Lógica de programação

Kayky Moreira Praxedes

Janeiro 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 de programação, ao se ter uma boa noção de *lógica de programação*, pode-se apenas familiarizar-se com a nova linguagem 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. As instruções seguem uma ordem linear, salvo quando estruturas alteram esse fluxo (bifurcações ou loops).

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 executa

As linguagens de programação podem ser divididas em:

- **Linguagens 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 apenas a representação direta das instruções em binário para texto (o que permite um entendimento infinitamente mais claro e uma depuração e manutenção mais precisa e menos trabalhosa).

- **Linguagens de baixo nível:** Tratam-se de linguagens que ainda tendo recursos próximos da máquina, como *gerenciamento manual de endereços de memória*, *contato com registradores*, etc. Todavia, elas possuem funções mais complexas definidas, permitindo a construção de códigos de maior complexidade técnica mais facilmente e em menos tempo, possuindo pouca diferença no tempo de execução em relação à linguagem de máquina.

Os códigos passam por um *compilador* que já conhece as funções, por onde essas são traduzidas para código de máquina e executados pela CPU, de modo que o fluxo é dado por:

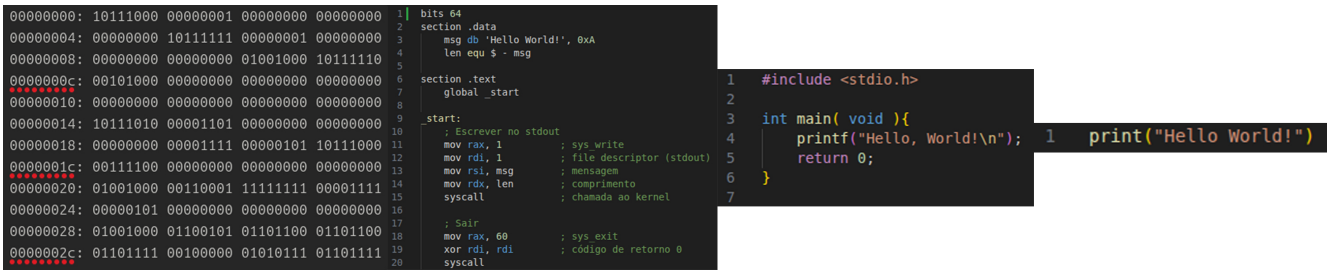
$$\text{Código} \rightarrow \text{Compilador} \rightarrow \text{Binário} \rightarrow \text{CPU}$$

Exemplos desse tipo de linguagem são C, C++, etc.

- **Linguagens de alto nível:** Tratam-se de linguagens cuja sintaxe e estrutura se aproximam mais da linguagem humana, sem gerenciamento manual de endereços de memória ou tratamento de registradores, esses sendo feitos e reorganizados *automaticamente*. Ela possui *tipos abstratos prontos* e normalmente trabalham com *Programação Orientada a Objetos* (POO).

  Nesses tipos de linguagem, geralmente **a informação não é compilada diretamente para binário**, sendo necessário que a informação seja traduzida e passe por um *interpretador* primeiro[02]:

$$\text{Código} \rightarrow \text{Tradutor} \rightarrow \text{Interpretador} \rightarrow \text{CPU}$$



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 rótulos para *espaços de memória* (que como já comentados, são acessíveis diretamente apenas por linguagens de baixo nível).

A retenção da informação é feita por meio da atribuição (=), com a variável à esquerda e a informação de interesse à direita [03].

$$\text{Variável} = \text{Informação}$$

A informação pode ser tanto de um dado estático (como o resultado de uma operação), quanto 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)
- **Booleano:** São valores para o representar verdadeiro (*True*) ou falso (*False*).

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**, sendo elas soma ($+$), subtração ($-$), multiplicação ($*$), divisão ($/$) e cálculo do resto de uma divisão (módulo) ($\%$), com algumas exceções, sendo válidas apenas para elementos numéricos[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 *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 condição inicial que pode assumir múltiplos valores, 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, pois polui o código e dificulta acompanhar o caminho das instruções e, por consequência, o debbugin).

8 Elementos de repetição (loop)

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

while e *do while*:

Realizam o loop até que uma condição seja alcançada (o while realizando a primeira operação depois de entrar no loop, e o do while realizando-a antes de entrar).

for

Define-se uma variável a ser usada no loop, de modo que as instruções continuarão se repetindo até que essa variável atinja um valor (esse valor delimitado por uma variável ou um outro valor pré-estabelecido). A estrutura também define explicitamente um padrão de crescimento ou decrescimento da variável (não ocorre no while).

break e *continue*:

Ferramentas para controlar ações dentro do loop. O *break* sai da repetição instantaneamente e o *continue* faz com que o resto do loop a partir da instrução seja ignorada, partindo para a próxima repetição.

Caso a condição do loop não seja alcançada, a repetição seguirá infinitamente. Normalmente as linguagens 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 ação direta ou indireta), sendo muito úteis para indicar o funcionamento de funções mais complexas, onde serão utilizadas variáveis, etc.

```
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

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 (geralmente iniciando em 0, ou seja *"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).

Conjuntos de tamanho variável:

O tratamento dessas estruturas varia 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* que representam elementos do mundo real cotidiano, esses 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 (*private* ou *protected*), dependendo de métodos públicos com regras e capacidades limitadas para qualquer interação, permitindo controle de efeitos colaterais (*getters* e *setters*).

```
1   class ContaBacaria {
2       private saldo()
3       public deposito()
4       public saque()
5   }
```

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*, essa podendo adicionar novos comportamentos ou sobrescrever 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, sem explicitar esse comportamento para evitar sobrescritas e erros.

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 especifico (esse sendo abstrato), esse 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] - Como **não existe uma passagem direta da informação para a CPU**, sendo necessário um processo de tradução e interpretação, os fluxos podem ser de dois tipos:

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 CPyton), Ruby (através do YARV), etc.

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

   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. Algumas linguagens como Python não possuem elementos *float*, sendo *Double* por padrão. Em C não existem elementos do tipo *Boolean* (sendo 0 igual a *False* e qualquer outro valor *True*), mas em outras linguagens eles são Objetos bem definidos, etc.

Manipulação de informações

[05] - Em linguagens de baixo nível e máquina, todas as informações são numéricas (caracteres por exemplo são códigos numéricos da tabela *ASCII*, com um identificador desse tipo de elemento), sendo elementos que podem ser manipulados por todas as operações aritméticas (a + b = Ã, já que em *ASCII* a= 97, b= 98 e Ã= 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.