

Lógica de programação

Kayky Moreira Praxedes

Fevereiro 2026

1 O que é lógica de programação

Trata-se do estudo das **estruturas lógicas** presentes na maioria das linguagens de programação.

Muitos elementos básicos são comuns à maioria das linguagens de programação. O entendimento dos pilares e regras presentes à generalidade das linguagens auxilia de maneira categórica na aprendizagem de uma nova, visto que (**a base normalmente é a mesma**), sendo as modificações, na maioria dos casos, apenas na sintaxe, nas bibliotecas e em algumas outras poucas peculiaridades.

Dessa maneira, ao invés de se reestudar todos os paradigmas e estruturas básicas de programação ao aprender uma nova linguagem, tendo uma boa noção de lógica de programação, pode-se apenas **familiarizar-se com a nova sintaxe e estudar suas individualidades** (processo extremamente mais rápido e eficiente).

2 O que é um programa

Algoritmos:

São uma **sequência lógica de passos necessária para resolver um problema** (não necessariamente linear, podendo ter bifurcações ou *loops* para alterar seu fluxo).

O algoritmo pode ser apenas um resumo informal, como também pode ser um pseudo-código detalhado ou um fluxograma indicando cada ação e diferenciando cada tipo de tomada de decisão.

O programa é a **implementação desse algoritmo** pelo computador através de uma linguagem de programação.

3 Como o programa é executado

As linguagens de programação podem ser divididas em:

- **Linguagens de baixo nível:** Também chamada de linguagem de máquina, trata-se da forma como o processador (*CPU*) entende as instruções. A informação é passada em **código binário**, com cada instrução tendo 32 ou 64 bits nos computadores modernos, a depender da arquitetura [01].

Como código binário é ilegível, existe o *Assembly*, que é a **representação direta das instruções em binário para texto**, com algumas abstrações como uso de *labels*, etc. (o que permite um entendimento infinitamente mais claro e uma depuração e manutenção mais precisa e menos trabalhosa).

- **Linguagens de médio nível:** Tratam-se de linguagens que ainda tendem recursos próximos da máquina, como *gerenciamento manual de endereços de memória*, *contato com registradores*, etc. Todavia, elas possuem mais **abstrações** (funções, tipos e *loops* por exemplo), permitindo a construção de códigos de maior complexidade técnica mais facilmente e em menos tempo.
- **Linguagens de alto nível:** Tratam-se de linguagens cuja sintaxe e estrutura se aproximam mais da **linguagem humana** (maior abstração), sem gerenciamento manual de endereços de memória ou tratamento de registradores, esses sendo feitos e reorganizados *automaticamente pela linguagem*. Em muitos casos essas linguagens trabalham com *Programação Orientada a Objetos (POO)*.

Informações para a *CPU*:

Como já foi explicado, o processador entende apenas código binário, sendo necessário uma **conversão dos códigos** para sua execução pela *CPU*, que pode ser **direta** (compilação), ou **indireta** (interpretação).

- **Compilação:** Os elementos da linguagem já possuem uma **tradução direta bem definida para linguagem de máquina**, podendo, depois de convertidas através de um compilador, serem executados pela *CPU*. Esse fenômeno ocorre principalmente (não exclusivamente) em linguagens de médio nível, por possuírem tipos definidos e estáticos, facilitando a tradução. Linguagens compiladas (C, C++, Rust, Go, etc.) tendem a ser mais rápidas, justamente por serem mais diretas.

Código → Compilador → Binário → CPU

- **Interpretação:** A linguagem **não pode ser compilada diretamente**, sendo necessário um passo intermediário de **interpretação** [02]. Comumente linguagens interpretadas são de alto nível, especialmente quando essa é **Orientada a Objetos**. Esse fenômeno se deve à variabilidade de tipos devido a possibilidade de criação de novos objetos, bem como sua modificação, possuir muitas abstrações, etc. Linguagens interpretadas (Java, C#, Python, etc.) tendem a ser mais lentas, justamente pela interpretação, embora as linguagens modernas possuam meios para contornar essa dificuldade.

Código → Tradutor → Interpretador → CPU

4 Armazenamento de informações

Para que o computador seja capaz de realizar ações, tanto as instruções quanto as informações intermediárias precisam ser guardadas em algum local.

O armazenamento das informações é feito através de **variáveis**. Essas nada mais são do que referência à um **modelo de memória definido pela linguagem** (objetos, registros, rótulos para espaços de memória, etc.).

A retenção da informação é feita pela operação de **atribuição** [03].

Variável = Informação

A informação pode ser tanto de um dado estático (informado diretamente pelo programador), quanto o resultado de uma operação, a informação contida em outra variável, um endereço, etc.

5 Tipos de dados

As **informações armazenadas nas variáveis podem assumir diferentes naturezas**, chamadas de tipos de dados.

Comumente as linguagens de programação possuem alguns tipos básicos de informação [04], sendo esses:

- **Texto:** Podem ser apenas caracteres (*char*) ou então textos longos (*strings*).
- **Numérico:** Podem ser números inteiros (*int*) ou racionais (*float* ou *double*, o segundo podendo representar números maiores e com maior precisão, mas ocupando mais memória), *long* (aumenta o espaço das variáveis), etc.
- **Booleano:** São valores para o representar verdadeiro ou falso.

6 Manipulação das informações:

As informações contidas nas variáveis podem ser manipuladas através de operações pré-definidas na linguagem. Por meio desses elementos básicos, podem ser montadas estruturas mais complexas que permitem uma manipulação mais específica desses valores (montando funções e bibliotecas).

As operações elementares são basicamente **aritméticas, comparativas** e de **atribuição**. [05].

7 Elementos de tomada de decisão

Tratam-se de ferramentas que permitem a **execução de instruções específicas à depender da ocorrência ou não de uma condição** previamente definida ou o redirecionamento do código livremente. Após a execução da decisão, o fluxo do programa continua normalmente.

if/else:

Trata-se de uma estrutura binária, onde, definida uma condição no *if*, se ela ocorrer, **será tomada uma ação, caso contrário, será executado o próximo teste** (ou é retomado o fluxo normal do código na ausência de um *else*).

A maioria das linguagens permite uma condição intermediária (sua sintaxe varia mas é algo próximo de *elif*), onde, negada a condição anterior, é testada a essa hipótese, sem limites para seu uso, permitindo uma estrutura *if/else* com inúmeros testes.

Switch case:

Dada uma expressão cujo valor é comparado com diferentes casos, são definidas *n* **condições, cada qual irá executar uma instrução diferente**.

go to:

Diferentemente das estruturas anteriores que alteram o fluxo do programa a partir de uma condição, essa instrução é uma ferramenta de **redirecionamento incondicional**, podendo levar para qualquer lugar do código (é mal visto no contexto da programação e sua utilização é desencorajada, pois polui o código e dificulta acompanhar o caminho das instruções e, por consequência, o *debugging*).

8 Elementos de repetição (*loop*)

Ferramentas que permitem a **repetição de instruções a depender de uma condição**.

while:

Realiza o **loop até que uma condição seja satisfeita** (não realizando nenhuma vez caso ela seja inicialmente falsa), a administração do alcance dessa condição ficando a cargo do programados.

for:

A **administração das condições de alcance da condição do loop são definidas no escopo**, onde o *for* permite definir explicitamente inicialização, condição de continuidade e atualização (não ocorre no *while*).

Algumas linguagens permitem o **uso do *for* de maneiras mais flexíveis**, como o Java que permite uma iteração sobre todos os elementos de uma coleção (*for(tipo nome in conjunto){}*).

break e continue:

Ferramentas para controlar ações dentro do *loop*. O *break* sai da repetição instantaneamente e o *continue* faz com que o resto das instruções do *loop* a partir dele sejam ignorados, partindo para a próxima repetição.

Caso a condição do *loop* não seja alcançada, a **repetição seguirá infinitamente**. Normalmente as *IDEs* tem um recurso de proteção que identifica esses casos e interrompe a execução do código.

9 Funções

São estruturas compostas por um conjunto de instruções, permitindo ao computador executar ações definidas pelo programador. Podem ser pré-definidas por *bibliotecas* (normalmente propósitos mais gerais) ou implementadas pelo próprio programador (para atender a propósitos específicos).

Boas práticas de programação recomendam que **uma função execute uma tarefa bem definida**, delegando etapas intermediárias a sub-funções. Isso contribui para um código mais organizado, legível, reutilizável e de fácil manutenção.

Uma função pode admitir argumentos, retornar valores e/ou modificar variáveis, embora **nenhum desses elementos seja obrigatório**:

- **Argumentos (*input*)**: Entradas que influenciam no comportamento da função.
- **Valores de retorno (*output*)**: Resultados produzidos e enviados pela função ao término de sua execução.

Comentários:

Possuem utilidade explicativa (**não realizam nenhuma alteração direta ou indireta no código**). São muito úteis para explicar o funcionamento de operações mais complexas.

```
def soma():
    return 2 + 5

x = soma() # Variável "x" recebe o valor de retorno da função "soma"
print(x)   # Função "print" com o argumento "x"
```

10 Conjuntos de elementos

A organização de elementos em grupos ao invés de tratar variáveis individualmente permite uma facilidade de trabalho muito grande, bem como um código mais limpo e organizado. Os conjuntos podem ser principalmente de dois tipos:

Tamanho pré-definido:

- **Vetores/Arrays**: Estruturas lineares e ordenadas que **armazenam elementos de mesmo tipo** (qualquer um dos especificados anteriormente ou tipos novos criados pelo programador), geralmente acessados por índices numéricos, seu início na posição 0 (“vetor”[0]).

Seu tamanho é fixo, e inserções ou remoções exigem **reorganização manual** dos elementos (itens removidos deixam um espaço vazio em sua posição).

- **Matrizes: Conjuntos de vetores de mesmo comprimento**, permitindo a organização de dados em duas dimensões (início em “matriz[0][0]”).

Uma matriz de dimensão n pode ser entendida como um conjunto de matrizes de dimensão $n - 1$ (normalmente não passam de 3 dimensões pela complexidade aumentar geometricamente).

Tamanho variável:

O tratamento dessas estruturas varia muito conforme a linguagem. Em C, são implementadas por meio de alocação dinâmica e arranjos encadeados. Linguagens de mais alto nível oferecem abstrações que permitem inserção e remoção de elementos com gerenciamento automático, como listas dinâmicas, filas e coleções baseadas em chaves.

11 Programação Orientada a Objeto (POO)

Trata-se de um paradigma que organiza o código em objetos. Esses muitas vezes representam **elementos do mundo real cotidiano**, possuindo atributos, comportamentos, derivações, etc. (nível maior de proximidade e abstração do humano que da máquina).

Diferente das linguagens procedurais, que estruturam o programa em funções sequenciais que manipulam dados separadamente, a POO **integra dados e funcionalidades em unidades coesas (objetos)**, promovendo maior modularidade, reuso e manutenção do código através de modelos mais intuitivos e próximos da realidade.

Seus pilares são: **encapsulamento, herança, polimorfismo e abstração**.

Encapsulamento:

Promove **ocultamento e proteção de dados de acessos e modificações**.

Esse objetivo é alcançado tornando elementos do código inacessíveis diretamente, privando completamente seu acesso fora dos métodos originais ou permitindo uma interação restrita (classes e variáveis *private* ou *protected*), dependendo muitas vezes de métodos públicos com regras e capacidades limitadas para qualquer interação, permitindo controle de efeitos colaterais (*getters*, *setters*, métodos públicos, etc.).

Herança:

Permite reutilização de código por meio da **derivação de características de uma classe para a outra**.

Através de uma hierarquia, a **superclasse** deriva **atributos e métodos para uma subclasse**, esta podendo adicionar novos comportamentos ou sobreescriver os existentes.

Um exemplo é pensar numa *superclasse veículo*, com as *subclasses carro, avião, trem*, etc.

Polimorfismo:

Permite que um mesmo método ou operação se **comporte de maneira diferente dependendo do objeto que o invoca**.

Essa diferença de tratamento pode ocorrer tanto por **sobrecarga** (várias versões de um método com parâmetros diferentes em uma mesma classe) quanto pela **sobrescrita** (redefinição de um método).

Abstração:

Tem o objetivo de tornar o código menos complexo, **definindo comportamentos essenciais de classes e ocultando detalhes de implementação**.

Um exemplo é o método *cálculo de área* para uma classe *polinômio*. Como toda subclasse vai ter um cálculo de área diferente, não há necessidade que o método da superclasse seja específico (esse sendo abstrato), apenas **indicando** que as subclasses terão uma variação desse método.

Apêndice A

Como o programa executa

[01] - Existem diferentes arquiteturas além das computacionais modernas, atendendo a diferentes propósitos, sejam elas arquiteturas mais antigas, ou que tem propósitos mais simples, não necessitando de tanta memória, tais quais processadores *nRisc* com 8 ou 16 bits.

[02] - Existem principalmente 2 tipos de fluxo de informação durante a interpretação:

1. **Não otimizado:** As instruções são traduzidas para *Bytecode* (linguagem pela qual podem ser entendidas *Virtual Machine*), depois as instruções traduzidas são interpretadas uma a uma. A *VM* irá compilar as instruções para código de máquina (processo bem mais demorado, visto que instruções iguais tem de ser re-interpretadas):

Código → Tradutor → Interpretador (Binário) → CPU

Alguns exemplos de linguagens que seguem essa estrutura são Python (através do CPython), Ruby (através do YARV), etc.

2. **Otimizado:** A *VM* não apenas interpreta, como também identifica trechos repetidos do código. Essas instruções são enviados para um passo intermediário *Just In Time* (JIT), esses sendo compilados diretamente para código de máquina (não sendo necessária uma interpretação redundante, tornando o processo quase tão rápido quanto uma compilação direta):

Código → Tradutor → Interpretador → JIT → Binário → CPU

Algumas linguagens que seguem essa estrutura são Java (através da JVM), C# (através da CLR), etc.

Armazenamento de informações

[03] - Essa estrutura pode variar em algumas linguagens, principalmente em Assembly (linguagem de máquina), onde a atribuição depende de um registrador sendo uma variável (mais à esquerda), e registradores à direita (podendo ser 1 ou 2 a depender da operação), sendo as informações a serem processadas pela:

operação (instrução a ser seguida) Registrador 1 Registrador 2 Registrador 3

Tipo de dados

[04] - Os tipos de dados são os elementos que tendem a mudar mais em linguagens de programação. Em C, por exemplo, *strings* são apenas *arrays* de *char*, mas outras linguagens, como Java, os tratam como elementos de natureza diferente e não se misturam. Algumas linguagens como Python não possuem elementos *float*, sendo *double* por padrão. Em C não existem elementos do tipo *boolean* (sendo 0 a representação de um valor falso e qualquer outro valor verdadeiro), mas em outras linguagens eles são objetos bem definidos, etc.

Manipulação de informações

[05] - As linguagens variam muito em suas operações (em grande parte por sua utilidade), sendo comum em C, por exemplo, operações booleanas bit a bit (ferramenta que não é comum a muitas outras linguagens).

Além disso, vale frisar que em linguagens de médio e baixo nível, todas as informações são numéricas (caracteres por exemplo são códigos numéricos da tabela *ASCII*, com um identificador de tipo *char*), sendo elementos que podem ser manipulados operações aritméticas ($a + b = \tilde{A}$, já que em *ASCII* $a = 97$, $b = 98$ e $\tilde{A} = 195$).

Fontes

1. DEITEL, Harvey M.; DEITEL, Paul J. Como programar em C. 2. ed. Rio de Janeiro: LTC, 1994.
2. DEITEL, Harvey M.; DEITEL, Paul J. Java: como programar. 10. ed. São Paulo: Pearson, 2017.
3. HARVARDX. CS50's introduction to programming with Python. 2022. Disponível em:
<https://learning.edx.org/course/course-v1:HarvardX+CS50P+Python/home>. Acesso em: 26 jan. 2026.
4. PEIXOTO, Daniela Cristina Cascini. Disciplina: Lógica de programação. Curso de graduação em Engenharia de Computação – Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG), 2024.
5. CAMPOS, Luciana Maria de Assis. Disciplina: Programação orientada a objetos. Curso de graduação em Engenharia de Computação – CEFET-MG, 2024.