Capítulo 9

Subprogramas

Tópicos

- Introdução
- Fundamentos dos subprogramas
- Questões de projeto para subprogramas
- Ambientes de referenciamento local
- Métodos de passagem de argumentos
- Parâmetros que são subprogramas
- Chamada indireta de subprogramas
- Questões de projeto para funções
- Subprogramas sobrecarregados
- Subprogramas genéricos
- Operadores sobrecarregados definidos pelo usuário
- Fechamentos (closures)
- Corrotinas

Introdução

- Existem dois recursos de abstração fundamentais em linguagens de programação:
  - Abstração de Processos
    - Surgiu desde os primórdios das linguagens de alto nível
    - É o conteúdo deste capítulo
  - Abstração de Dados
    - Começou a ganhar ênfase a partir de 1980
    - É assunto do capítulo 11

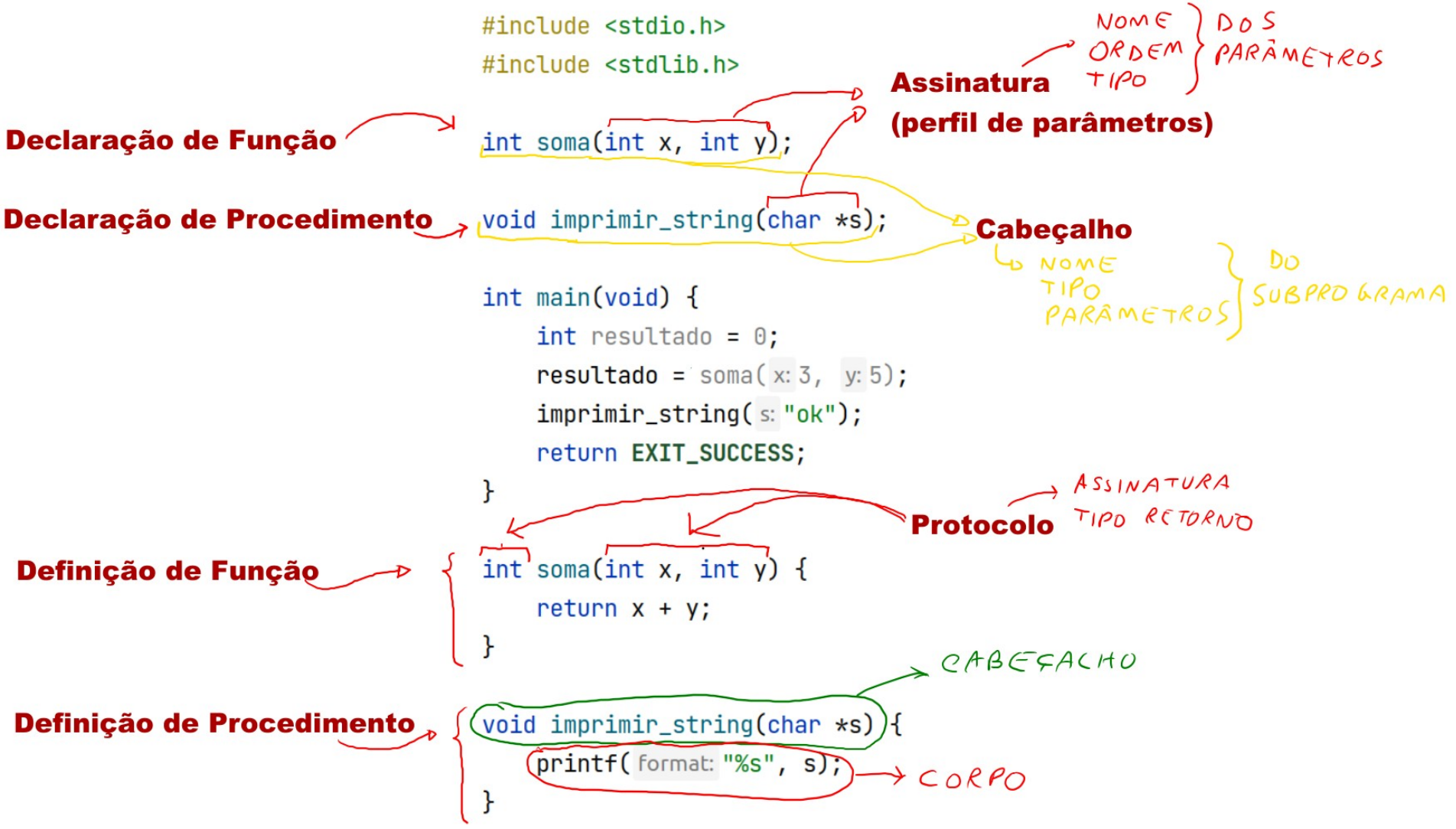
Fundamentos dos Subprogramas

- Com exceção das corrotinas e das unidades concorrentes (capítulo 13), cada subprograma:
  - Tem um único ponto de entrada
  - O programa que chamou fica suspenso durante a exeção do subprograma chamado
  - Quando o subprograma termina, o controle sempre retorne para quem chamou
  - Em geral, têm nomes (alguns são anônimos)

Definições básicas

- A **definição** de um subprograma descreve a interface e as ações da abstração do subprogama (protocolo e corpo).
- Uma **chamada** de subprograma é o pedido explícito para que um subprograma específico seja executado.
- O **cabeçalho** de um subprograma é a primeira parte da definição, incluindo:
  - Nome (exceto de for anônimo)
  - Tipo de subprograma (procedimentos e funções)
  - Parâmetros formais (perfil de parâmetros)
- O **perfil de parâmetros** (**assinatura**) de um subprograma corresponde ao número, ordem e tipo de seus parâmetros formais
- O **protocolo** corresponde ao perfil de parâmetros e o tipo de retorno (principalmente se função)
- Os subprogramas podem ter tanto
  - **declarações** (protótipos): só cabeçalho
  - **definições**: cabeçalho e corpo

Definições básicas



Definições básicas

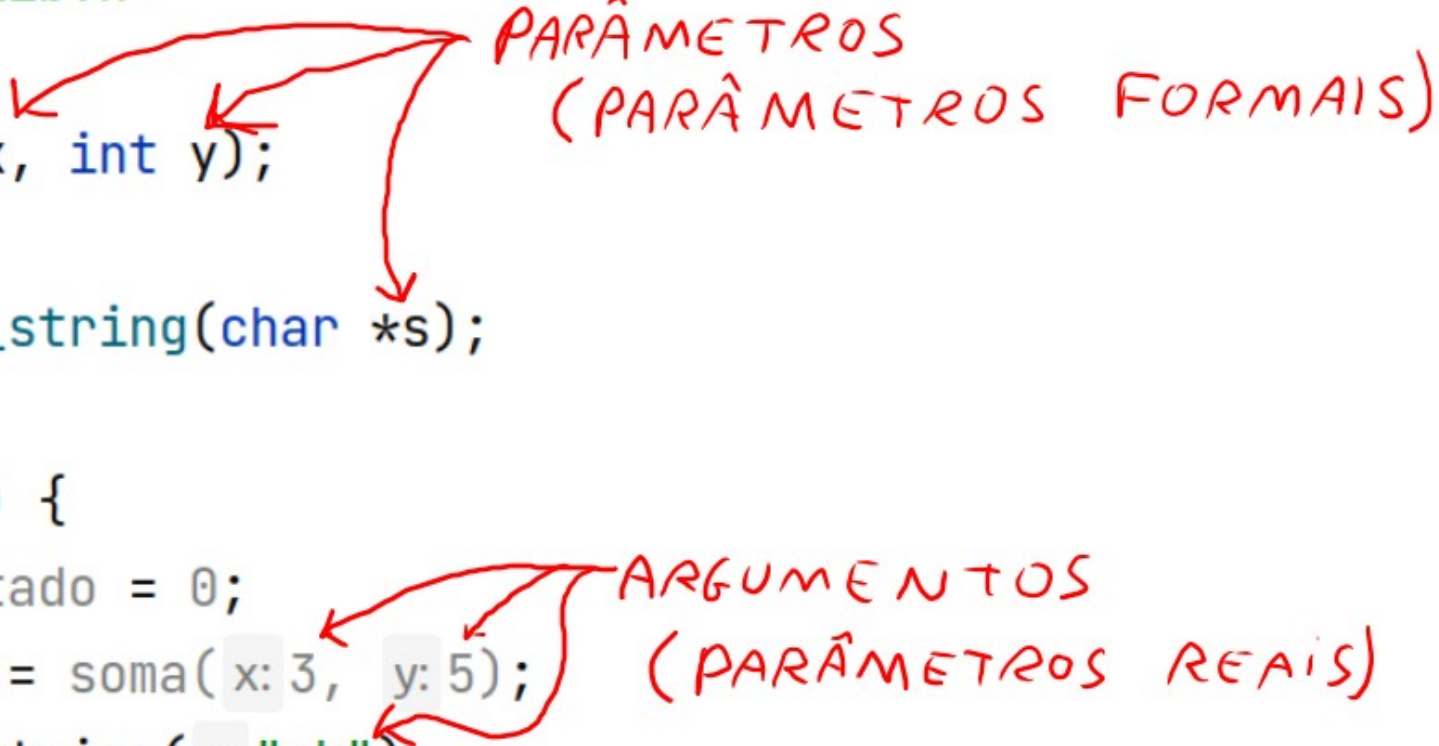
- Um **parâmetro formal** é uma variável dummy (fictícia) listada no cabeçalho e utilizada no corpo do subprograma

- Um **argumento** (**parâmetro real**) representa o valor ou endereço que é utilizado na chamada do subprograma.

- Confusão:
  - Parâmetro x argumento
  - Parâmetro formal x parâmetro real

Definições básicas

```c
#include <stdio.h>
#include <stdlib.h>

int soma(int x, int y);

void imprimir_string(char *s);

int main(void) {
    int resultado = 0;
    resultado = soma( x: 3,  y: 5);
    imprimir_string( s: "ok");
    return EXIT_SUCCESS;
}
```



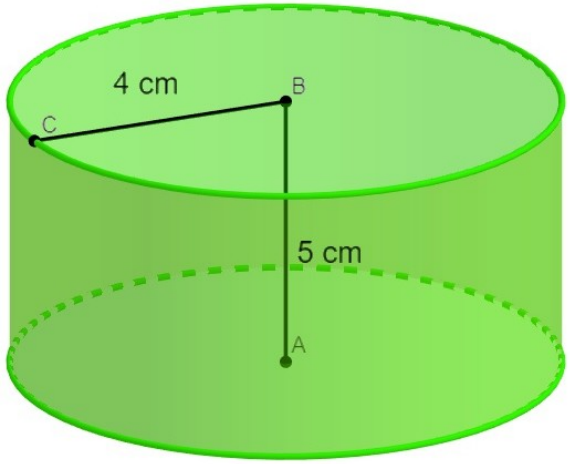
Casamento: parâmetro x argumento

- Posição
  - O víndulo entre os parâmetros e os argumentos é dado pela /posição: o primeiro argumento é vinculado ao primeiro parâmetro e assim por diante
  - Vantagem: Seguro e eficiente
  - Desvantagem: precisa saber a ordem

- Keyword
  - O nome do parâmetro ao qual um argumento deve ser vinculado é especificado na chamada do subprograma
  - Vantagem: parâmetros podem aparecer em qualquer ordem, evitando erros na posição
  - Desvantagem: precisa saber o nome dos parâmetros

Casamento: parâmetro x argumento

```python
import math


def calcula_area_cilindro(h, r):
    return round(math.pi * h * pow(r, 2), 2)


print(calcula_area_cilindro(5, 4))
print(calcula_area_cilindro(r = 4, h = 5))
```



Valor padrão para parâmetros

- Algumas linguagens (C++, Python, Ruby, PHP) permitem informar valores padrão para os parâmetros (são usados se os argumentos não forem informados)
  - Verifique a documentação! Ex.: em C++, parâmetros com valor padrão precisam ficar nas últimas posições

```
class NoRBT:
    def __init__(self, chave, esq=None, dir=None, pai=None, cor="R"):
        self.chave = chave
        self.esq = esq
        self.dir = dir
        self.pai = pai
        self.cor = cor


no1 = NoRBT(14, cor="B")
no2 = NoRBT(7, pai=no1)
no1.esq = no2
```

Número variável de parâmetros

- Algumas linguagens (C#, Lisp, Scheme) permitem utilizar um número variável de argumentos!
  - C# precisa que os argumentos sejam do mesmo tipo, pois são colocados em um array precedido pela palavra **`params`**
  - `Lisp é matador!`

```
(defun teste (primeiro &rest resto)
  (list 'você 'informou first 'seguido 'por (length resto) 'argumentos 'adicionais))

(teste 1 2 3 4 5)
(VOCÊ INFORMOU 1 SEGUIDO POR 4 ARGUMENTOS ADICIONAIS)


(teste 'abrantes 'gosta 'de 'programas 'em 'lisp)
(VOCÊ INFORMOU ABRANTES SEGUIDO POR 5 ARGUMENTOS ADICIONAIS)

(teste)
Error: TESTE got 0 args, wanted at least 1 arg.
  [condition type: PROGRAM-ERROR]
```

```
(+ 1 2)
3

(+ 1 2 3)
6

(+ 1 2 3 4 5 6 7 8 9 10)
55

(+ 6)
6

(+)
0
```

```
(defun conta-argumentos (&rest args) (length args))

(conta-argumentos 1 2 3 4 5)
5

(conta-argumentos)
0
```

Número variável de parâmetros

- Em Python há duas maneiras de fazer:

```
def funcao1(*argumentos):
    for i in argumentos:
        print(i)
```

```
funcao1("teste", 1, "Casa", 2)
```

```
def funcao2(primeiro, *resto):
    print(primeiro)
    for i in resto:
        print(i)
```

```
funcao2("Abrantes", "Flávia", "Tomás", "Elis")
```

```
def funcao3(**argumentos):
    for key, value in argumentos.items():
        print(f"{key} = {value}")
```

```
funcao3(nome="Abrantes", cargo="Professor", status="Novato")
```

Procedimentos e Funções

- Existem 2 tipos de subprogramas
  - **Procedimentos** são sentenças que realizam alguma tarefa mas não retornam nenhum resultado para quem chamou
  - **Funções** também são sentenças que realizam alguma tarefa e retornam algum resultado para quem chamou
- Em geral:
  - Espera-se que não causem efeitos colaterais
  - Na prática, se não tomar cuidado, ocorrerão efeitos colaterais

Questões de projeto: subprogramas

- As variáveis locais serão estáticas ou dinâmicas?
- A definição de um subprograma pode aparecer dentro de outro subprograma?
- Que métodos de passagem de argumentos serão fornecidos?
- Os tipos dos parâmetros serão checados?
- Se os subprogramas podem ser passados como argumentos e se os subprogramas podem ser aninhados, qual é o ambiente de referenciamento de um subprograma passado como argumento?
- Efeitos colaterais serão permitidos?
- Que tipos de valores podem ser retornados pelas funções?
- Quantos valores podem ser retornados pelas funções?
- Os subprogramas podem ser sobrecarregados?
- Os subprogramas pode ser genéricos?
- Se a linguagem permite subprogramas aninhados, haverá suporte para fechamentos (closures)?

Ambiente de referenciamento local

- Variáveis locais podem ser stack-dynamic (dinâmicas de pilha)
  - Vantagens
    - Suporte para recursão
    - Armazenamento é compartilhado dentro os subprogramas
  - Desvantagens
    - Tempo de inicialização, alocação/desaloação
    - Endereçamento indireto
- Variáveis locais podm ser static (estáticas)
  - Vantagens e desvantagens são opostas

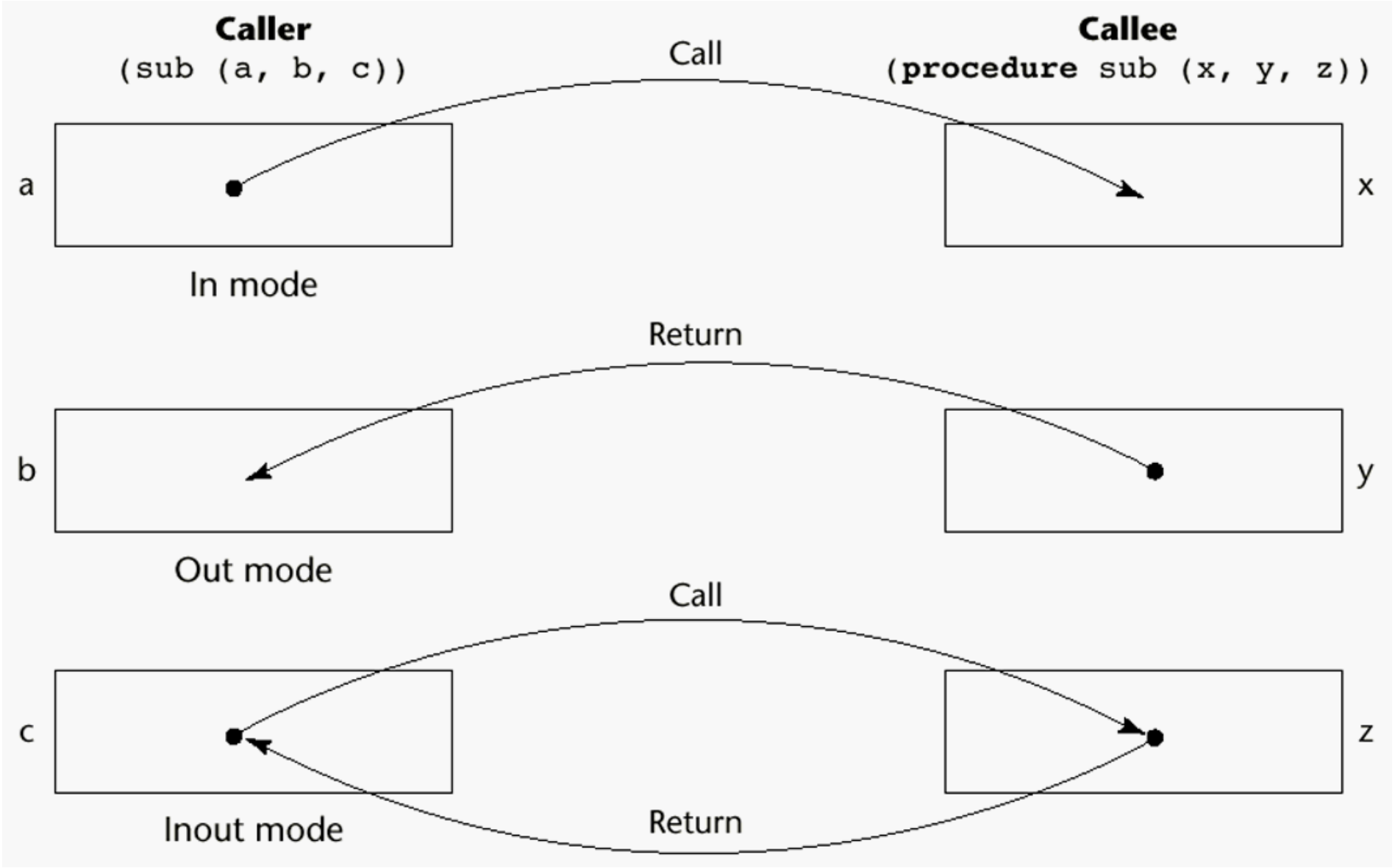
Ambiente de referenciamento local: examplos

- Na maioria das linguagens atuais, as variáveis locais são stack dynamic
- Linguagem derivadas de C têm variáveis locais stack dynamic por padrão, mas podem ser declaradas com `static`
- Métodos em C++, Java, Python e C# somente possuem variáveis locais stack dynamic

Modelos de passagem de parâmetros

- Modo de entrada (in)
- Modo de saída (out)
- Mode de entrada e saída (inout)

Modelos de passagem de parâmetros



Modelos de passagem de parâmetros

```
def teste(l1, l2):
    if len(l1) == len(l2):
        for i in range(len(l1)):
            l2[i] = l1[i] + l2[i]
    else:
        return
    l3 = l1 + l2
    return l3


l1 = [1, 2, 3]
l2 = [3, 2, 1]
x = teste(l1, l2)
print(l2, x)
```

Implementação da passagem

- Mover fisicamente o valor
- Mover um caminho de acesso até o valor

Passagem por Valor (modo entrada)

- O valor do argumento é utilizado para inicializar o parâmetro correspondente
  - Implementação é, geralmente, por **cópia do valor**
  - Pode ser implementado através da transmissão de um caminho de acesso, mas isso não é recomentado (garantir proteção de escrita é difícil)
  - *Desvantagem* (se por cópia): precisa de armazenamento adicional e o custo da cópia pode ser grande (se o argumento for grande)
  - *Desvantagem* (se por caminho de acesso): deve haver proteção contra escrita

Passagem por resultado (modo saída)

- Imediatamente antes do controle retornar para o chamador, o valor do parâmetro (que atua como uma variável local) é transmitido de volta para a variável que foi passada como argumento
  - Desvantagem: ocupa espaço por é passado por cópia

- Tem alguns problemas potenciais:

```
void Fixer(out int x, out int y) {
  x = 17;
  y = 35;
}
. . .
f.Fixer(out  a,  out  a);
```

Passagem por valor-resultado (inout)

- É uma combinação de:
  - Passagem por valor
  - Passagem por resultado
- Também chamado de passagem por cópia, pois tudo é feito por cópia de valores
- Parâmetros têm armazenamento local
- Desvantagens:
  - Todas da passagem por resultado
  - Todas da passagem por valor

Passagem por referência (inout)

- O que é passado é o caminho de acesso
- Vantagem: o processo de passagem é eficiente (não há cópias nem armazenamento duplicado)
- Desvantagens:
  - Acesso aos parâmetros é mais lento (quando comparado à passagem por valor)
  - Podem ocorrer efeitos colaterais
  - Aliases não desejáveis

Passagem por nome (inout)

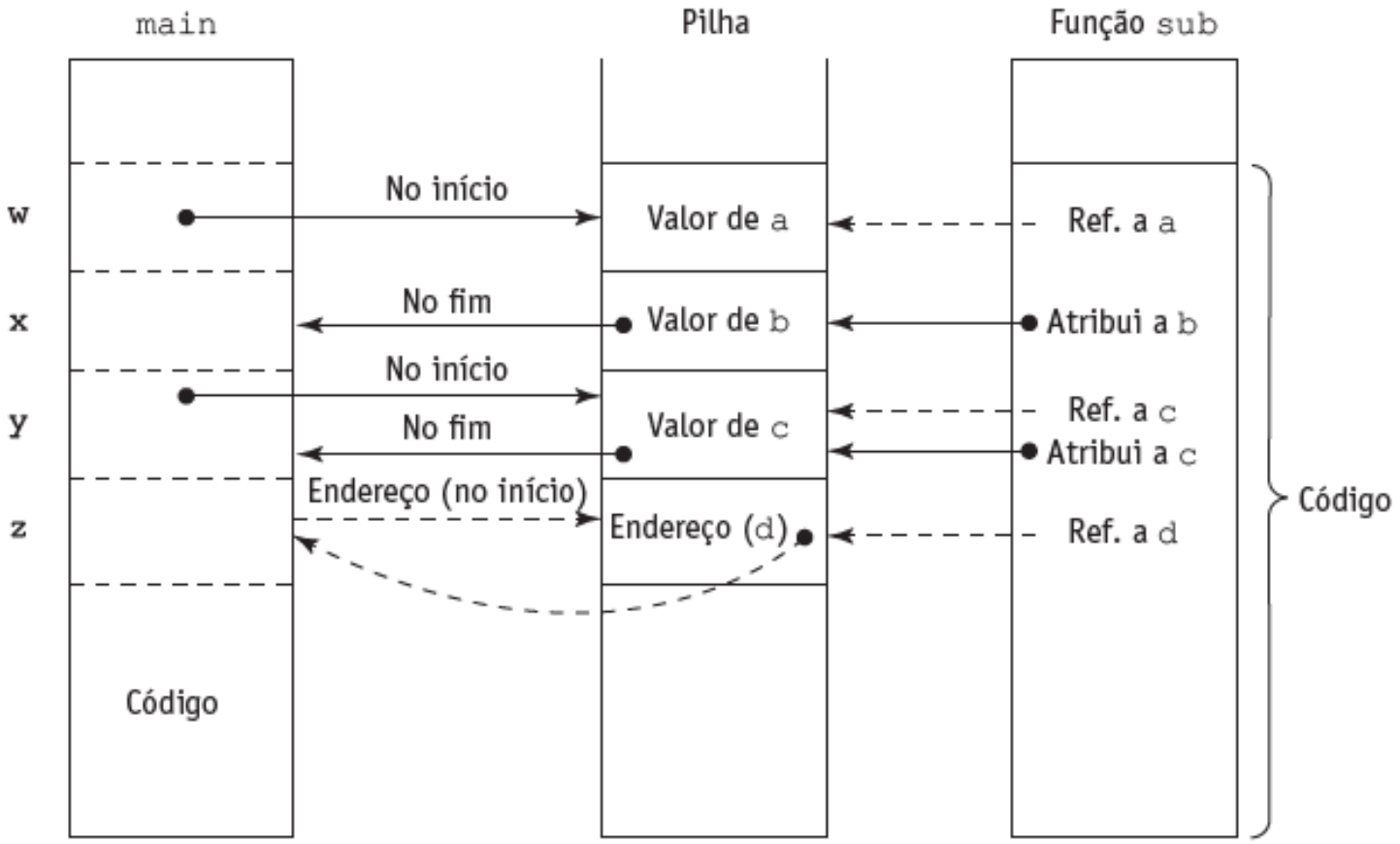
- Ocorre uma substituição textual no nome dos parâmetros
- Coisa de louco
- Não faz parte de nenhuma linguagem amplamente utilizada
- Esqueça
- Se cair na prova, chute

Implementação dos métodos de passagem

- Na maioria das linguagem a implementação dos métodos de passagem de argumentos ocorre no stack de execução

- Passagem por referência é o mais simples de implementar; apenas um endereço é colocado no stack

Implementação dos métodos de passagem



Cabeçalho da função: **void** sub (**int** a, **int** b, **int** c, **int** d)
Chamada da função em main: sub (w, x, y, z)
(passa w por valor, x por resultado, y por valor-resultado, z por referência)

FIGURA 9.2
Uma possível implementação de pilha dos métodos de passagem de parâmetros comuns.

Passagem x Linguagens

- C
  - Passagem por valor
  - Passagem por referência pode ser feita usando-se ponteiros como argumentos
- C++
  - Passagem por valor
  - Usa um tipo especial de ponteiro cahamado “tipo de referência” para passagem por referência
- Java
  - Todos os argumentos são passados por valor
  - Argumentos que são objetos são passador por referência

Passagem x Linguagens

- C#
  - Passagem por valor
  - Passagem por referência pode ser feita usando-se a palavra chave “ref” antes dos parâmetros e argumentos

- Python e Ruby:
  - Passagem por **atribuição** (sim, outro tipo...)
  - Na verdade é um tipo de passagem por referência

Checagem do tipo de parâmetros

- Importante para a confiabilidade
- FORTRAN 77 e C original: não tinha
- Pascal e Java: sempre exigiram
- ANSI C e C++: tem
- Algumas linguagem novas (Perl, JavaScript, PHP) não exigem checagem de tipos
- Python e Ruby: as variáveis não têm tipos e, portanto, a checagem de tipos dos parâmetros também não é possível

Arrays multi-d como parâmetros

- Em algumas linguagens, se um array multidimensional é passado como parâmetro para um subprograma, o compilador precisa saber o tamanho do array
- Em C e C++ deve-se informar todos os tamanhos, exceto o primeiro!
  - Diminui a flexibilidade
  - Solução: usar ponteiros
- Em Java e C# os arrays têm uma propriedade (length) que pode ser usada

Arrays multi-d como parâmetros

```c
#include <stdio.h>

int somar_elem_matriz(int matriz[][5]);

int main(void) {
    int matriz[5][5];
    for (int l = 0; l < 5; ++l) {
        for (int c = 0; c < 5; ++c) {
            matriz[l][c] = l + c;
        }
    }
    printf("%d", somar_elem_matriz(matriz));
    return 0;
}

int somar_elem_matriz(int matriz[][5]) {
    int resultado = 0;
    for (int l = 0; l < 5; ++l) {
        for (int c = 0; c < 5; ++c) {
            resultado += matriz[l][c];
        }
    }
    return resultado;
}
```

Arrays multi-d como parâmetros

```c
#include <stdio.h>

int somar_elem_matriz(int matriz[][5][5]);

int main(void) {
    int matriz[5][5][5];
    for (int l = 0; l < 5; ++l) {
        for (int c = 0; c < 5; ++c) {
            for (int p = 0; p < 5; ++p) {
                matriz[l][c][p] = l + c + p;
            }
        }
    }
    printf("%d", somar_elem_matriz(matriz));
    return 0;
}

int somar_elem_matriz(int matriz[][5][5]) {
    int resultado = 0;
    for (int l = 0; l < 5; ++l) {
        for (int c = 0; c < 5; ++c) {
            for (int p = 0; p < 5; ++p) {
                resultado += matriz[l][c][p];
            }
        }
    }
    return resultado;
}
```

Subprogramas Sobrecarregados

- Um subprograma sobrecarregado é aquele que tem o mesmo nome em dois ou mais subprogramas diferentes no mesmo ambiente de referenciamento, mas protocolos únicos (diferentes)
- Diversas linguagens já incluem subprogramas sobrecarregados pré-definidos
- C não suporta subprogramas carregados (“macete” com ponteiros)
- C++ suporta!

Subprogramas Sobrecarregados

```c
#include <stdio.h>

int soma(int a, int b);

double soma(double a, double b);

int main(void) {
    printf("%d\n", soma(3, 4));
    printf("%f\n", soma(3.3, 4.4));
}

int soma(int a, int b) {
    return a + b;
}

double soma(double a, double b) {
    return a + b;
}
```

Subprogramas Genéricos

- Um subprograma **genérico** (também chamado de subprograma **polimórfico**) executa com parâmetros de tipos diferentes em diferentes execuções
- Fornecem uma forma de polimorfismo ad hoc
- Suportados em diversas linguagens: C++, Java
- Difícil de explicar, fácil de entender com exemplos

Subprogramas Genéricos

```c
#include <stdio.h>
#define MAX 3

int pilha[MAX];
int topo = -1;

void push(int n, int p_pilha[]);
int pop(int[]);

int main(void) {
    push(3, pilha);
    push(5, pilha);
    printf("%d", pop(pilha));
    return 0;
}
```

```c
void push(int n, int p_pilha[]) {
    if (topo == MAX - 1) {
        printf("%s", "Pilha cheia.");
    } else {
        ++topo;
        pilha[topo] = n;
    }
}

int pop(int p_pilha[]) {
    if (topo == -1) {
        printf("%s", "Pilha vazia.");
        return -1;
    } else {
        int val = p_pilha[topo];
        --topo;
        return val;
    }
}
```

Subprogramas Genéricos

```
public class Pilha<Item> {
    private final Item[] a;
    private int n;

    public Pilha(int capacidade) {
        a = (Item[]) new Object[capacidade];
    }

    public void push(Item item) {
        a[++n] = item;
    }

    public Item pop() {
        return a[--n];
    }
}
```

Sobrecarga de operadores pelo usuár.

- Algumas linguagens permitem que o usuário defina uma nova sobrecarga para os operadores (Ada, C++, Python, Ruby, ...)

- Exemplo em Python

```
def __add__ (self, second):
  return Complex(self.real + second.real, self.imag +
   second.imag)
```

Fechamento (closure)

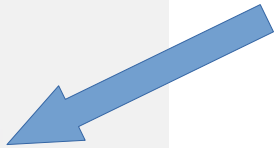
- Um **fechamento** (**closure**) é um subprograma e o ambiente de referenciamento no qual o subprograma foi definido
  - Uma closure dá acesso ao escopo de uma função externa a partir de uma função interna
  - Linguagem de escopo estático que não permitem subprograma aninhados não precisam de fechamentos
  - Fechamentos somente são necessários se um subprograma puder acessar variáveis em escopos aninhados e ele puder ser chamado de qualquer lugar
  - Para suportar fechamentos a linguagem pode precisar fornecer extensões ilimitadas para variáveis (acesso à variáveis que não estão “vivas”)

Fechamento (closure)

```
function init() {
  var name = "Mozilla"; // name é uma variável local
  function displayName() {
    // displayName() é a função interna
    console.log(name); // usa a variável declarada na função pai
  }
  displayName();
}
init();
```

Fechamento (closure)

```
function makeFunc() {
  var name = "Mozilla";
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```



Note que a função `displayName` foi retornada ANTES de ser executada!

Variáveis locais em funções (como a `name`) só existem enquanto a função executa. Então por que o código funciona?

`myFunc` se tornou um fechamento: é a função retornada E O AMBIENTE onde ela foi definida.

Fechamento (closure)

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

print(add5(2));  // 7
print(add10(2)); // 12
```

As funções `add5` e `add10` são closures.

Compartilham o mesmo corpo de definição de função mas armazenam diferentes ambientes.

No ambiente da `add5`, por exemplo, x equivale a 5, enquanto na `add10` o valor de x é 10.

Fechamento (closure)

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}
h2 {
  font-size: 1.2em;
}
```

```
<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

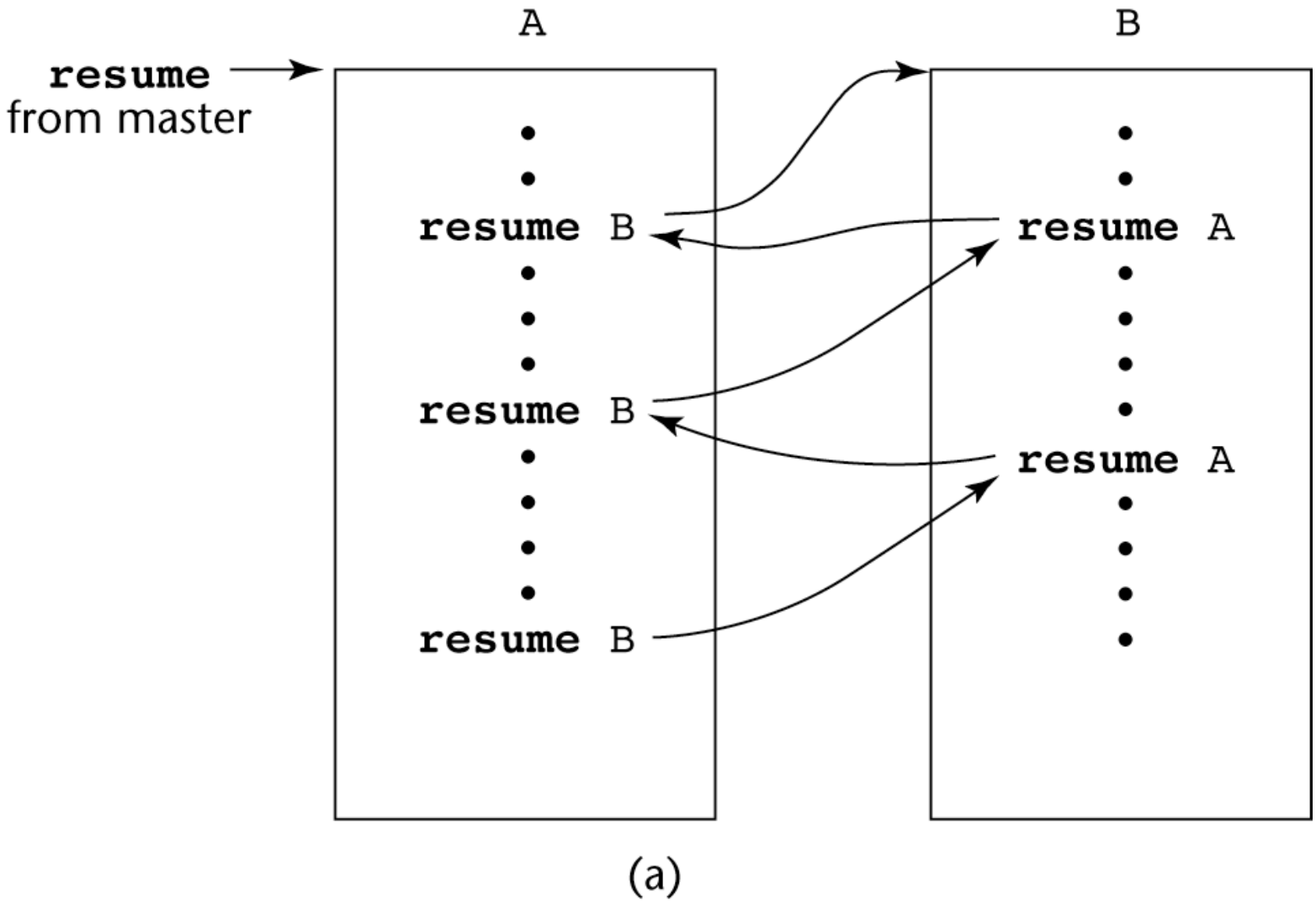
```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);
```

```
document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

Corrotinas

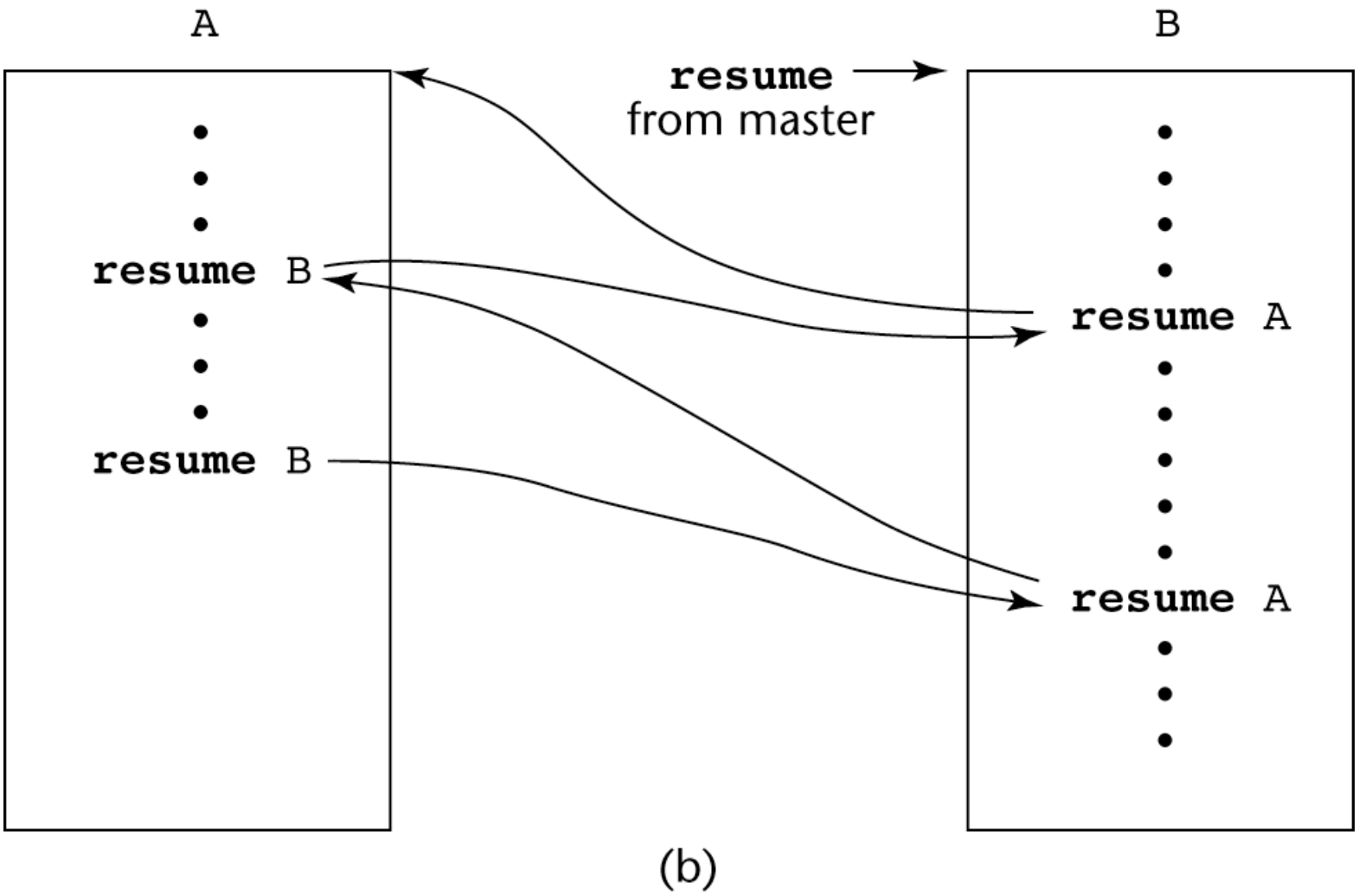
- Uma **corrotina** é um subprograma que tem **múltiplas entradas** e que controla por si mesma essas entradas. Suportadas diretamente só em Lua
- Não existe uma relação “mestre-escravo”, quem chamou e quem foi chamado são mais igualitários. São também chamadas de **retomada** (resume).
- A primeira chamada de uma corrotina começa em seu início mas chamadas subseqüentes começam exatamente no ponto após a última execução
- Corrotina geralmente retomam umas às outras, possivelmente para sempre
- São um meio de exeução quasi-concorrente: a execução é intercalada mas no sobreposta (só uma está em execução a qualquer momento)

Funcionamento das Corrotinas

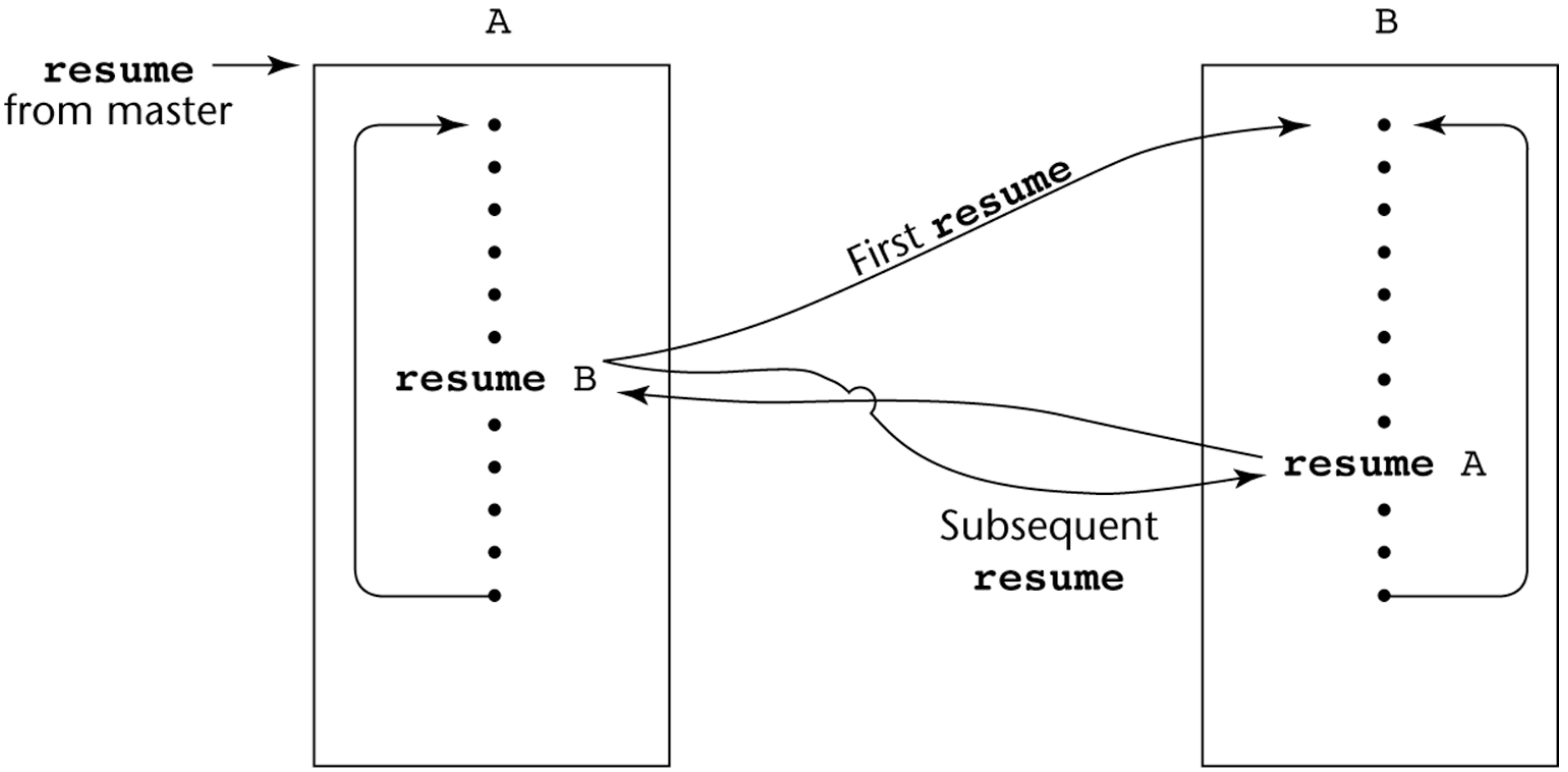


(a)

Funcionamento das Corrotinas



Funcionamento das Corrotinas



Importante!

- A definição de um subprograma descreve as ações realizadas por esse subprograma
- Subprogramas podem ser funções ou procedimentos
- Variáveis locais nos subprocedimentos podem ser dinâmicas de pilha (stack-dynamic) ou estáticas (static)
- Três métodos de passar parâmetros: entrada (in), saída (out) e entrada/saída (inout)
- Algumas linguagens permitem sobrecarga de subprogramas e/ou de operadores
- Algumas linguagem permitem sugprogramas genéricos
- Um fechamento é um subprograma e seu ambiente de referenciamento
- Corrotina é um subprograma com múltimas entradas