Paradigmas de Linguagens de Programação

Subprogramas

Introdução

- Subprogramas são **elementos fundamentais dos programas** e estão entre os **conceitos mais importantes** no projeto de linguagens de programação.
- Os subprogramas são conjuntos de sentenças que definem computações parametrizadas.
- Permitem **abstrair processos,** ou seja, nomear e reutilizar partes de código.
- O projeto de subprogramas envolve:
  - Métodos de **passagem de parâmetros**
  - **Ambientes de referência local**
  - **Sobrecarga**, **apelidos** e **efeitos colaterais**
  - Subprogramas **genéricos**, **fechamentos** e **corrotinas**

Introdução

- Dois recursos de abstração podem ser incluídos em linguagens de programação:
  - **Abstração de processos:** modelada pelos **subprogramas**.
  - **Abstração de dados:** modelada pelos **tipos de dados** e estruturas.
- Historicamente, as linguagens de alto nível focavam apenas na **abstração de processos**.
- A partir dos anos 1980, reconheceu-se que a **abstração de dados** é igualmente essencial.

Introdução

- O primeiro computador programável, a **Máquina Analítica de Babbage** (1840), já previa a **reutilização** de coleções de instruções.
- Essa ideia é o embrião do **subprograma moderno**:
  - Um bloco de sentenças reutilizável.
  - Chamada por meio de uma **instrução única**
  - Promove **reuso**, **economia de memória** e **redução de tempo de codificação**.

Introdução

- A principal vantagem é o **reuso de código** e a **abstração de detalhes de implementação**.
- Detalhes da computação são substituídos por uma **sentença de chamada**.
- Aumenta a **legibilidade e modularidade** dos programas, ao enfatizar a **estrutura lógica** em vez dos detalhes de baixo nível.

Introdução

- Os **métodos** das linguagens **orientadas a objetos** são **intimamente relacionados** aos **subprogramas**
  - Compartilham recursos/mecanismos como **parâmetros** e **variáveis locais**.

- As principais diferenças estão:
  - No **modo como são chamados**, e
  - Nas **associações com classes e objetos**.

Fundamentos dos Subprogramas

Características gerais dos subprogramas

- Todos os subprogramas (discutidos no livro), exceto as corrotinas, têm as seguintes características:
  - **Cada subprograma tem um único ponto de entrada.**
  - **A unidade de programa chamadora é suspensa durante a execução do subprograma chamado**, o que significa que existe apenas um subprograma em execução em determinado momento.
  - **Quando a execução do subprograma termina, o controle sempre retorna para o chamador.**
- As alternativas a isso resultam em **corrotinas** e em **unidades concorrentes**.
- Em sua maioria, os subprogramas têm **nomes**, embora alguns sejam **anônimos**.

Fundamentos dos Subprogramas

Definições Básicas

- A **definição de um subprograma** descreve a **interface** e as **ações** da abstração.
- Uma **chamada de subprograma** é o pedido explícito para que um subprograma específico seja executado.
- Diz-se que um subprograma está **ativo** se, depois de ser chamado, iniciou a execução mas ainda não a concluiu.
- Os dois tipos fundamentais de subprogramas são:
  - **Procedimentos**
  - **Funções**

Fundamentos dos Subprogramas

Definições Básicas

- O **cabeçalho** é a **primeira parte da definição** do subprograma.
- Ele tem três finalidades principais:
  - Especificar que a unidade sintática seguinte é a definição de um subprograma.
  - Fornecer um **nome** (caso o subprograma não seja **anônimo**).
  - Especificar uma **lista de parâmetros**.
- Nas linguagens que possuem mais de um tipo de subprograma, o tipo do subprograma normalmente é especificado com uma palavra especial.
  - **procedure** e **function** em Pascal e Ada.

Fundamentos dos Subprogramas

Definições Básicas

- Exemplo em Python

def adder(parâmetros):

- Cabeçalho de um subprograma em **Python** chamado *adder*.
- Em **Ruby**, os cabeçalhos também começam com **def**.
- Em **JavaScript**, um subprograma começa com **function**.

Fundamentos dos Subprogramas

Definições Básicas

- Exemplo em C

void adder(parâmetros)

- A palavra reservada **void** indica que o subprograma **não retorna valor**.
- O **corpo** do subprograma define suas ações.
- Em linguagens baseadas em C (e em JavaScript):
  - O corpo é delimitado por **chaves** { }.
- Em Ruby:
  - Uma **sentença end** finaliza o corpo.
- Em Python:
  - O corpo é **indentado** e termina quando a indentação retorna ao nível anterior.

Fundamentos dos Subprogramas

Definições Básicas

- Exemplo em Lua

  *function cube(x) return x * x * x end*

  *cube = function (x) return x * x * x end*

- Ambas são equivalentes.
- A primeira usa **sintaxe convencional**.
- A segunda mostra o **anonimato** das funções.

Fundamentos dos Subprogramas

Definições Básicas

- O **perfil de parâmetros** de um subprograma especifica:
  - O **número**, **ordem** e **tipo** dos parâmetros formais.
- O **protocolo de um subprograma** é o perfil mais o **tipo de retorno** (se houver).
- Em linguagens com tipos, o protocolo deve estar **declarado** antes do uso.

Fundamentos dos Subprogramas

Definições Básicas

- Subprogramas podem ter **declarações** ou **definições.**
- C e C++ exigem **declarações de função** (protótipos) antes do uso.
- Outras linguagens (como Python, Ruby, Lua, JavaScript):
  - **Não exigem declarações**, pois não há restrição quanto à ordem de definição e chamada.

Fundamentos dos Subprogramas

Parâmetros

- Normalmente, os **subprogramas descrevem computações**.
- Há duas maneiras pelas quais um subprograma ou método pode acessar os dados que deve processar:
  - **Acesso direto a variáveis não locais**, ou
  - **Passagem de parâmetros**.
- Os dados passados por meio de parâmetros são acessados como **nomes locais** ao subprograma.
- A passagem de parâmetros é **mais flexível** que o acesso direto a variáveis não locais.

Fundamentos dos Subprogramas

Parâmetros

- Um subprograma com acesso por meio de parâmetros é uma **computação parametrizada**.
- Ele pode operar sobre **quaisquer dados** recebidos por meio dos parâmetros.
- O acesso direto a variáveis não locais pode **reduzir a confiabilidade**, pois essas variáveis acabam sendo visíveis além do necessário.
- O uso sistemático de variáveis não locais **dificulta o isolamento** e **a depuração**.

Fundamentos dos Subprogramas

Parâmetros

- **Métodos** também podem acessar dados externos por meio de referências e parâmetros não locais.
- Contudo, o principal dado de um **método** é o **objeto** por meio do qual ele é chamado.
- O acesso de um método a variáveis de classe está ligado ao conceito de **dados não locais**.
- Alterações nesses dados podem **afetar o estado de outros objetos**, tornando o método **menos confiável**.

Fundamentos dos Subprogramas

Parâmetros

- Linguagens funcionais puras (como **Haskell**) **não possuem dados mutáveis**.
- Funções não alteram o ambiente de execução:
  - Recebem valores,
  - Efetuam cálculos,
  - Retornam o valor resultante (ou uma função).
- Assim, **não há variáveis não locais** nem **efeitos colaterais**.

Fundamentos dos Subprogramas

Parâmetros

- Os **nomes de parâmetros declarados** no cabeçalho de um subprograma são chamados de **parâmetros formais**.
- As **expressões passadas** na chamada são chamadas de **parâmetros reais**.
- A correspondência entre ambos é chamada de **vinculação de parâmetros reais e formais**.
- Alguns autores chamam os **parâmetros reais** de **argumentos** e os **parâmetros formais** apenas de **parâmetros**.

Fundamentos dos Subprogramas

Parâmetros

- Em muitas linguagens, os parâmetros reais são vinculados aos formais **pela posição**:
  - O **primeiro parâmetro real** é vinculado ao **primeiro formal**, e assim por diante.
- São chamados de **parâmetros posicionais**.
- Método **eficiente e seguro**, desde que as listas de parâmetros sejam relativamente curtas.

Fundamentos dos Subprogramas

Parâmetros

- Quando as listas são longas, a correspondência posicional pode gerar erros.
- Linguagens como **Python** permitem o uso de **parâmetros de palavra-chave**, em que o nome do parâmetro formal é explicitado na chamada.
- Exemplo:

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

- Define parâmetros formais length, list e sum.

Fundamentos dos Subprogramas

Parâmetros

- Em Python, é possível misturar parâmetros **posicionais** e **de palavra-chave**:

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

- Regra: após um parâmetro de palavra-chave, todos os demais devem ser de palavra-chave.

Fundamentos dos Subprogramas

Parâmetros

- Em linguagens como **Python**, **Ruby**, **C++** e **PHP**, parâmetros formais podem ter **valores padrão**.
- O valor padrão é usado quando **nenhum argumento real** é passado.
- Exemplo em Python:

```
def compute_pay(income, exemptions = 1, tax_rate)
```

- Pode ser chamada como:

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```

Fundamentos dos Subprogramas

Parâmetros

- C++ não oferece suporte para parâmetros de palavra-chave
- Os parâmetros padrão devem aparecer **por último**.

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

- Chamada válida:

```
pay = compute_pay(20000.0, 0.15);
```

Fundamentos dos Subprogramas

Parâmetros

- Algumas linguagens como C, C++, Perl, Python, JavaScript e Lua permitem **número variável de parâmetros**:
- Exemplo em **C**:

```
int printf(const char *format, ...);
printf("%d %f", x, y);
```

- O número de parâmetros é determinado em tempo de execução e apesar de útil e conveniente é propenso a erros.
- Exemplo em Python:

```
def report(*args, **kwargs):
report(10, 20, user=”user1", ok=True)
```

Fundamentos dos Subprogramas

Parâmetros

- **C#** permite que os métodos aceitem um número variável de parâmetros, desde que sejam do mesmo tipo.
- O modificador **params** permite passar **vetores** ou **listas** de valores.

```
public void DisplayList(params int[] list) {
    foreach (int next in list) {
        Console.WriteLine("Next value {0}", next);
    }
}
```

Fundamentos dos Subprogramas

Parâmetros

- Se *DisplayList* está definido para a classe *MyClass* e temos as declarações:

```
Myclass myObject = new Myclass;
int[] myList = new int[6] {2, 4, 6, 8, 10, 12};
```

- *DisplayList* poderia ser chamada com um dos seguintes:

```
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3 * x - 1, 17);
```

Fundamentos dos Subprogramas

Parâmetros

- Ruby suporta uma **configuração flexível** de parâmetros.
- Permite **misturar argumentos posicionais, pares chave-valor e vetores**.
- **Não há obrigatoriedade** de que todos os argumentos sejam do mesmo tipo.
- Exemplo:

```
list = [2, 4, 6, 8]
def tester(p1, p2, p3, *p4)
  . . .
end . . .
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

Fundamentos dos Subprogramas

Parâmetros

- Dentro de *tester*, os valores de seus parâmetros formais são os seguintes:

```
p1 is 'first'
p2 is {mon => 72, tue => 68, wed => 59}
p3 is 2
p4 is [4, 6, 8]
```

- Python aceita parâmetros semelhantes aos de Ruby.

Fundamentos dos Subprogramas

Parâmetros

- Lua usa **reticências (...)** para representar parâmetros variáveis.

```
function multiply (. . .)
  local product = 1
  for i, next  in  ipairs{. . .}  do
    product = product * next
   end return  sum
end
```

- *ipairs* é um iterador para vetores e . . . é um vetor dos valores de parâmetro real.

Fundamentos dos Subprogramas

Parâmetros

- Outro exemplo em Lua

```
function DoIt (. . .)
  local a, b, c = . . .
  . . .
end
```

```
doit(4, 7, 3)
```

- Nesse exemplo, a, b e c serão inicializados na função com os valores 4, 7 e 3, respectivamente.
- O parâmetro de três pontos não precisa ser o único, ele pode aparecer no final de uma lista de parâmetros formais nomeados.

Fundamentos dos Subprogramas

Procedimentos e funções

- Existem **duas categorias distintas de subprogramas**:
  - **procedimentos** e **funções**.
- Ambas são **estratégias para ampliar a linguagem** e definem **computações parametrizadas**.
- As **funções** retornam valores; os **procedimentos**, não.
- Em linguagens que não incluem procedimentos como forma distinta, as funções podem ser definidas de forma a **não retornarem valores**, sendo usadas **como procedimentos**.

Fundamentos dos Subprogramas

Procedimentos e funções

- As computações de um procedimento são **representadas por sentenças de chamada simples**.
- Exemplo:
  - Se uma linguagem não tiver uma sentença de ordenação, um usuário pode criar um **procedimento** para ordenar vetores e usá-lo no lugar da sentença inexistente.

ordena(vetor)

- Linguagens antigas como **Fortran** e **Ada** suportam procedimentos nativamente.

Fundamentos dos Subprogramas

Procedimentos e funções

- Os procedimentos podem produzir resultados na unidade de programa chamadora de dois modos:
  - Se houver **variáveis não locais visíveis** tanto no procedimento quanto na unidade chamadora, o procedimento poderá **alterá-las**.
  - Se o procedimento tiver **parâmetros formais que permitam transferência de dados** para o chamador, esses parâmetros poderão **ser alterados**.

Fundamentos dos Subprogramas

Procedimentos e funções

- Funções são chamadas por meio de **expressões** com seus nomes e parâmetros.
- O valor retornado **substitui a chamada** na expressão.
- Semanticamente modeladas como funções matemáticas.
- Por exemplo:
  - o valor da expressão f(x) é qualquer valor produzido por f quando chamada com o parâmetro x.
- Para uma função que não produz efeitos colaterais, o valor retornado é seu único efeito.

Fundamentos dos Subprogramas

Procedimentos e funções

- Funções são chamadas por meio de **expressões** com seus nomes e parâmetros.
- O valor retornado **substitui a chamada** na expressão.
- Semanticamente modeladas como funções matemáticas.
- Por exemplo:
  - o valor da expressão f(x) é qualquer valor produzido por f quando chamada com o parâmetro x.
- Para uma função que não produz efeitos colaterais, o valor retornado é seu único efeito.

Fundamentos dos Subprogramas

Procedimentos e funções

- Exemplo em C++:

```
float power(float base, float exp)
```

- o qual poderia ser chamado com:

```
result = 3.4 * power(10.0, x)
```

Fundamentos dos Subprogramas

Questões de Projeto para Subprogramas

- Subprogramas são **estruturas complexas**, e há muitas questões envolvidas em seu projeto.
- Um problema óbvio é a **escolha do método de passagem de parâmetros**.
- A ampla variedade de estratégias existentes reflete a diversidade de opiniões sobre o assunto.
- Questão relacionada: **os tipos dos parâmetros formais devem ser verificados?**

Fundamentos dos Subprogramas

Questões de Projeto para Subprogramas

- Subprogramas podem ter definições aninhadas, podem ser sobrecarregados e genéricos.
- **Subprograma sobrecarregado**: tem **o mesmo nome** de outro no mesmo ambiente de referenciamento.
- **Subprograma genérico**: pode operar sobre **diferentes tipos de dados**.
- **Fechamento (closure)**: subprograma aninhado + ambiente de referenciamento, permitindo que seja chamado de qualquer lugar do programa.

Fundamentos dos Subprogramas

Questões de Projeto para Subprogramas

- As variáveis locais são alocadas estática ou dinamicamente?
- Definições de subprograma podem aparecer em outras definições de subprograma?
- Qual método (ou métodos) de passagem de parâmetros é usado?
- Os tipos dos parâmetros são verificados em relação aos tipos dos parâmetros formais?
- Se subprogramas podem ser passados como parâmetros e podem ser aninhados, qual é o ambiente de referenciamento de um subprograma passado?

Fundamentos dos Subprogramas

Questões de Projeto para Subprogramas

- São permitidos efeitos colaterais funcionais?
- Quais tipos de valores podem ser retornados de funções?
- Quantos valores podem ser retornados de funções?
- Os subprogramas podem ser sobrecarregados?
- Os subprogramas podem ser genéricos?
- Se a linguagem permite subprogramas aninhados, fechamentos são suportados?

Ambientes de Referenciamento Local

Variáveis locais

- Subprogramas podem definir suas **próprias variáveis**, criando um **ambiente local de referenciamento**.
- As variáveis definidas dentro de subprogramas são chamadas de **variáveis locais**.
- O **escopo** das variáveis locais normalmente é o **corpo do subprograma** onde foram declaradas.
- Quando o subprograma termina, essas variáveis **deixam de existir**, a menos que sejam **estáticas**.

Ambientes de Referenciamento Local

Variáveis locais

- **Variáveis locais estáticas** são criadas uma única vez e mantêm seu valor durante toda a execução do programa.
  - Vantagens: acesso rápido e preservação do valor entre chamadas.
  - Desvantagens: não suportam recursão e ocupam memória por mais tempo.
- **Variáveis locais dinâmicas da pilha** são criadas a cada chamada e destruídas ao final da execução.
  - Vantagens: permitem recursão e uso mais eficiente da memória.
  - Desvantagens: acesso um pouco mais lento e perda do valor anterior.
- **Subprogramas recursivos** precisam de variáveis locais dinâmicas da pilha.
- **Variáveis estáticas** são úteis quando o subprograma precisa lembrar resultados anteriores.

Ambientes de Referenciamento Local

Variáveis locais

- Exemplo em C
- A variável sum mantém seu valor entre chamadas da função, pois é estática.
- A variável count é criada e destruída a cada execução, pois é dinâmica.
- Assim, a função acumula o valor total de sum ao longo das chamadas.

```
int adder(int list[], int listlen) {
   static int sum = 0; //local estática
   int count; // local dinâmica da pilha

   for (count = 0; count < listlen; count++)
      sum += list[count];

   return sum;
}
```

Ambientes de Referenciamento Local

Variáveis locais

- **C e C++**: variáveis locais são dinâmicas da pilha por padrão.
  - Podem ser declaradas como estáticas com a palavra-chave **static**.
- **Java e C#**: todas as variáveis locais são dinâmicas da pilha.
- **Python**: variáveis declaradas em métodos são locais e dinâmicas da pilha.
  - Para acessar variáveis externas, usa-se global ou nonlocal.
- **Lua**: todas as variáveis são globais por padrão.
  - Para criar variáveis locais, usa-se a palavra-chave local:

Ambientes de Referenciamento Local

Subprogramas Aninhados

- A ideia de **aninhar subprogramas** se originou com **ALGOL 60**.
- O objetivo era permitir uma **hierarquia de lógica e escopos** dentro do programa.
- Se um subprograma é necessário apenas dentro de outro, pode ser **declarado dentro dele** e **ocultado do restante do programa**.
- Linguagens que permitem subprogramas aninhados geralmente utilizam **escopo estático**.
- Isso permite uma **estruturação hierárquica** e o **acesso a variáveis não locais** dos subprogramas envolventes.

Ambientes de Referenciamento Local

Subprogramas Aninhados

- Por muito tempo, apenas as descendentes de **ALGOL 60** (como **ALGOL 68**, **Pascal** e **Ada**) permitiam subprogramas aninhados.
- Linguagens descendentes de **C** (como C, C++, Java e C#) **não permitem** aninhamento de subprogramas.
- Recentemente, linguagens modernas voltaram a permitir essa característica:
  - **JavaScript, Python, Ruby e Lua.**
- Além disso, a **maioria das linguagens funcionais** também permite **subprogramas aninhados**.

Métodos de Passagem de Parâmetros

- Métodos de passagem de parâmetros determinam **como os dados são transmitidos** entre subprogramas.
- O objetivo é especificar **de que modo** os parâmetros formais e reais se comunicam.
- Três principais aspectos são considerados:
  - **Modelos semânticos** (o significado da passagem).
  - **Modelos de implementação** (como isso é feito na prática).
  - **Decisões de projeto** (como cada linguagem escolhe um modelo).

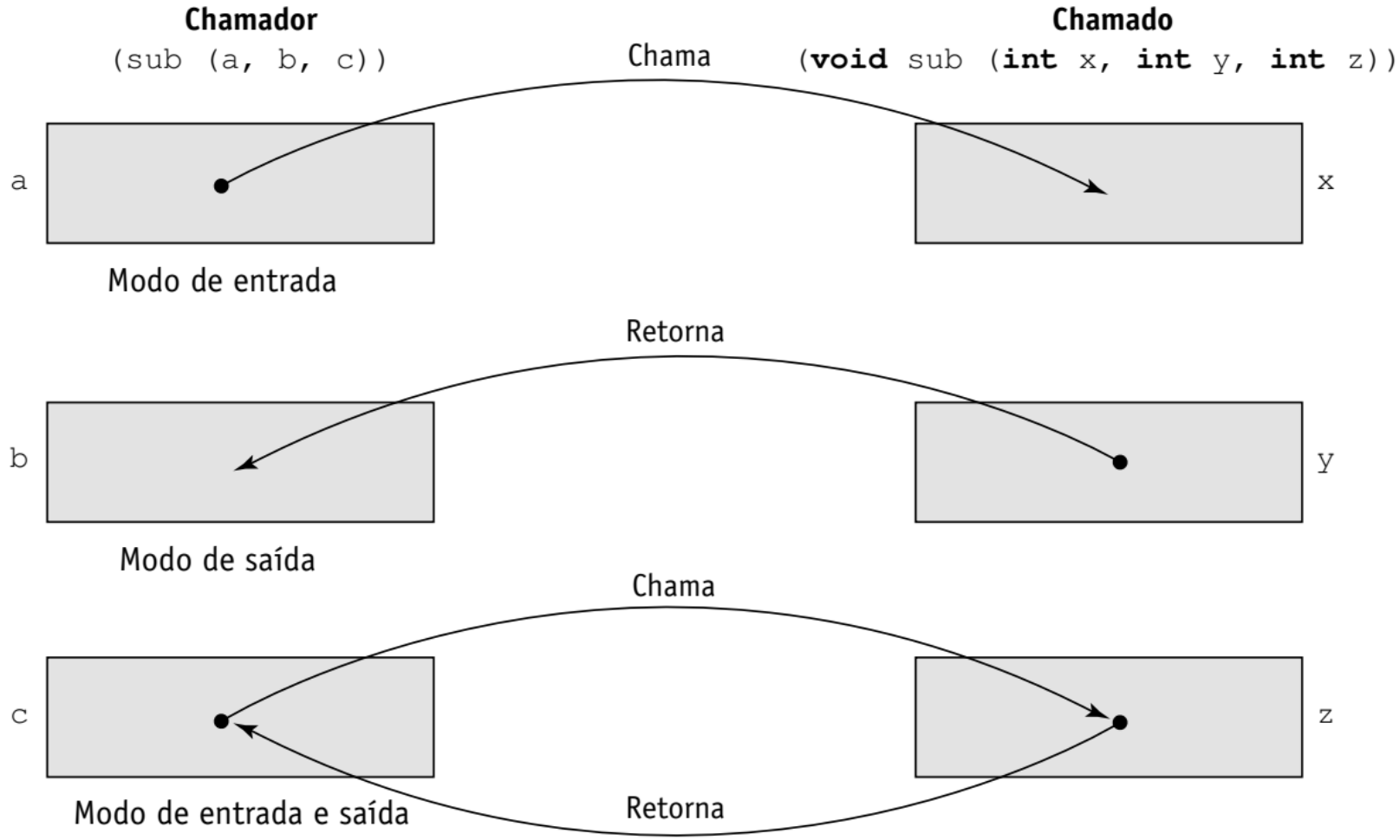
Métodos de Passagem de Parâmetros

Modelos Semânticos de Passagem de Parâmetros

- Os parâmetros formais podem seguir **três modelos semânticos básicos**:
  - **Modo de entrada** (*in mode*)
    - O subprograma **recebe dados** do chamador, mas **não devolve alterações**.
  - **Modo de saída** (*out mode*)
    - O subprograma **envia dados** de volta ao chamador, sem receber valores iniciais.
  - **Modo de entrada e saída** (*inout mode*)
    - O parâmetro é usado **para entrada e saída**, recebendo um valor inicial e retornando o valor atualizado.

Métodos de Passagem de Parâmetros

Modelos Semânticos de Passagem de Parâmetros



Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por valor

- Quando um parâmetro é **passado por valor**, o valor do parâmetro real é usado para **inicializar o parâmetro formal correspondente**.
- Esse parâmetro formal **atua como uma variável local** no subprograma, implementando assim a **semântica de modo de entrada (*in mode*)**.

Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por valor

- Normalmente, a **passagem por valor é implementada por cópia**, pois os acessos são mais eficientes com essa estratégia.
- Também poderia ser implementada pela transmissão de um **caminho de acesso** ao valor no chamador, mas isso exigiria que o valor estivesse em **uma célula protegida contra escrita** (somente leitura).
  - Nem sempre é simples impor essa proteção.

Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por valor

- **Vantagem da passagem por valor:**
  - Para tipos **escalares**, é **rápida** e **eficiente** em termos de acesso.
- **Desvantagem:**
  - Requer **armazenamento adicional** para o parâmetro formal (a cópia).
  - O parâmetro real **não é alterado**.
  - Pode ser **ineficiente** quando aplicado a **vetores grandes**, devido ao custo da cópia.

Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por valor

- **Vantagem da passagem por valor:**
  - Para tipos **escalares**, é **rápida** e **eficiente** em termos de acesso.
- **Desvantagem:**
  - Requer **armazenamento adicional** para o parâmetro formal (a cópia).
  - O parâmetro real **não é alterado**.
  - Pode ser **ineficiente** quando aplicado a **vetores grandes**, devido ao custo da cópia.

Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por Referência

- é um modelo de implementação para **parâmetros de modo de entrada e saída**.
- Em vez de copiar valores de dados, é transmitido **um caminho de acesso** (geralmente um **endereço**) ao subprograma chamado.

Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por Referência

- Isso fornece ao subprograma acesso direto à **célula de memória** que armazena o parâmetro real.
- Assim, o subprograma pode **acessar e modificar** o valor da variável no programa chamador.
- O parâmetro real é, portanto, **compartilhado** com o subprograma chamado.

Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por Referência

- **Vantagens**
  - Processo de passagem **eficiente em tempo e espaço**.
  - Nenhuma cópia é necessária, não há armazenamento duplicado.
- **Desvantagens**
  - O acesso indireto é **mais lento** que na passagem por valor.
  - Há risco de **alterações acidentais** no parâmetro real.
  - Pode criar **apelidos** (*aliases*), isto é, diferentes nomes que referenciam o mesmo endereço de memória.
    - Isso prejudica a **legibilidade** e **confiabilidade** do programa.
    - Torna a **verificação de erros mais difícil**.

Métodos de Passagem de Parâmetros

Modelos de implementação de passagem de parâmetros

Passagem por Referência

- Exemplo em C++

void fun(int &first, int &second);

fun(total, total);

- first e second tornam-se **apelidos** (**aliases**) da mesma variável (*total*).
- Alterar *first* também altera *second*.

Métodos de Passagem de Parâmetros

Implementação de métodos de passagem de parâmetros

- Nas linguagens modernas, a **pilha de tempo de execução** é usada para gerenciar chamadas e transmitir parâmetros.
- Cada chamada de subprograma cria um **novo registro de ativação** na pilha.
- Essa pilha armazena os **valores ou endereços** dos parâmetros, conforme o método usado.

Métodos de Passagem de Parâmetros

Implementação de métodos de passagem de parâmetros

Passagem por Valor

- O **valor do parâmetro real** é **copiado** para o subprograma.
- O parâmetro formal atua como uma **variável local independente**.
- Alterações no subprograma **não afetam** o valor original.
- Implementação: o **valor** é armazenado na pilha.

Métodos de Passagem de Parâmetros

Implementação de métodos de passagem de parâmetros

Passagem por Referência

- O **endereço do parâmetro real** é transmitido ao subprograma.
- O subprograma acessa e pode **modificar o valor original**.
- Implementação: o **endereço** é armazenado na pilha.

Métodos de Passagem de Parâmetros

Métodos de Passagem de Parâmetros de Algumas Linguagens Comuns

- C usa **passagem por valor**.
- A **semântica de passagem por referência** é obtida pelo uso de **ponteiros como parâmetros**.
- O valor do ponteiro é disponibilizado à função chamada.
- A função pode alterar os dados originais por meio desse caminho de acesso.
- C copiou esse modelo do ALGOL 68.

Métodos de Passagem de Parâmetros

Métodos de Passagem de Parâmetros de Algumas Linguagens Comuns

- Em C++**:**
- Parâmetros formais podem ser **ponteiros** ou **referências**.
  - Tipo especial: & → **tipo de referência**.
  - Desreferenciado implicitamente na função.
- Semântica de passagem por referência.
- Podem ser definidos como **constantes**, impedindo modificação

Métodos de Passagem de Parâmetros

Métodos de Passagem de Parâmetros de Algumas Linguagens Comuns

- Em C++**:**

```
void fun(const int &p1, int p2, int &p3) {
        p3 = p1 + p2;  // altera a variável real passada em p3
}
int a = 10, b = 20, c = 30;
fun(a, b, c);
```

- p1 é passado por referência constante, p2 por valor e p3 por referência.

Métodos de Passagem de Parâmetros

Métodos de Passagem de Parâmetros de Algumas Linguagens Comuns

- Em Java:
- Todos os parâmetros são passados **por valor**.
- Objetos são passados **por valor da referência**.
- O método recebe uma cópia da referência.
- Pode alterar o objeto, mas não trocar a referência.
- Escalares (como int, float) nunca são passados por referência.

Métodos de Passagem de Parâmetros

Métodos de Passagem de Parâmetros de Algumas Linguagens Comuns

- C#:
- Passagem por valor por padrão.
- Passagem por referência com o modificador **ref**:

void sumer(ref int oldSum, int newOne) { … }

sumer(ref sum, newValue);

- *oldSum* é passado por referência, *newOne* por valor.

Métodos de Passagem de Parâmetros

Métodos de Passagem de Parâmetros de Algumas Linguagens Comuns

- Python e Ruby: Passagem por Atribuição
- Todos os valores são **objetos**, e toda variável é uma **referência**.
- O valor do parâmetro formal é **atribuído ao parâmetro real**.
- A semântica é parecida com a **passagem por referência**, mas:
- Objetos imutáveis (ex: inteiros, strings) não são modificados.
- Objetos mutáveis (ex: listas, dicionários) podem ser alterados.

Métodos de Passagem de Parâmetros

Métodos de Passagem de Parâmetros de Algumas Linguagens Comuns

- Python e Ruby: Passagem por Atribuição
- Exemplo:

```
x = 1
x = x + 1  # cria novo objeto, não altera o anterior


list = [3]
def f(lst): lst[0] = 47
f(list)   # list[0] agora é 47
```

Métodos de Passagem de Parâmetros

Verificação de Tipos de Parâmetros

- A confiabilidade do software exige que se verifique a consistência entre os tipos dos **parâmetros reais** e os **parâmetros formais**.
- Sem verificação de tipos, pequenos erros podem gerar falhas difíceis de diagnosticar.
- Exemplo:

```
result = sub1(1);  // parâmetro real é inteiro
```

Se o parâmetro formal de sub1 for double, o compilador deve detectar o erro.

Métodos de Passagem de Parâmetros

Verificação de Tipos de Parâmetros

- O tipo do parâmetro real é comparado ao tipo do parâmetro formal.
- Se forem diferentes, o compilador pode aplicar **coerção**, como converter int → double.
- Essa coerção é permitida quando há compatibilidade entre tipos numéricos.
- Se os tipos forem incompatíveis, o compilador gera erro de tipo.
- Em C e C++, todas as funções devem ter seus parâmetros declarados com tipos.

Métodos de Passagem de Parâmetros

Verificação de Tipos de Parâmetros

- Funções com **número variável de parâmetros**, como printf, não têm verificação de tipos completa:

int printf(const char format_string, . . .);*

printf("The sum is %d\n", sum);

Métodos de Passagem de Parâmetros

Verificação de Tipos de Parâmetros

- Em **C#**, coerções são aceitas apenas por valor, não por referência (ref).
  - O tipo de um parâmetro ref deve corresponder exatamente.
- Em **Python** e **Ruby**, não há verificação de tipos, pois são linguagens de **tipagem dinâmica**.

Métodos de Passagem de Parâmetros

Verificação de Tipos de Parâmetros

- Em **C#**, coerções são aceitas apenas por valor, não por referência (ref).
  - O tipo de um parâmetro ref deve corresponder exatamente.
- Em **Python** e **Ruby**, não há verificação de tipos, pois são linguagens de **tipagem dinâmica**.

Questões de Projeto para Funções

As questões de projeto a seguir são específicas para funções:

- Efeitos colaterais são permitidos?
- Quais tipos de valores podem ser retornados?
- Quantos valores podem ser retornados?

Questões de Projeto para Funções

Efeitos colaterais funcionais

- Funções com efeitos colaterais podem modificar variáveis fora de seu escopo.
- Em algumas linguagens, como **Ada**, isso é impedido: apenas parâmetros de **modo de entrada** são permitidos.
- Assim, funções Ada **não causam efeitos colaterais** por meio de parâmetros ou variáveis globais.
- Outras linguagens imperativas (C, C++, etc.) permitem efeitos colaterais por meio de parâmetros passados **por referência**.
- Funções funcionais puras (como em **Haskell**) não têm variáveis e, portanto, **não produzem efeitos colaterais**.

Questões de Projeto para Funções

Tipos de valores retornados

- A maioria das linguagens imperativas restringe os tipos que podem ser retornados por funções.
- **C** permite retornar qualquer tipo, **exceto vetores e funções**, mas permite o retorno de **ponteiros para funções**.
- **C++** é semelhante a C, mas também permite retornar **tipos definidos pelo usuário** (como classes).
- **Ada**, **Python**, **Ruby** e **Lua** permitem o retorno de **valores de qualquer tipo**.

Questões de Projeto para Funções

Número de valores retornados

- A maioria das linguagens permite **apenas um valor de retorno**.
- **Ruby, Python** e **Lua** permitem retorno de **múltiplos valores**, separados por vírgulas:

return 3, sum, index

a, b, c = fun()

Em **F#**, vários valores podem ser retornados como uma **tupla**.

Subprogramas Sobrecarregados

- Um **subprograma sobrecarregado** é aquele cujo nome é igual ao de outro subprograma no mesmo ambiente de referenciamento.
- Cada versão tem um **perfil de parâmetros exclusivo** (número, ordem ou tipos diferentes).
- O significado de uma chamada é determinado pela **lista de parâmetros reais** e, possivelmente, pelo **tipo do valor retornado**.
- A implementação do mecanismo de sobrecarga varia entre as linguagens.

Subprogramas Sobrecarregados

- Em C++, Java e C#
- Essas linguagens incluem subprogramas sobrecarregados **predefinidos**, e permitem que o programador defina outros.
- O compilador tenta identificar qual versão do subprograma deve ser chamada, verificando:
  - O número de parâmetros,
  - Seus tipos e a ordem em que aparecem.
- Quando ocorrem **coerções de tipos**, o processo pode se tornar ambíguo e complicado.

Subprogramas Sobrecarregados

- Os subprogramas sobrecarregados que têm parâmetros padrão podem levar a chamadas de subprograma ambíguas. Por exemplo, considere o código C++ a seguir:

```
void fun(float b = 0.0);
void fun();
. . .
fun();
```

- A chamada é ambígua e causará um erro de compilação.

Operadores Sobrecarregados Definidos pelo Usuário

- Operadores podem ser **sobrecarregados pelo usuário** em linguagens como **Ada, C++, Python e Ruby**.
- Isso permite que operadores (como +, -, *, /) tenham comportamento específico para **novos tipos definidos pelo programador**.

Operadores Sobrecarregados Definidos pelo Usuário

- Exemplo em Python
- uma classe Complex pode representar **números complexos** com membros real e imag.
- A expressão **x + y** é traduzida para **x.__add__(y)**.
- Em Python, a referência ao objeto atual deve ser enviada **explicitamente** como self.

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

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

Fechamentos

- **Fechamento (closure)** = **função + ambiente de referência** onde foi criada.
- Permite que uma função **“lembre” variáveis** do escopo onde foi definida, mesmo após esse escopo ter sido encerrado.
- Útil quando uma função interna precisa acessar dados da função externa.
- Muito comum em linguagens como **Python, JavaScript e Ruby**.

Fechamentos

- Exemplo em Python
- adder é uma **função interna** que **“fecha”** sobre a variável x.
- Mesmo depois que make_adder termina, adder **ainda lembra o valor de x**.
- Assim, add10 e add5 são duas **funções diferentes**, cada uma com seu próprio ambiente salvo.

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

add10 = make_adder(10)
add5 = make_adder(5)

print(add10(20))  # 30
print(add5(20))   # 25
```

Fechamentos

- Exemplo em Python
- Cada chamada de make_adder cria **um novo ambiente** com uma cópia de x.
- O fechamento mantém vivo esse ambiente,permitindo que adder continue acessando x.
- Em outras palavras, um fechamento é uma função **com memória** do contexto onde nasceu.

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

add10 = make_adder(10)
add5 = make_adder(5)

print(add10(20))  # 30
print(add5(20))   # 25
```

Paradigmas de Linguagens de Programação

Subprogramas