;
  default:
    console.log('Animal não identificado')
    break
}
```

No código acima, o **case** representa uma comparação entre a condição e a variável declarada. Caso a condição esteja correta, o comando **break** encerrará as verificações no laço disponível e ignorará os laços restantes. Por último, temos o **default**, que significa que nenhum dos casos corresponde à verificação.

Repetição

Estruturas de repetições repetem determinado bloco de comandos enquanto a condição atende ao requisito. As estruturas de repetição são representadas por **while**, **do while** e **for**.

Vamos declarar uma variável e iniciá-la em 0 para, em seguida, usar o **while** e passar a instrução que, enquanto nossa variável for menor que 11, a multiplicaremos por 5 (resultando na tabuada do 5).

```
let i = 0;
while(i < 11){
  console.log('5 x ' + i + ' = ' + 5*i)
  i++
}
```

O resultado ao executar esse código será:

```
5 x 0 = 0
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Do while é parecido com **while**, mas nele, a condição só acontece depois que os comandos do bloco forem executados. Abaixo, declaramos uma variável iniciada com 0. Enquanto for menor que 5 (condição passada no bloco do **while**), ela passará novamente pelo bloco **do** e acrescentará um número inicial de variável.

```
let contador = 0;

do{
  console.log("O contador vale: " + contador);
  contador++;
}while(contador < 5)
```

O resultado:

```
O Contador vale: 0
O Contador vale: 1
O Contador vale: 2
O Contador vale: 3
O Contador vale: 4
```

Embora siga o mesmo princípio do **while**, o **for** é utilizado quando temos definida a quantidade de iterações de repetições necessárias. Por parâmetro, O **for** recebe três atributos: o primeiro é uma variável, que inicia a nossa condição; o segundo, a verificação da condição; o último, o que ele deve fazer caso a condição seja verdadeira.

Dentro do nosso **for**, iniciamos a variável **i** recebendo o valor 0. Enquanto for menor que 11, nossa variável acrescentará mais um número e realizará o cálculo passado em nosso `console.log (2*i)`.

```
for(let i = 0; i < 11; i++){
  console.log("2 x " + i + " = " + 2*i);
}
```

O resultado:

$$\begin{aligned}
 2 \times 0 &= 0 \\
 2 \times 1 &= 2 \\
 2 \times 2 &= 4 \\
 2 \times 3 &= 6 \\
 2 \times 4 &= 8 \\
 2 \times 5 &= 10 \\
 2 \times 6 &= 12 \\
 2 \times 7 &= 14 \\
 2 \times 8 &= 16 \\
 2 \times 9 &= 18 \\
 2 \times 10 &= 20
 \end{aligned}$$

Tabela da verdade

A tabela da verdade é muito utilizada na lógica de programação para desenvolvimento do raciocínio lógico. Seu objetivo é verificar a validade lógica de uma condição composta (argumento formado por duas ou mais condições simples).

Para utilizar a tabela da verdade, precisamos primeiro conhecer os símbolos utilizados na lógica:

Símbolo	Operação	Descrição	Exemplo
p		Proposição 1	p= Amanda é alta.
q		Proposição 2	q= Lívia é baixa.
~	Negação	não	Se Amanda é alta, " $\sim p$ " é FALSO.
^	Conjunção	e	$p \wedge q$ = Amanda é alta e Lívia é baixa.
v	Disjunção	ou	$p \vee q$ = Amanda é alta ou Lívia é baixa.
→	Condicional	se... então	$p \rightarrow q$ = Se Amanda é alta então Lívia é baixa.
↔	Bicondicional	se e somente se	$p \leftrightarrow q$ = Amanda é alta se e somente se Lívia é baixa.

Agora, veremos o uso da tabela e seus símbolos:

- **Negação** é representada pelo sinal \sim . A operação lógica da negação é a mais simples, e muitas vezes não demanda uso da tabela da verdade. No mesmo exemplo da tabela anterior, se Amanda é alta (**p**), dizer que Amanda não é alta ($\sim p$) é FALSO e vice-versa:

p	$\sim p$
V	F
F	V

- **Conjunção** é simbolizada pelo símbolo \wedge ou **e**. No exemplo abaixo, a afirmação “Amanda é alta e Livia é baixa” é simbolizada por “ **$p \wedge q$** ”.

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

Na conjunção, é necessário que todas as informações sejam verdadeiras para que o resultado da proposição composta seja VERDADEIRO.

- **Disjunção** é simbolizada por **v** representando o **ou**. Ao trocarmos o conectivo do exemplo acima para **ou**, temos “Amanda é alta ou Livia é baixa”. Nesse caso, a frase é simbolizada por “**p v q**”.

p	q	p v q
v	v	v
v	f	v
f	v	v
f	f	f

Na disjunção, basta que uma das condições seja verdadeira para que o resultado também seja.

- **Condicional** é simbolizada por \rightarrow e representada pelos conectivos **se** e **então**, que interligam as condições simples em uma relação de causalidade. No exemplo “Se Rodolfo é mineiro, então ele é brasileiro” a condicional se torna “**p \rightarrow q**”.

p	q	p \rightarrow q
v	v	v
v	f	f
f	v	v
f	f	v

As condições compostas condicionais (contendo os conectivos **se** e **então**) só serão falsas se a primeira proposição for verdadeira e a segunda falsa.

- **Bicondicional** é simbolizada por \leftrightarrow e lida através dos conectivos **se e somente se**, que interligam as condições simples em uma relação de equivalência. No exemplo "Amanda fica feliz **se e somente se** Lívia sorri", **se** vira " $p \leftrightarrow q$ ".

p	q	$p \leftrightarrow q$
V	V	V
V	F	F
F	V	F
F	F	V

As bicondicionais sugerem uma ideia de interdependência: Como o próprio nome diz, a bicondicional é composta por duas condicionais, uma que parte de **p** para **q** ($p \rightarrow q$) e outra no sentido contrário ($q \rightarrow p$). As condições bicondicionais só serão verdadeiras quando todas forem verdadeiras ou todas falsas.

FICHA TÉCNICA

Equipe Olabi: Gabriela Agustini, Silvana Bahia, Aldren Flores, Davi Arloy, Joyce Santos, Amanda Oliveira, Roberta Hércias, Clara Queiroz e Rodrigo Schmitt

Gestão do Projeto: Aldren Flores

Assistente do Projeto: Joyce Santos

Facilitadora: Vera Félix

Comunicação: Raiz Digital

Social media: Raiz Digital e Amanda Oliveira

Coordenadora Técnica: Amanda Silva

Percurso metodológico: Amanda Silva

Conteúdo metodológico das apostilas: Amanda Silva

Revisão gramatical: Luciana Moletta

Design: Bruna Martins

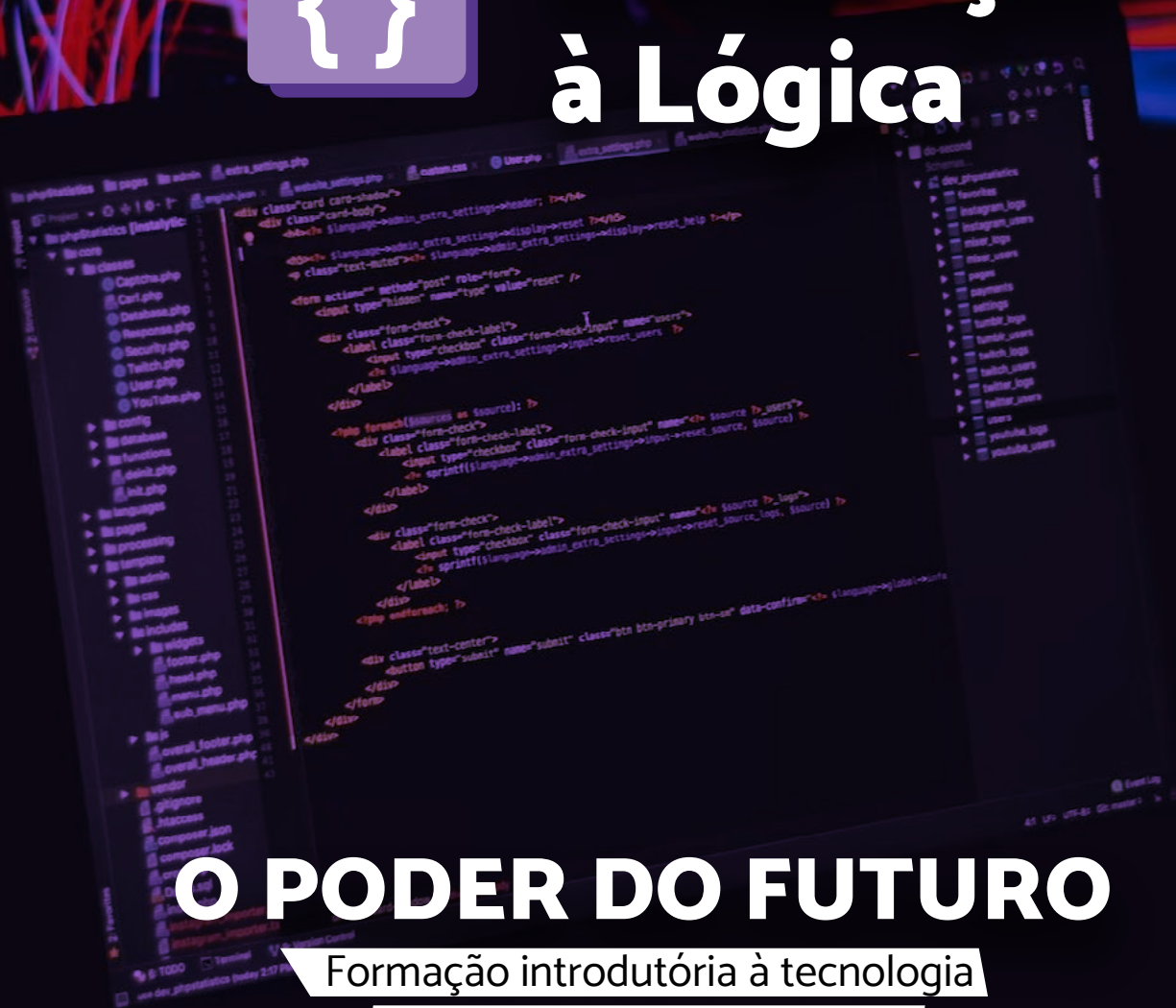
Professoras: Amanda Silva, Letícia Furtado, Lisandra Souza, Mônica Santana, Renata Nunes e Simara Conceição

Monitoras: Ana Beatriz dos Santos, Angela Karolina Lopo e Renata Silva

Apoio: Disney



Introdução à Lógica



O PODER DO FUTURO

Formação introdutória à tecnologia
para jovens mulheres negras

