UNIVERSIDADE FEDERAL DO CEARÁ - UFC CAMPUS CRATEÚS

CURSO DE SISTEMAS DE INFORMAÇÃO

Disciplina: CRT0031 - Linguagens de Programação

Professor: BRUNO DE CASTRO HONORATO SILVA

Semestre: 2025.1

Trabalho de Fundamentos de Linguagens de Programação

Crateús

2025

Desafios

1. Introdução às Linguagens de Programação

EVOLUÇÃO DAS PRINCIPAIS LINGUAGENS DE PROGRAMAÇÃO

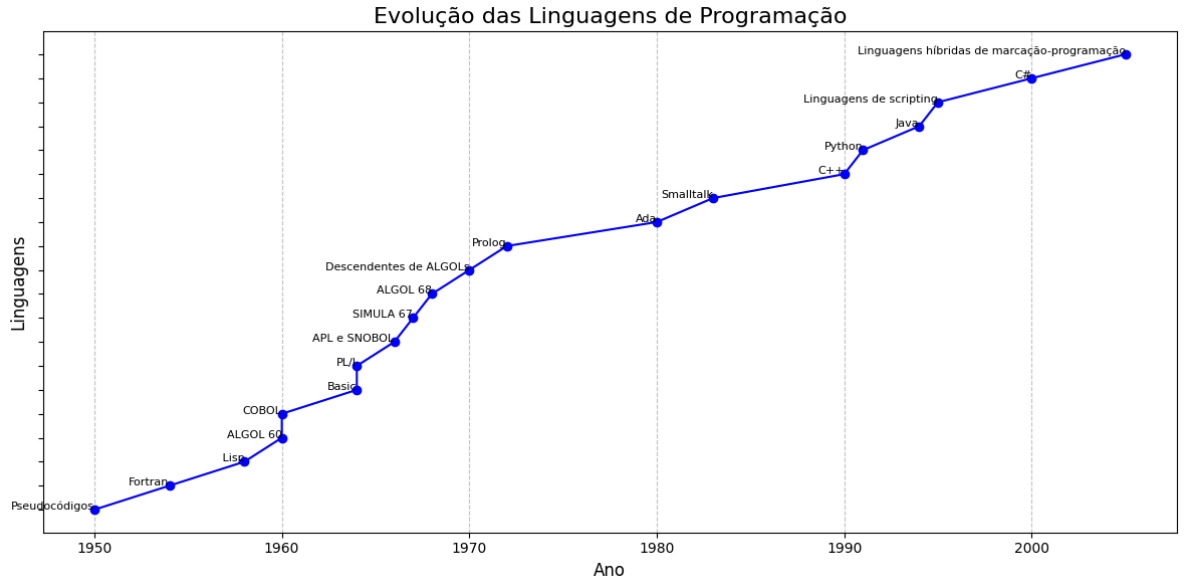


Gráfico plotado em Python, referência **: Conceitos de Linguagens de Programação de Robert W. Sebesta (cap. 02)**

O gráfico gerado em Python com a biblioteca *matplotlib* ilustra a evolução das linguagens de programação desde a década de 1950 até os dias atuais. Observa-se uma transição de representações rudimentares, como pseudocódigos, para linguagens cada vez mais compreensíveis e próximas da linguagem humana. É importante ressaltar que o gráfico não contempla todas as linguagens existentes nem suas versões posteriores, limitando-se a indicar seus marcos iniciais.

Embora não representada na imagem, vale destacar a linguagem *Plankalkül*, desenvolvida por Konrad Zuse entre 1936 a 1945. Apesar de não implementada na prática, sua sofisticação contribuiu para o entendimento da evolução sintático-semântica das linguagens.

Os pseudocódigos da década de 1950 não correspondem à definição moderna do termo, sendo necessário desvincular suas interpretações temporais. Em 1954, surge o *Fortran*, a primeira linguagem científica, com paradigma imperativo. Entre 1958 e 1968, destacam-se linguagens fundamentais como *Lisp* (programação funcional e manipulação de

listas), *Algol 60* (estruturação de programas), *Cobol* (aplicações comerciais com sintaxe próxima à linguagem natural), *Basic* (popularização da programação), *PL/I* (integração entre ciência e negócios), *APL* (notação matemática concisa), *Snobol* (manipulação de cadeias), *Simula 67* (fundamentos da orientação a objetos) e *Algol 68* (tipos de dados avançados).

Entre 1970 e 1990, surgem linguagens como *Pascal* e *Modula-2* (descendentes de Algol com forte tipagem), *Prolog* (programação lógica e IA), *Ada* (segurança e modularidade para sistemas críticos), *Smalltalk* (ambientes interativos e OO), e *C++* (eficiência com orientação a objetos).

De 1991 aos anos 2000, surgem linguagens que ampliam o escopo da programação: *Python* (simplicidade e versatilidade), *Java* (portabilidade e robustez), *JavaScript* e *PHP* (scripting voltado à web), *C#* (produtividade com base no C++ e Java), além de linguagens híbridas como HTML+JavaScript e XML+XSLT, que integram estrutura e processamento dinâmico.

2. Ambientes de Programação

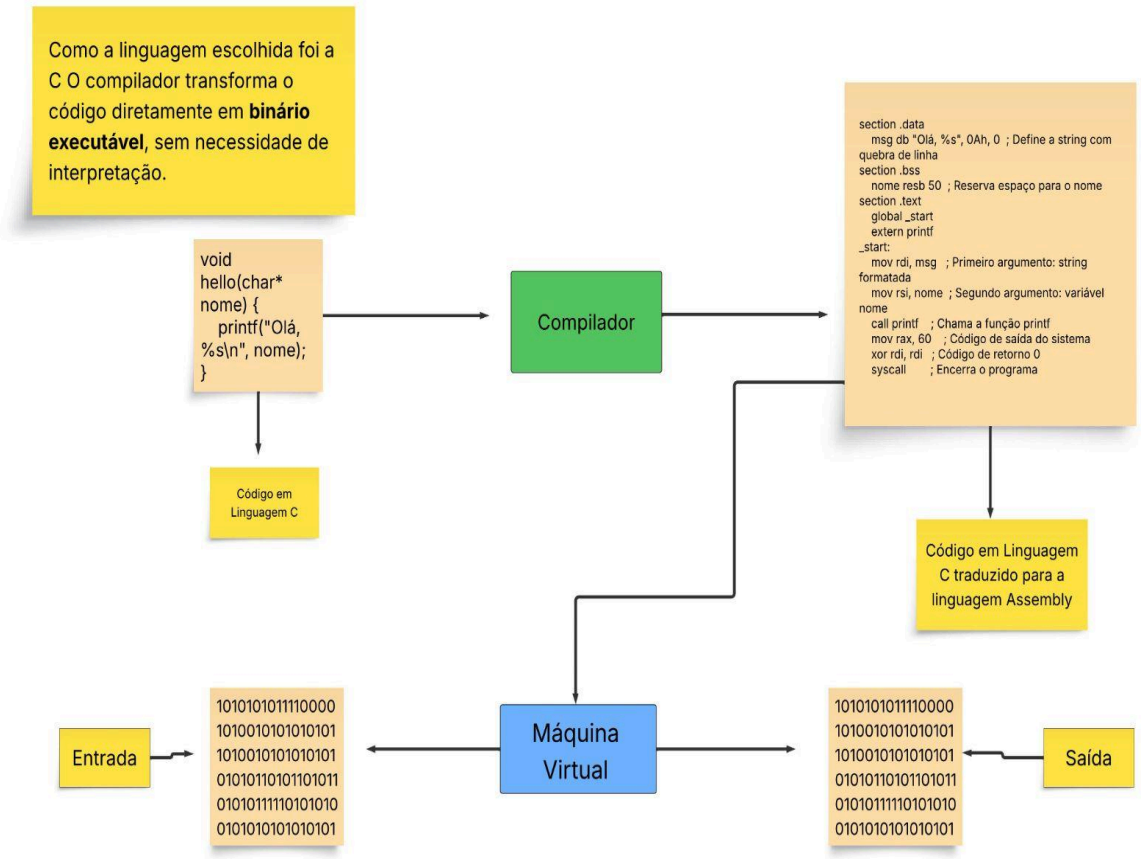


Diagrama gerado com referência ao encontrado no livro: BARBOSA, Cynthia da S.; LENZ, Maikon L.; LACERDA, Paulo S. Pádua de; et al. **Compiladores**. Porto Alegre: SAGAH, 2021. *E-book.* p.14. ISBN 9786556902906. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9786556902906/. Acesso em: 17 mai. 2025.

Definição de Compilador, Interpretador e Máquina Virtual:

- **Compilador:** é um programa que transforma o código escrito em uma linguagem de alto nível (como C, Java ou Python) em linguagem de máquina (binário), que o computador consegue executar. Esse processo é feito antes da execução do programa.
- **Interpretador:** diferente do compilador, o interpretador lê e executa o código linha por linha, sem convertê-lo previamente para linguagem de máquina. É comum em linguagens como Python e JavaScript.
- **Máquina virtual:** é um ambiente de execução que simula um computador dentro de outro. Em linguagens como Java, o código é compilado para um formato

intermediário (bytecode), que é executado por uma máquina virtual (como a JVM - Java Virtual Machine). Isso permite maior portabilidade entre sistemas operacionais.

No exemplo acima, a linguagem utilizada foi C. Essa linguagem, a princípio, não utiliza um interpretador, pois o compilador já se encarrega de traduzi-la diretamente para linguagem de máquina, composta por códigos binários (0 e 1), que o sistema é capaz de interpretar. Para ilustrar como ocorre essa tradução, utilizei a linguagem Assembly, demonstrando como a máquina "enxerga" o código em C, ou seja, como o compilador transforma o texto em C para uma forma intermediária antes da conversão final em linguagem de máquina.

3. Descrições Sintáticas e Semânticas

Linguagem Flawless - Mini Gramática

3.1. Introdução

A linguagem *Flawless* é uma linguagem fictícia inspirada nos álbuns *'Renaissance'* de Beyoncé, *'ArtPop'* de Lady Gaga*, 'The Rise and Fall of a Midwest Princess'* de Chappell Roan, e *'Brat'* de Charli Xcx. É uma linguagem performática, emocional, glamurosa e expressiva, com vocabulário e estrutura únicos e *cunts* , assim como os álbuns e artistas mencionados.

3.2. Tokens e Análise Léxica

1. **Identificadores:** [a-zA-Z_][a-zA-Z0-9_]*
2. **Números:** [0-9]+
3. **String:** "([^"]*)"
4. **Operadores básicos:** + | - | * | / |
5. **Delimitadores:** ; | (|) | { | }
6. **Palavras Reservadas:** serve, glam, reflect, pop, iconic, dual, feel
7. **Comentários:** // texto até o fim da linha; " " para textos com mais de uma linhas

3.3. Dicionário de Termos da Linguagem *Flawless*

★ ***serve***: Define uma função. Representa o ato de 'entregar tudo'. Equivalente a *'function'*.

★ ***glam****:* Define uma variável numérica com valor de brilho, energia ou intensidade. ***reflect:*** Imprime uma mensagem introspectiva, como um espelho emocional. Equivalente a '*print"*.

★ ***pop:*** Executa uma ação performática. Equivale a chamar uma função.

★ ***iconic:*** Define constantes imutáveis com valor simbólico.

★ ***dual:*** Declara variáveis que representam conflitos internos ou ambiguidades.

★ ***feel* (...)**: Executa um sentimento. Usada para expressar vulnerabilidade e emoção.

3.4. Exemplos de Erros Sintáticos

1. Falta de ponto e vírgula
Exemplo:
glam energy = 100
Erro: Esperado ';' ao final.

Mensagem Flawless: "Girl, you forgot to end the vibe properly. Add a ';'."

2. Palavra-chave como nome de variável
Exemplo:
iconic pop = 90;
Erro: 'pop' e palavra-chave.
Mensagem Flawless: "You're trying to rename a legend. Pick another name, babe.

4. Tipos de Dados

Linguagens escolhidas: *Python,* C e *JavaScript*

Cenário de aplicação: Depósito em conta bancária

Aspectos que serão comparados:

1. **Tipo de tipagem:** estática ou dinâmica;
2. **Verificação de tipos:** em tempo de compilação ou execução;
3. **Conversão de tipos:** implícita ou explícita.

4.1. Python

- **Tipagem**: **dinâmica e forte**
- Os tipos são definidos automaticamente, mas **não misturam-se sem conversão**. Python detecta tipos em tempo de execução. Embora flexível, exige cuidado ao misturar tipos, pois pode gerar erros de execução.

Exemplo de implementação:

```
saldo = 1000.0          # float
deposito = 500          # int

saldo += deposito       # OK: Python converte int para float automaticamente

# saldo = "1000" + 500   # Erro: não permite somar string com número (tipagem forte)
```

4.2. C

- **Tipagem**: estática e fraca

- Os tipos são definidos **explicitamente** e verificados em tempo de compilação, mas a linguagem permite misturar alguns tipos com **conversões implícitas arriscadas**. C é mais seguro que JavaScript nesse aspecto, mas mais permissivo que Python em operações entre numéricos (int → float, por exemplo).

Exemplo de implementação:

```
#include <stdio.h>

int main() {
```

```
    float saldo = 1000.0;       // float
    int deposito = 500;         // int

    saldo += deposito;          // OK: int convertido implicitamente para float

    char* nome = "João";
    saldo += nome;              // ERRO em compilação: tipos incompatíveis
    return 0;
}
```

4.3. JavaScript

- **Tipagem**: **dinâmica e fraca**

- Os tipos são detectados em tempo de execução e há **muita conversão implícita**, o que pode causar bugs sutis. A tipagem fraca permite erros inesperados — como somar uma string com número e obter outro tipo (string). Isso exige muito cuidado.

Exemplo de implementação:

```
let saldo = 1000.0;      // float
let deposito = 500;      // int (na prática, tudo é Number)

saldo += deposito;       // OK: soma numérica

saldo = "1000" + 500;    // Resultado: "1000500" (concatenação de string)
```

Linguagem	Tipagem	Verificação	Conversão Implícita	Segurança de

				Tipos
Python	Dinâmica, forte	Execução	Limitada	Alta (para tipos mistos)
C	Estática, fraca	Compilação	Sim (ex. int → float)	Média
JavaScript	Dinâmica, fraca	Execução	Sim (ex. num ↔ string)	Baixa

5. Estruturas de Controle

DataFrame fictício de funcionários

```
import pandas as pd

df = pd.DataFrame(
    {
        "Nome": ["Antonio Silva", "Carlos Souza", "Debora Machado"],
        "Idade": [22, 35, 37],
        "Gênero": ["M", "M", "F"],
        "Tempo de Empresa (em anos)": [1, 2, 5]
    }
)

print("Lista de funcionários:\n")
print(df)
print("\nVerificação do tempo de empresa:\n")

# Estrutura de repetição e seleção
for index, row in df.iterrows():
    if row["Tempo de Empresa (em anos)"] == 1:
        print(f"{row['Nome']} é novo na empresa!")
    elif row["Tempo de Empresa (em anos)"] <= 3:
        print(f"{row['Nome']} já tem alguma experiência.")
    else:
        print(f"{row['Nome']} é um veterano na empresa!")

print("\nFinalizando verificação.")
```

O DataFrame criado em Python armazena informações sobre nome, idade, gênero e tempo de empresa dos funcionários. Esses dados serão analisados utilizando estruturas de controle de fluxo, como *for, if, elif* e *else*, permitindo a verificação e categorização das informações de maneira dinâmica.

6. Subprogramas

Em Python

```python
def altera_por_valor(x):
    x = 10  # Modificação dentro da função
    print(f"Dentro da função: x = {x}")

def altera_por_referencia(lista):
    lista.append(100)  # Modificando a lista original
    print(f"Dentro da função: lista = {lista}")

# Testando passagem por valor (imutável)
num = 5
print(f"Antes da função: num = {num}")
altera_por_valor(num)
print(f"Depois da função: num = {num}")
print("\n---\n")


numeros = [1, 2, 3]
print(f"Antes da função: lista = {numeros}")
altera_por_referencia(numeros)
print(f"Depois da função: lista = {numeros}")  # A lista é alterada
```

- **Tipos imutáveis** (como *int*, *str*) passam por valor — qualquer alteração dentro da função não afeta a variável original.
- **Tipos mutáveis** (como *list*, *dict*) passam por referência — mudanças feitas dentro da função refletem na variável original.

Em Java

```java
import java.util.ArrayList;
import java.util.List;

public class PassagemParametros {
    // Passagem por valor (imutável)
    static void alteraPorValor(int x) {
        x = 10;
        System.out.println("Dentro da função: x = " + x);
    }
```

```java
    // Passagem por referência (mutável)
    static void alteraPorReferencia(List<Integer> lista) {
        lista.add(100);
        System.out.println("Dentro da função: lista = " + lista);
    }

    public static void main(String[] args) {
        // Testando passagem por valor
        int num = 5;
        System.out.println("Antes da função: num = " + num);
        alteraPorValor(num);
        System.out.println("Depois da função: num = " + num); // Não foi alterado

        System.out.println("\n---\n");


        List<Integer> numeros = new ArrayList<>();
        numeros.add(1);
        numeros.add(2);
        numeros.add(3);
        System.out.println("Antes da função: lista = " + numeros);
        alteraPorReferencia(numeros);
        System.out.println("Depois da função: lista = " + numeros); // Foi alterada
    }
}
```

Em Java, tipos primitivos (*int*, double, etc.) são passados por valor, ou seja, a cópia da variável é enviada para a função e não é alterada fora dela.

Já objetos (*List*, Array, *StringBuilder*, etc.) são passados por referência, permitindo que alterações dentro da função reflitam no objeto original.

Portanto, a **passagem por valor** ocorre quando uma **cópia** da variável é enviada para a função, e alterações dentro da função **não afetam** a variável original.

A **passagem por referência** ocorre quando o próprio **endereço de memória** do objeto é enviado, permitindo que mudanças na função **modifiquem o original**. Linguagens como **Python** e **Java** têm comportamentos distintos: em Python, listas e dicionários são mutáveis, enquanto em Java, objetos podem ser alterados por referência.

7. Implementação de Subprogramas

Exemplo – Fatorial Recursivo com Python

```python
def fatorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * fatorial(n - 1)

print(fatorial(3))
```

Como funciona a pilha de chamadas

Chamando fatorial(3):

- Primeira chamada: fatorial(3)
    - 3 * fatorial(2)
    - A execução é pausada até fatorial(2) retornar.
- Segunda chamada: fatorial(2)
    - 2 * fatorial(1)
    - A execução é pausada até fatorial(1) retornar.
- Terceira chamada: fatorial(1)
    - Condição de parada: retorna 1.

Representação da Pilha de Chamadas

A pilha funciona no esquema LIFO (Last In, First Out). Cada chamada cria um novo frame na pilha com suas variáveis locais.

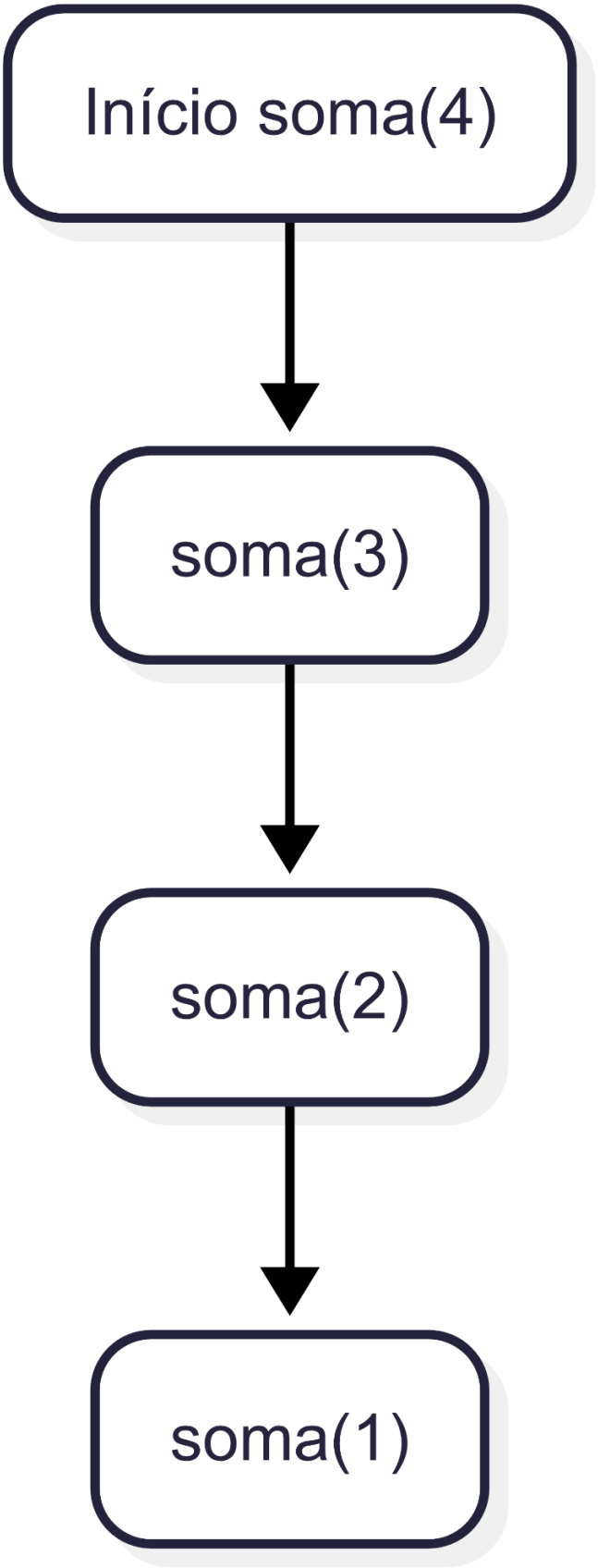
- **Antes da primeira chamada:**
    - Pilha vazia
- **Chamada fatorial(3):**
    - fatorial(3): n = 3
- **Chamada fatorial(2) (empilha acima):**
    - fatorial(2): n = 2
    - fatorial(3): n = 3

- **4. Chamada fatorial(1) (empilha acima):**
    - fatorial(1): n = 1
    - fatorial(2): n = 2
    - fatorial(3): n = 3
- **5. Retornos (desempilhando)**
    - fatorial(1) retorna 1 → volta para fatorial(2)
    - fatorial(2) calcula 2 * 1 = 2 → volta para fatorial(3)
    - fatorial(3) calcula 3 * 2 = 6

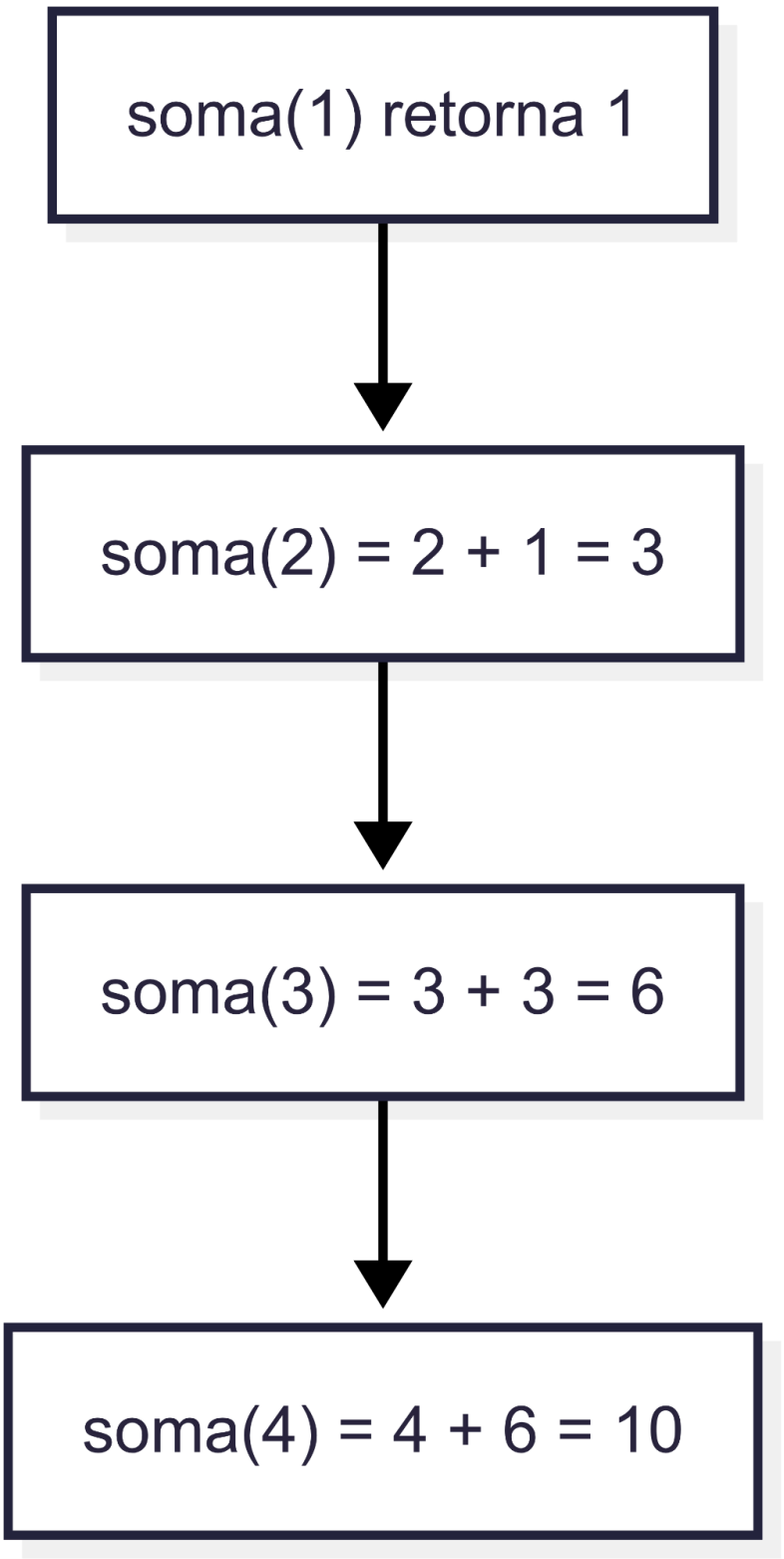
Resultado final: 6

- Cada chamada **pausa sua execução** e empilha o estado atual (valor de n, endereço de retorno).
- Ao atingir a condição de parada, as chamadas começam a **desempilhar** em ordem inversa.
- Essa estrutura permite que a função "lembre" onde parou em cada chamada.

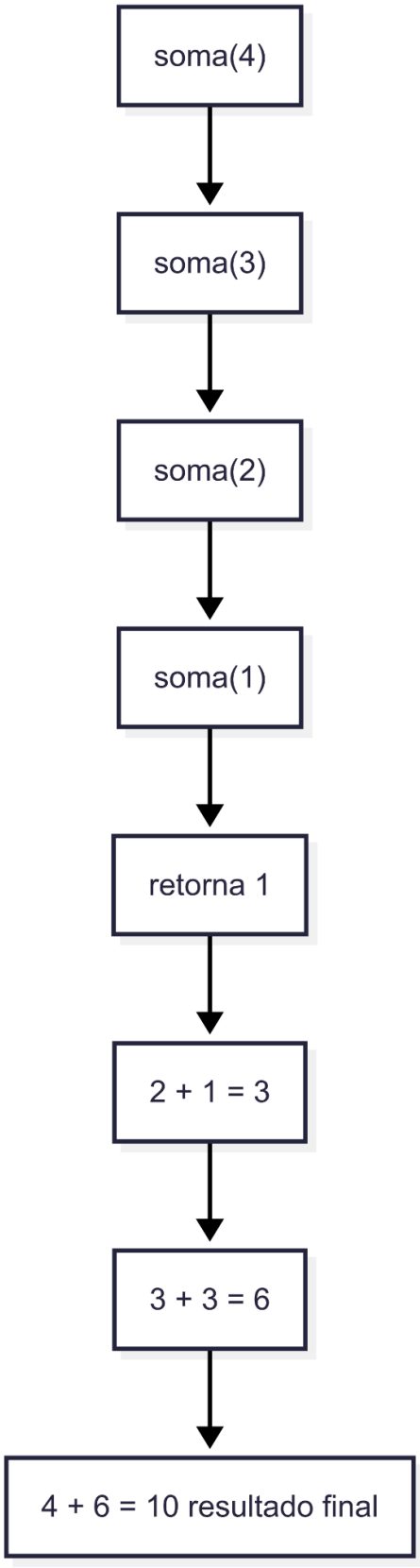
Pilha de chamadas – Empilhando



Pilha de chamadas – Desempilhando (retornos)



Pilha completa (ida e volta simplificada)



8. Programação Orientada a Objetos

Domínio escolhido: Serviços Bancários

- **Classe base (superclasse):** ContaBancaria
- **Subclasses:**
  - ContaCorrente
  - ContaPoupanca

Cada uma vai ter atributos e métodos próprios, além de herdar funcionalidades da classe base.

Código em Python

```
# Classe base
class ContaBancaria:
    def __init__(self, numero, titular, saldo=0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo

    def depositar(self, valor):
        self.saldo += valor
        print(f"Depósito de R${valor} realizado. Saldo atual: R${self.saldo}")

    def sacar(self, valor):
        if valor <= self.saldo:
            self.saldo -= valor
            print(f"Saque de R${valor} realizado. Saldo atual: R${self.saldo}")
        else:
            print("Saldo insuficiente.")
```

```
    def exibir_saldo(self):
        print(f"Conta {self.numero} - Titular: {self.titular} - Saldo: R${self.saldo}")


# Subclasse: Conta Corrente
class ContaCorrente(ContaBancaria):
    def __init__(self, numero, titular, saldo=0, limite=500):
        super().__init__(numero, titular, saldo)
        self.limite = limite

    def sacar(self, valor):
        if valor <= self.saldo + self.limite:
            self.saldo -= valor
            print(f"Saque de R${valor} realizado com limite. Saldo atual: R${self.saldo}")
        else:
            print("Limite excedido.")


# Subclasse: Conta Poupança
class ContaPoupanca(ContaBancaria):
    def __init__(self, numero, titular, saldo=0, rendimento=0.03):
        super().__init__(numero, titular, saldo)
        self.rendimento = rendimento
```

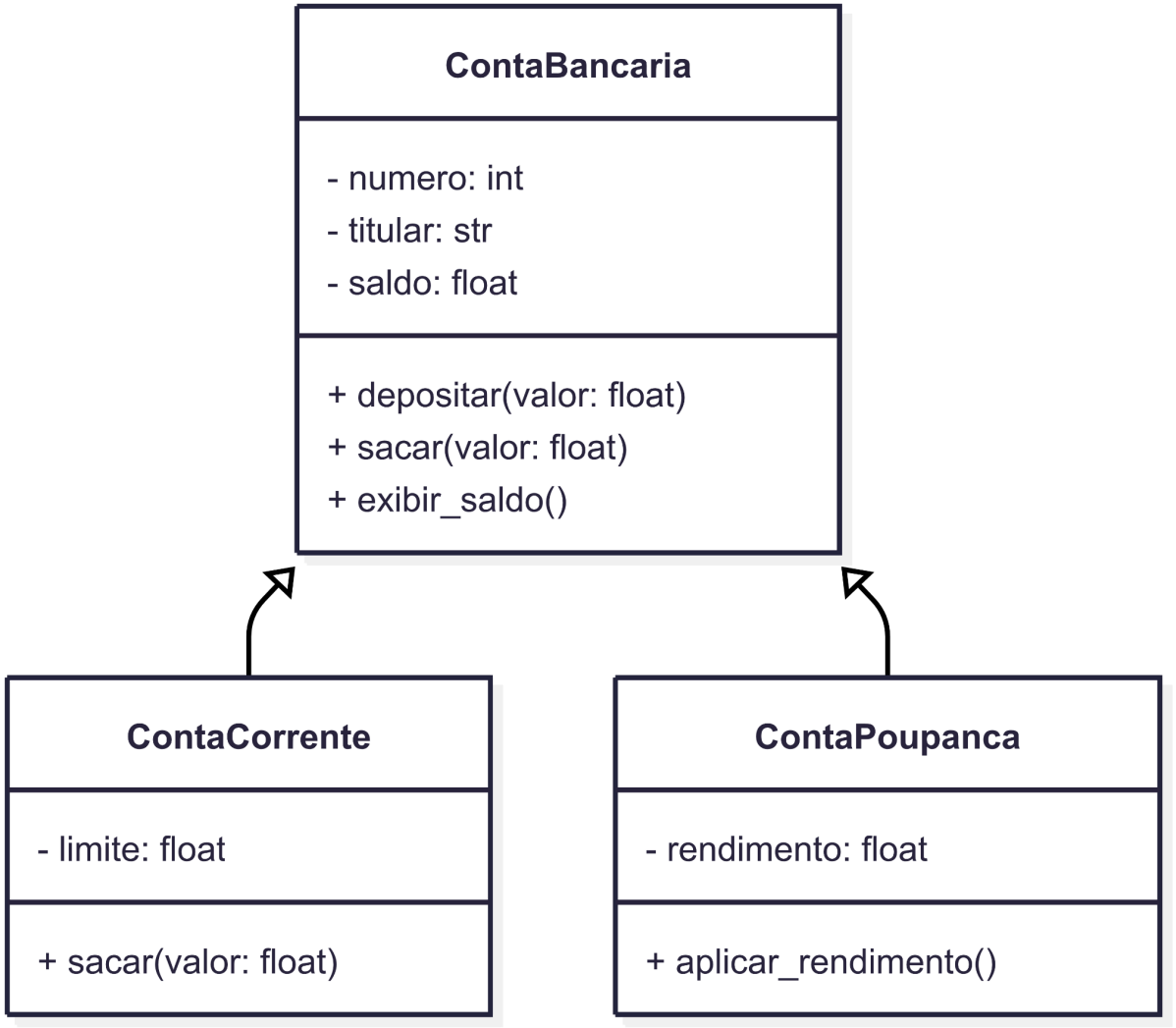
```
    def aplicar_rendimento(self):
        self.saldo += self.saldo * self.rendimento
        print(f"Rendimento aplicado. Saldo atual: R${self.saldo:.2f}")



# Exemplo de uso
conta1 = ContaCorrente(101, "Ana", 1000)
conta2 = ContaPoupanca(102, "Ruth", 500)

conta1.sacar(1200)
conta2.aplicar_rendimento()
conta2.exibir_saldo()
```

Diagrama de Classes



9. Concorrência

Diferença entre Threads e Processos

Processos

- São programas independentes em execução.
- Cada processo tem seu próprio espaço de memória.
- Comunicação entre processos é mais complexa (usa pipes, sockets, etc.).
- Geralmente usados quando tarefas não compartilham dados e podem rodar isoladas.
- Criar processos consome mais recursos do sistema.

Threads

- São subtarefas de um mesmo processo.
- Compartilham o mesmo espaço de memória do processo principal.
- Comunicação é mais simples (compartilham variáveis globais), mas precisa de sincronização para evitar problemas de concorrência (race conditions).
- Mais leves e rápidas de criar que processos.
- Usadas quando há tarefas relacionadas e que precisam compartilhar dados.

Exemplo de Concorrência Personalizado

Situação: Simular o processamento de pedidos bancários

- Uma thread verifica saldo.
- Outra thread realiza saques simultâneos.

Código com Threads (compartilham saldo)

```
import threading
import time

# Saldo inicial
saldo = 1000
lock = threading.Lock()

def verificar_saldo():
    global saldo
    for _ in range(5):
        time.sleep(1)
```

```
        with lock:
            print(f"[Verificação] Saldo atual: R$ {saldo}")

def sacar(valor):
    global saldo
    for _ in range(5):
        time.sleep(1.5)
        with lock:  # Impede que duas threads modifiquem o saldo ao mesmo tempo
            if saldo >= valor:
                saldo -= valor
                print(f"[Saque] Saque de R$ {valor} realizado. Saldo: R$ {saldo}")
            else:
                print("[Saque] Saldo insuficiente!")

# Criando threads
t1 = threading.Thread(target=verificar_saldo)
t2 = threading.Thread(target=sacar, args=(200,))

t1.start()
t2.start()

t1.join()
t2.join()

print("Transações concluídas!")
import threading
import time

# Saldo inicial
saldo = 1000
lock = threading.Lock()

def verificar_saldo():
    global saldo
    for _ in range(5):
        time.sleep(1)
        with lock:
            print(f"[Verificação] Saldo atual: R$ {saldo}")

def sacar(valor):
    global saldo
    for _ in range(5):
```

```
        time.sleep(1.5)
        with lock:  # Impede que duas threads modifiquem o saldo ao mesmo tempo
            if saldo >= valor:
                saldo -= valor
                print(f"[Saque] Saque de R$ {valor} realizado. Saldo: R$ {saldo}")
            else:
                print("[Saque] Saldo insuficiente!")

# Criando threads
t1 = threading.Thread(target=verificar_saldo)
t2 = threading.Thread(target=sacar, args=(200,))

t1.start()
t2.start()

t1.join()
t2.join()

print("Transações concluídas!")
```

Código com Processos (cada processo tem seu saldo separado)

```
from multiprocessing import Process, Value
import time

def verificar_saldo(saldo):
    for _ in range(5):
        time.sleep(1)
        print(f"[Verificação] Saldo atual: R$ {saldo.value}")

def sacar(saldo, valor):
    for _ in range(5):
        time.sleep(1.5)
        if saldo.value >= valor:
            saldo.value -= valor
            print(f"[Saque] Saque de R$ {valor} realizado. Saldo: R$ {saldo.value}")
        else:
            print("[Saque] Saldo insuficiente!")

if __name__ == '__main__':
    saldo = Value('i', 1000)  # Valor compartilhado entre processos
```

```
p1 = Process(target=verificar_saldo, args=(saldo,))
p2 = Process(target=sacar, args=(saldo, 200))

p1.start()
p2.start()

p1.join()
p2.join()

print("Transações concluídas!")
```

10. Gerenciamento de Memória

Comparativo: Gerenciamento de Memória (Python x C)

Aspecto	Python	C
Tipo de gerenciamento	Automático (Garbage Collector)	Manual (programador controla tudo)
Alocação de memória	Automática ao criar objetos (heap)	malloc(), calloc() ou realloc()
Liberação de memória	Automática (coleta de lixo)	free() precisa ser chamada pelo usuário
Coleta de lixo	Usa contagem de referências e GC cíclico para objetos não utilizados	Não possui coleta automática de lixo
Riscos comuns	Vazamento raro (exceto ciclos não coletados) e fragmentação baixa	Vazamentos de memória e ponteiros danificados
Controle do programador	Pouco controle direto; foca na lógica	Controle total sobre cada byte alocado
Facilidade	Simples para o desenvolvedor	Mais complexa, exige atenção constante

Python: O programador não precisa se preocupar em liberar memória. O Garbage Collector monitora os objetos e libera espaço quando não há mais referências para eles.

C: O programador deve alocar e liberar manualmente a memória. Isso oferece mais controle, mas aumenta o risco de erros como memory leak ou segfault.

11. Programação Funcional

Problema escolhido: Calcular o total de compras e aplicar desconto em itens caros

Cenário real:

Temos uma lista de valores de produtos comprados no banco (por exemplo, valores de taxas ou serviços). Queremos:

- Somar todos os valores (recursão).
- Filtrar apenas os produtos acima de R$ 100 (função de alta ordem).
- Aplicar desconto de 10% nesses produtos (função de alta ordem).

Implementação

```
from functools import reduce

# Lista de valores de compras
compras = [50, 120, 300, 80, 200]

# --- Função recursiva para somar ---
def soma_recursiva(lista):
    if not lista:
        return 0
    return lista[0] + soma_recursiva(lista[1:])

# --- Funções de alta ordem ---
def aplicar_desconto(valor):
    return valor * 0.9  # desconto de 10%

# Filtrar itens acima de 100
itens_caros = list(filter(lambda x: x > 100, compras))

# Aplicar desconto nos itens caros
itens_com_desconto = list(map(aplicar_desconto, itens_caros))

# Calcular total com recursão
total = soma_recursiva(itens_com_desconto)

# Também podemos usar reduce para comparar
total_reduce = reduce(lambda x, y: x + y, itens_com_desconto, 0)
```

```
print("Itens caros:", itens_caros)
print("Itens com desconto:", itens_com_desconto)
print("Total (recursão):", total)
print("Total (reduce):", total_reduce)
```

Saída Esperada

```
Itens caros: [120, 300, 200]
Itens com desconto: [108.0, 270.0, 180.0]
Total (recursão): 558.0
Total (reduce): 558.0
```

12. Programação Lógica

Problema: Relações de Família

Queremos modelar a árvore genealógica e responder perguntas como:

- "Quem é pai de quem?"
- "Quem é avô?"
- "Quem são irmãos?"

Fatos (conhecimento base)

```
% Relações de paternidade
pai(joao, maria).
pai(joao, pedro).
pai(pedro, ana).

mae(ana_clara, maria).
mae(ana_clara, pedro).

% Relações derivadas
avo(X, Y) :- pai(X, Z), pai(Z, Y).
avo(X, Y) :- pai(X, Z), mae(Z, Y).
irmao(X, Y) :- pai(P, X), pai(P, Y), X \= Y.
```

Consultas possíveis

- **Quem são os filhos de João?**

```
?- pai(joao, X).
```

Resultado

```
X = maria ;
X = pedro.
```

Quem é avô de Ana?

?- avo(X, ana).

Resultado

X = joao.

Maria e Pedro São Irmão?

?- irmao(maria, pedro).

Resultado

true

Explicação

- Em Prolog, declaramos fatos (ex.: pai(joao, maria).) e regras (ex.: avo(X,Y) :- pai(X,Z), pai(Z,Y).)
- As consultas (?- ...) pedem para o sistema verificar se algo é verdadeiro ou encontrar valores que satisfaçam a regra.

13. Linguagens para Scripts e Web

Contexto do Script

Imagine que você trabalha em um banco e precisa ler um arquivo CSV com transações bancárias, identificar quais transações são suspeitas (valor acima de R$10.000) e gerar um relatório resumido automaticamente.

Script em Python

```python
import csv

# Arquivo de entrada e saída
entrada = "transacoes.csv"
saida = "relatorio_suspeitas.txt"

# Função para detectar transações suspeitas
def transacao_suspeita(valor):
    return valor > 10000

# Ler arquivo CSV e processar
suspeitas = []
with open(entrada, newline='', encoding='utf-8') as arquivo:
    leitor = csv.DictReader(arquivo)
    for linha in leitor:
        valor = float(linha["valor"])
        if transacao_suspeita(valor):
            suspeitas.append(f"{linha['id_transacao']} - Cliente: {linha['cliente']} - Valor: R$ {valor}")

# Gerar relatório
with open(saida, "w", encoding='utf-8') as relatorio:
    relatorio.write("RELATÓRIO DE TRANSAÇÕES SUSPEITAS\n")
    relatorio.write("=" * 40 + "\n")
    if suspeitas:
        relatorio.write("\n".join(suspeitas))
    else:
        relatorio.write("Nenhuma transação suspeita encontrada.")

print(f"Relatório gerado em: {saida}")
```

Exemplo do CSV (transacoes.csv)

id_transacao,cliente,valor

1,Ana,5000

2,Carlos,12000

3,Mariana,8000

4,João,20000

Saída esperada no relatório (relatorio_suspeitas.txt)

RELATÓRIO DE TRANSAÇÕES SUSPEITAS

==========

2 - Cliente: Carlos - Valor: R$ 12000.0

4 - Cliente: João - Valor: R$ 20000.0

14. Tendências em Linguagens de Programação

Linguagem escolhida: *Dart*

Dart é uma linguagem de programação criada pelo *Google* em 2011, orientada a objetos e de código aberto. Ela tem uma sintaxe parecida com C, *Java* e *JavaScript*, tornando-a mais fácil de aprender para quem já conhece essas linguagens.

Sua principal aplicação é no desenvolvimento de aplicativos móveis e web, sendo usada principalmente com o *Flutter*, um *framework* que permite criar apps para *Android* e *iOS* usando um único código. Além disso, Dart também pode ser utilizado no *back-end* para desenvolver servidores.

Vantagens e Desvantagens

Vantagens:

- Fácil de aprender, especialmente para quem já tem experiência em outras linguagens.
- Multiplataforma, permitindo criar aplicativos para diferentes sistemas sem precisar reescrever o código.
- Alta velocidade de execução, com compilação otimizada.
- Bom suporte da comunidade, com documentação extensa e recursos de código aberto.

Desvantagens:

- Ainda é uma linguagem nova, então pode ser mais difícil encontrar vagas de emprego ou desenvolvedores experientes.
- Tem menos bibliotecas disponíveis em comparação com linguagens mais populares.
- Há esforço técnico para converter código *Dart* em *JavaScript*.

Sistemas que utilizam Dart

Dart é bastante usado em aplicativos móveis que utilizam *Flutter*, incluindo:

- ***Cred*:** Fintech indiana que oferece pagamentos e recompensas.
- ***Mews*:** Plataforma de gestão hoteleira que automatiza processos operacionais.
- ***Insider*:** Empresa de marketing digital focada em personalização.

Além disso, o Google aposta no Flutter para futuros projetos, o que torna o aprendizado de *Dart* uma opção promissora para quem quer trabalhar com desenvolvimento

de aplicativos. O Google tem investido bastante na linguagem e no *Flutter*, e cada vez mais empresas estão adotando essa tecnologia. É uma ótima linguagem para quem quer trabalhar com desenvolvimento móvel e web.

Referências

BARBOSA, Cynthia da S.; LENZ, Maikon L.; LACERDA, Paulo S. Pádua de; et al. **Compiladores**. Porto Alegre: SAGAH, 2021. *E-book.* p.14. ISBN 9786556902906. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9786556902906/. Acesso em: 17 mai. 2025.

MARTINS, Vinicius. *Linguagem Dart: o que é, para que serve e primeiros passos!* Blog da Trybe, 18 set. 2024. Disponível em: Blog da Trybe. Acesso em: 24 maio 2025.

SEBESTA, Robert. **Conceitos de linguagens de programação**. 11. ed. Porto Alegre: Bookman, 2018. *E-book.* p.161. ISBN 9788582604694. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788582604694/. Acesso em: 10 mai. 2025.