Subprogramas

Linguagens de Programação

Marco A L Barbosa

Conteúdo

Introdução

Introdução

- Abstração de processo
  - São os subprogramas
  - Economia de memória e tempo de programação
  - Aumento da legibilidade
- Abstração de dados
  - Capítulos 11 e 12

Fundamentos

Fundamentos

- Características gerais dos subprogramas
  - Cada subprograma tem um único ponto de entrada
  - Toda unidade de programa chamadora é suspensa durante a execução do subprograma chamado
  - O controle sempre retorna para o chamador quando a execução do subprograma chamado termina

Fundamentos

- Definições básicas
  - Uma **definição de subprograma** descreve a interface e as ações do subprograma
  - Uma **chamada de subprograma** é a solicitação explícita para executar o subprograma
  - Um subprograma está **ativo** se depois de chamado, ele iniciou a sua execução, mas ainda não a concluiu

Fundamentos

- Definições básicas
  - O **cabeçalho do subprograma** é a primeira parte da definição
    - Especifica o tipo (função, procedimento, etc)
    - Especifica o nome
    - Pode especificar a lista de parâmetros

Fundamentos

- Definições básicas
  - O **cabeçalho do subprograma** é a primeira parte da definição
    - Especifica o tipo (função, procedimento, etc)
    - Especifica o nome
    - Pode especificar a lista de parâmetros
  - Exemplos
  - Fortran: `Subrotine Adder(parameters)`
  - Ada: `procedure Adder(parameters)`
  - Python: `def adder(parameters):`
  - C: `void adder(parameters)`
  - Lua (funções são entidades de primeira classe)
    - `function cube(x) return x * x * x end`
    - `cube = function (x) return x * x * x end`

Fundamentos

- Definições básicas
  - O **perfil dos parâmetros** é o número, a ordem e o tipo dos parâmetros formais
  - O **protocolo** é o perfil dos parâmetros mais o tipo de retorno (em caso de funções)
  - Declarações
    - A declaração especifica o protocolo do subprograma, mas não as ações
    - Necessário em linguagens que não permitem referenciar subprogramas definido após o uso
    - Em C/C++ uma declaração é chamada de protótipo

Parâmetros

- Existem duas maneiras de um subprograma acessar os dados para processar

Parâmetros

- Existem duas maneiras de um subprograma acessar os dados para processar
  - Acesso direto as variáveis não locais
    - O acesso extensivo a variáveis não locais podem criar diversos problemas

Parâmetros

- Existem duas maneiras de um subprograma acessar os dados para processar
  - Acesso direto as variáveis não locais
    - O acesso extensivo a variáveis não locais podem criar diversos problemas
  - Passagem de parâmetros
    - Dados passados por parâmetro são acessados por nomes que são locais ao subprograma

Parâmetros

- Os parâmetros no cabeçalho do subprograma são chamados de **parâmetros formais**
- Os parâmetros passados em uma chamada de subprograma são chamados de **parâmetros reais**
- Vinculação entre os parâmetros reais e os parâmetros formais
  - A maioria das linguagens faz a vinculação através da posição (**parâmetros posicionais**): o primeiro parâmetro real é vinculado com o primeiro parâmetro formal, e assim por diante
    - Funciona bem quando o número de parâmetros é pequeno
  - Existem outras formas?

Parâmetros

- Parâmetros de palavras-chave (Ada, Fortran 95, Python)
  - Exemplo em Python

```
def soma(lista, inicio, fim):
    ...
soma(inicio = 1, fim = 2, lista = [4, 5, 6])
soma([4, 5, 6], fim = 1, inicio = 2)
```

Parâmetros

- Parâmetros de palavras-chave (Ada, Fortran 95, Python)
  - Exemplo em Python

```
def soma(lista, inicio, fim):
    ...
soma(inicio = 1, fim = 2, lista = [4, 5, 6])
soma([4, 5, 6], fim = 1, inicio = 2)
```

- Parâmetros com valor padrão (Python, Ruby, C++, Fortran 95, Ada)
  - Exemplo em Python

```
def compute_pay(income, exemptions = 1, tax_rate):
  ...
pay = compute_pay(20000.0, tax_rate = 0.15)
```

  - Exemplo em C++

```
float compute_pay(float income,
                  float tax_rate,
                  int exemptions = 1) { ... }
pay = compute_pay(20000.0, 0.15);
```

Parâmetros

- Número variável de parâmetros (C/C++/C#, Python, Java, Javascript, Lua, etc)
  - Exemplo em C#

```
public void DisplayList(params int[] list) {
    foreach (int next in list) {
        Console.WriteLine("Next value {0}", next);
    }
}
int[] list = new int[6] {2, 4, 6, 8, 10, 12};
DisplayList(list);
DisplayList(2, 4, 3 * x - 1, 17);
```

Parâmetros

- Número variável de parâmetros (C/C++/C#, Python, Java, Javascript, Lua, etc)
  - Exemplo em C#

```
public void DisplayList(params int[] list) {
    foreach (int next in list) {
        Console.WriteLine("Next value {0}", next);
    }
}
int[] list = new int[6] {2, 4, 6, 8, 10, 12};
DisplayList(list);
DisplayList(2, 4, 3 * x - 1, 17);
```

  - Exemplo em Python

```
def fun1(p1, p2, *p3, **p4):
    print 'p1 =', p1, 'p2 =', p2,
          'p3 =', p3, 'p4 =', p4

> fun1(2, 4, 6, 8, mon=68, tue=72, wed=77)
p1 = 2  p2 = 4  p3 = (6, 8)
p4 = {'wed': 77, 'mon': 68, 'tue': 72}
```

Procedimentos e funções

- Existem duas categorias de subprogramas
  - **Procedimentos** são coleções de instruções que definem uma computação parametrizada
    - Produzem resultados para a unidade chamadora de duas formas: através das variáveis não locais e alterando os parâmetros
    - São usados para criar novas instruções (sentenças)
  - **Funções** são baseadas no conceito matemático de função
    - Retorna um valor, que ó o efeito desejado
    - São usadas para criar novos operadores
    - Uma função sem efeito colateral é chamada de **função pura**

Questões de projeto referentes aos subprogramas

Questões de projeto referentes aos subprogramas

- As variáveis locais são alocadas estaticamente ou dinamicamente?
- As definições de subprogramas podem aparecer em outra definição de subprograma?
- Quais métodos de passagem de parâmetros são usados?
- Os tipos dos parâmetros reais são checados em relação ao tipo dos parâmetros formais?
- Se subprogramas podem ser passados como parâmetros e os subprogramas podem ser aninhados, qual é o ambiente de e referenciamento do subprograma passado?
- Os subprogramas podem ser sobrecarregados?
- Os subprogramas podem ser genéricos?

Ambientes de referência local

Ambientes de referência local

- Variáveis locais
  - São definidas dentro de subprogramas
  - Podem ser estáticas ou dinâmicas na pilha
    - Vantagens e desvantagens (cap 5 e seção 9.4.1)

Ambientes de referência local

- Variáveis locais
  - São definidas dentro de subprogramas
  - Podem ser estáticas ou dinâmicas na pilha
    - Vantagens e desvantagens (cap 5 e seção 9.4.1)
  - Poucas linguagens utilizam apenas vinculação estática

Ambientes de referência local

- Variáveis locais
  - São definidas dentro de subprogramas
  - Podem ser estáticas ou dinâmicas na pilha
    - Vantagens e desvantagens (cap 5 e seção 9.4.1)
  - Poucas linguagens utilizam apenas vinculação estática
  - Ada, Java e C# permitem apenas variáveis locais dinâmicas na pilha

Ambientes de referência local

- Variáveis locais
  - São definidas dentro de subprogramas
  - Podem ser estáticas ou dinâmicas na pilha
    - Vantagens e desvantagens (cap 5 e seção 9.4.1)
  - Poucas linguagens utilizam apenas vinculação estática
  - Ada, Java e C# permitem apenas variáveis locais dinâmicas na pilha
  - A linguagem C, permite o programado escolher

```
int adder(int list[], int listlen) {
  static in sum = 0;
  int count;
  for (count = 0; count < listlen; count++)
      sum += list[count]; return sum;
}
```

Ambientes de referência local

- Variáveis locais
  - São definidas dentro de subprogramas
  - Podem ser estáticas ou dinâmicas na pilha
    - Vantagens e desvantagens (cap 5 e seção 9.4.1)
  - Poucas linguagens utilizam apenas vinculação estática
  - Ada, Java e C# permitem apenas variáveis locais dinâmicas na pilha
  - A linguagem C, permite o programado escolher

```
int adder(int list[], int listlen) {
  static in sum = 0;
  int count;
  for (count = 0; count < listlen; count++)
      sum += list[count]; return sum;
}
```

- Subprogramas aninhados

Métodos de passagens de parâmetros

Métodos de passagens de parâmetros

- Um **método de passagem de parâmetro** é a maneira como os parâmetros são transmitidos para (ou/e do) subprograma chamado
- Os parâmetros formais são caracterizados por um de três modelos semânticos
  - Eles podem receber dados dos parâmetros reais (in mode)
  - Eles podem transmitir dados para os parâmetros reais (out mode)
  - Eles podem fazer ambos (inout mode)

Métodos de passagens de parâmetros

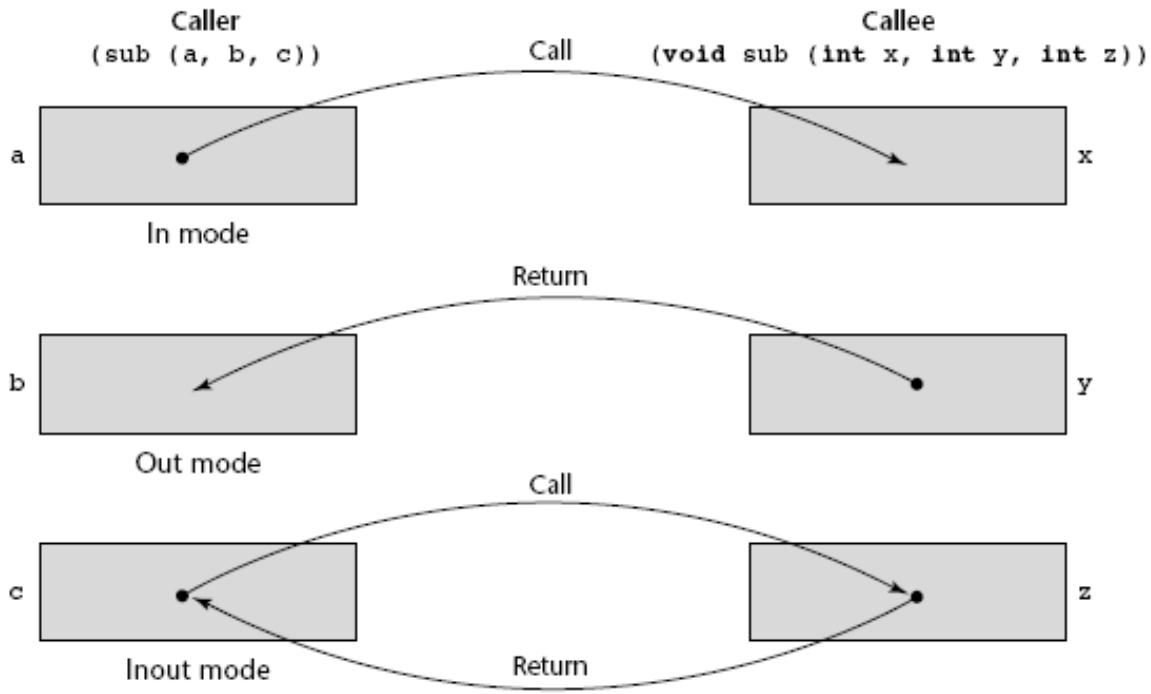
- Existem dois modelos conceituais sobre como os dados são transferidos na passagem de parâmetros
  - O valor real é copiado
  - Um caminho de acesso é transmitido

Métodos de passagens de parâmetros

- Existem dois modelos conceituais sobre como os dados são transferidos na passagem de parâmetros
  - O valor real é copiado
  - Um caminho de acesso é transmitido

Figure 9.1

The three semantics models of parameter passing when physical moves are used



Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por valor
    - Quando um parâmetro ó passado por valor, o valor do parâmetro real é utilizado para inicializar o parâmetro formal correspondente (in mode)
    - A passagem por valor geralmente é implementada por cópia, mas pode ser implementada transmitindo-se o caminho de acesso

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por valor
    - Quando um parâmetro ó passado por valor, o valor do parâmetro real é utilizado para inicializar o parâmetro formal correspondente (in mode)
    - A passagem por valor geralmente é implementada por cópia, mas pode ser implementada transmitindo-se o caminho de acesso
    - Vantagem: rápido para valores escalares

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por valor
    - Quando um parâmetro ó passado por valor, o valor do parâmetro real é utilizado para inicializar o parâmetro formal correspondente (in mode)
    - A passagem por valor geralmente é implementada por cópia, mas pode ser implementada transmitindo-se o caminho de acesso
    - Vantagem: rápido para valores escalares
    - Desvantagem: memória extra e tempo de cópia (para parâmetros que ocupam bastante memória)

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por resultado
    - É uma implementação do modelo out mode
    - Quando um parâmetro é passado por resultado, nenhum valor é transmitido para o subprograma
    - O parâmetro formal funciona como uma variável local
    - Antes do retorno do subprograma, o valor é transmitido de volta para o parâmetro real

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por resultado
    - Mesmas vantagens e desvantagens da passagem por valor
    - Outras questões
    - Colisão de parâmetros reais
    - Momento da avaliação do endereço dos parâmetros reais

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por resultado
    - Mesmas vantagens e desvantagens da passagem por valor
    - Outras questões
    - Colisão de parâmetros reais
    - Momento da avaliação do endereço dos parâmetros reais
    - Exemplo em C#

```
void Fixer (out int x, out int y) {
    x = 17; y = 35;
}
...
Fixer(out a, out a);
...
void DoIt(out int x, out int index) {
   x = 17; index = 42;
}
...
sub = 21;
DoIt(out list[sub], out sub);
```

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por valor-resultado (por cópia)
    - É uma implementação do modelo inout mode
    - É uma combinação da passagem por valor e passagem por resultado
    - Compartilha os mesmos problemas da passagem por valor e passagem por resultado

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por referência
    - É uma implementação do modelo inout mode
    - Ao invés de copiar os dados, um caminho de acesso é transmitido (geralmente um endereço)

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por referência
    - É uma implementação do modelo inout mode
    - Ao invés de copiar os dados, um caminho de acesso é transmitido (geralmente um endereço)
    - Vantagens: eficiente em termos de espaço e tempo
    - Desvantagens: acesso mais lento devido a indireção, apelidos podem ser criados

Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por referência
    - É uma implementação do modelo inout mode
    - Ao invés de copiar os dados, um caminho de acesso é transmitido (geralmente um endereço)
    - Vantagens: eficiente em termos de espaço e tempo
    - Desvantagens: acesso mais lento devido a indireção, apelidos podem ser criados
    - Exemplo em C++

```
void fun (int &first, int &second){...}
...
fun(total, total);
fun(list[i], list[j]);
fun1(list[i], list);
int *global;
void main() {
   sub(global);
}
void sub(int *param) {...}
```

Métodos de passagens de parâmetros

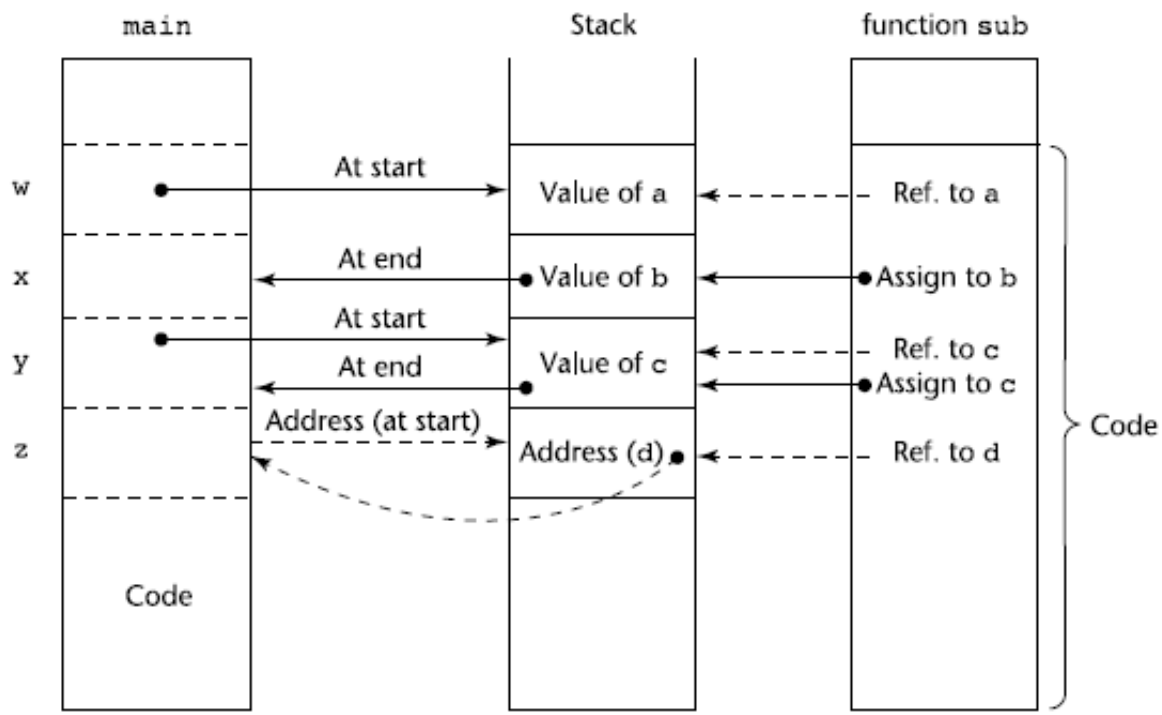
- Modelos de implementação
  - Passagem por nome
    - É um método de passagem de parâmetro inout mode
    - Não corresponde a um único modelo de implementação
    - O parâmetro real substitui textualmente o parâmetro formal em todas as ocorrências do subprograma
    - Usando em meta programação

Métodos de passagens de parâmetros

- Implementação
  - Na maioria das linguagens contemporâneas, a comunicação dos parâmetros acontece através da pilha
  - A pilha é inicializada e mantida pelo sistema
  - Exemplo

Figure 9.2

One possible stack implementation of the common parameter-passing methods



Métodos de passagens de parâmetros

- Exemplos de algumas linguagens
  - C usa passagem por valor
  - Passagem por referência pode ser obtida usando ponteiros
  - Em C e C++, os parâmetro formais podem ter o tipo ponteiro para constante
  - C++ incluí um tipo especial de ponteiro, chamado de tipo referência

    ```
    void fun(const int &p1, int p2, int &p3) { ... }
    ```

  - Todos os parâmetro em Java são passados por valor. Como os objetos são acessados por referência, os parâmetros dos tipos objetos são efetivamente passados por referência

Métodos de passagens de parâmetros

- Exemplos de algumas linguagens
  - Ada implementa os três modelos semânticos

```
procedure Adder(A : in out Integer;
                B : in Integer;
                C : out Float)
```

  - Parâmetros como out mode podem ser atribuídos mas não referenciados
  - Parâmetros como in mode podem ser referenciados mas não atribuídos
  - Parâmetros in out mode podem ser referenciados e atribuídos
  - Em Ada 95, todos os escalares são passados por cópia e todos os valores de tipos estruturados são passados por referência
  - Fortran 95 é similar ao Ada
  - E as linguagens de scripts?

Métodos de passagens de parâmetros

- Checagem de tipos dos parâmetros
  - As primeiras linguagens de programação (como Fortran 77 e C) não requeriam checagem dos tipos dos parâmetros
  - A maioria das linguagens atuais fazem esta checagem (e as linguagens de scripts?)

Métodos de passagens de parâmetros

- Checagem de tipos dos parâmetros
  - As primeiras linguagens de programação (como Fortran 77 e C) não requeriam checagem dos tipos dos parâmetros
  - A maioria das linguagens atuais fazem esta checagem (e as linguagens de scripts?)
  - Em C89, o programador pode escolher

```
double sin(x) // sem checagem
  double c;
{...}
double value;
int count;
value = sin(count);

double sin(double x) // com checagem
{ ... }
```

Métodos de passagens de parâmetros

- Arranjos multimensionais como parâmetros
  - O compilador precisa saber o tamanho do arranjo multidimensional para criar a função de mapeamento
  - Em C/C++ o programador tem que declarar todos os tamanhos (menos do primeiro subscrito)

```
void fun(int matrix[][10]) {...}
void main() {
    int mat[5][10];
    fun(mat);
}
```

  - Ada, Java e C# não tem este problema, o tamanho do arranjo a faz parte do objeto

Métodos de passagens de parâmetros

- Considerações de projeto
  - Considerações importantes
    - Eficiência
    - Transferência de dados em uma ou duas direções
  - Estas considerações estão em conflito
    - As boas práticas de programação sugerem limitar o acesso as variáveis, o que implica em usar transferência em uma direção quando possível
    - Mas passagem por referência é mais eficiente para estruturas com tamanho significativo

Parâmetros que são subprogramas

Parâmetros que são subprogramas

- Existem muitas situações que é conveniente passar um nome de subprograma como parâmetro para outros subprogramas
  - A ação que deve ser realiza quando um evento ocorre (ex: clique de botão)
  - A função de comparação utilizada por um subprograma de ordenação
  - Etc

Parâmetros que são subprogramas

- Existem muitas situações que é conveniente passar um nome de subprograma como parâmetro para outros subprogramas
  - A ação que deve ser realiza quando um evento ocorre (ex: clique de botão)
  - A função de comparação utilizada por um subprograma de ordenação
  - Etc
- Simples se apenas o endereço da função fosse necessário, mas existem duas questões que devem ser consideradas
  - Os parâmetros do subprograma passado como parâmetro são checados?
  - Qual é o ambiente de referenciamento usado na execução do subprograma passado como parâmetro?

Parâmetros que são subprogramas

- Os parâmetros do subprograma passado como parâmetro são checados?
  - Versão original do Pascal permitia a passagem de subprogramas como parâmetro sem incluir informações dos tipos dos parâmetros
  - Fortran, C/C++ incluem informações dos tipos
  - Ada não permite parâmetros que são subprogramas (uma forma alternativa é fornecida através de construções genéricas)
  - Java não permite parâmetros que são métodos

Parâmetros que são subprogramas

- Qual é o ambiente de referenciamento usado na execução do subprograma passado como parâmetro?
  - **Vinculação rasa**: o ambiente da instrução de chamada que ativa o subprograma passado - natural para linguagens com escopo dinâmico
  - **Vinculação profunda**: o ambiente da definição do subprograma passado
    - natural para linguagens com escopo estático
  - **vinculação ad hoc**: o ambiente da instrução de chamada que passou o subprograma como parâmetro real

Parâmetros que são subprogramas

- Exemplo usando a sintaxe de Javascript

```
function sub1() {
  var x;
  function sub2() {
    print(x);
  }
  function sub3() {
    var x = 3;
    sub4(sub2);
  }
  function sub4(subx) {
    var x = 4;
    subx();
  }
  x = 1;
  sub3();
}
```

Qual será o valor impresso na função `sub2` quando ela for chamada em `sub4`?

- Vinculação rasa:

Qual será o valor impresso na função `sub2` quando ela for chamada em `sub4`?

- Vinculação rasa: 4 (`x` de `sub4`)

Qual será o valor impresso na função `sub2` quando ela for chamada em `sub4`?

- Vinculação rasa: 4 (`x` de `sub4`)
- Vinculação profunda:

Qual será o valor impresso na função `sub2` quando ela for chamada em `sub4`?

- Vinculação rasa: 4 (`x` de `sub4`)
- Vinculação profunda: 1 (`x` de `sub1`)

Qual será o valor impresso na função `sub2` quando ela for chamada em `sub4`?

- Vinculação rasa: 4 (`x` de `sub4`)
- Vinculação profunda: 1 (`x` de `sub1`)
- Vinculação ad hoc:

Qual será o valor impresso na função `sub2` quando ela for chamada em `sub4`?

- Vinculação rasa: 4 (`x` de `sub4`)
- Vinculação profunda: 1 (`x` de `sub1`)
- Vinculação ad hoc: 3 (`x` de `sub3`)

Subprogramas sobrecarregados

Subprogramas sobrecarregados

- Um **subprograma sobrecarregado** é um subprograma que tem o mesmo nome de outro subprograma no mesmo ambiente de referenciamento
  - Cada versão precisa ter um único protocolo
  - O significado de uma chamada é determinado pela lista de parâmetros reais (ou/e pelo tipo de retorno, no caso de funções)

Subprogramas sobrecarregados

- Um **subprograma sobrecarregado** é um subprograma que tem o mesmo nome de outro subprograma no mesmo ambiente de referenciamento
  - Cada versão precisa ter um único protocolo
  - O significado de uma chamada é determinado pela lista de parâmetros reais (ou/e pelo tipo de retorno, no caso de funções)
  - Quando coercão de parâmetros são permitidas, o processo de distinção fica complicado. Exemplo e C++

```
int f(float x) { ... }
int f(double x) { ... }
int a = f(2); // erro de compilação
```

Subprogramas sobrecarregados

- Um **subprograma sobrecarregado** é um subprograma que tem o mesmo nome de outro subprograma no mesmo ambiente de referenciamento
  - Cada versão precisa ter um único protocolo
  - O significado de uma chamada é determinado pela lista de parâmetros reais (ou/e pelo tipo de retorno, no caso de funções)
  - Quando coercão de parâmetros são permitidas, o processo de distinção fica complicado. Exemplo e C++

```
int f(float x) { ... }
int f(double x) { ... }
int a = f(2); // erro de compilação
```

  - Subprogramas sobrecarregados com parâmetros padrões podem levar a uma chamada ambígua. Exemplo em C++

```
int f(double x = 1.0) { ... }
int f() { ... }
int a = f(); // erro de compilação
```

Subprogramas sobrecarregados

- Exemplos
  - C não permite subprograma sobrecarregados
  - Python, Lua e outras linguagens de scripts também não permitem
  - C++, Java, Ada e C# permitem (e incluem) subprogramas sobrecarregados
  - Ada pode usar o tipo de retorno da função para fazer distinção entre funções sobrecarregadas

Subprogramas sobrecarregados

- Exemplos
  - C não permite subprograma sobrecarregados
  - Python, Lua e outras linguagens de scripts também não permitem
  - C++, Java, Ada e C# permitem (e incluem) subprogramas sobrecarregados
  - Ada pode usar o tipo de retorno da função para fazer distinção entre funções sobrecarregadas
- Vantagem
  - Aumenta a legibilidade

Subprogramas sobrecarregados

- Exemplos
  - C não permite subprograma sobrecarregados
  - Python, Lua e outras linguagens de scripts também não permitem
  - C++, Java, Ada e C# permitem (e incluem) subprogramas sobrecarregados
  - Ada pode usar o tipo de retorno da função para fazer distinção entre funções sobrecarregadas
- Vantagem
  - Aumenta a legibilidade
- Desvantagem
  - Dificulta a utilização de reflexão

Subprogramas genéricos

Subprogramas genéricos

- Um **subprograma polimórfico** recebe diferente tipos de parâmetros em diferentes ativações
- Subprogramas sobrecarregados fornecem o chamado **polimorfismo ad hoc**
- Python e Ruby fornecem um tipo mais geral de polimorfismo (em tempo de execução)
- **Polimorfismo paramétrico** é fornecido por um subprograma que recebe parâmetros genéricos que são usados em expressões de tipos que descrevem os tipos dos parâmetros do subprograma
- Os subprogramas com polimorfismo paramétricos são chamados de **subprogramas genéricos**

Subprogramas genéricos

- Ada
  - Um versão do subprograma genérico é criado pelo compilador quando instanciado explicitamente em uma instrução de declaração
  - Precedido pela cláusula `generic` que lista as variáveis genérias, que podem ser tipos ou outros subprogramas

Subprogramas genéricos - Exemplo em Ada

```
generic
  type Index_Type is (<>);
  type Element_Type is private;
  type Vector is array (Index_Type range <>) of Element_Type;
  with function ">"(left, right : Element_Type) return Boolean is <>;
  procedure Generic_Sort(List : in out Vector);
  procedure Generic_Sort(List : in out Vector) is
    Temp : Element_Type;
    begin
      for Top in List'First .. Index_Type'Pred(List'Last) loop
        for Bottom in Index_type'Succ(Top) .. List'Last loop
          if List(Top) > List(Bottom) then
            Temp := List(Top);
            List(Top) := List(Bottom);
            List(Bottom) := Temp;
          end if;
        end loop;
      end loop;
    end Generic_Sort;
```

Subprogramas genéricos - Exemplo em Ada

```
type Int_Array is array(Integer range <>) of Integer;
procedure Integer_Sort is new Generic_Sort(
Index_Type => Integer, Element_Type => Integer, Vector => Int_Array);
```

Subprogramas genéricos

- C++
  - Versões do subprograma genérico são criados implicitamente quando o subprograma é chamado ou utilizado com o operador &
  - Precedido pela cláusula `template` que lista as variáveis genéricas, que podem ser nomes de tipos, inteiros, etc

Subprogramas genéricos - Exemplo em C++

```
template <class Type>
Type max(Type first, Type second) {
  return first > second ? first : second;
}
struct P {};
void test_max() {
  int a, b, c;
  char d, e, f;
  P g, h, i;
  // instanciação implícita
  c = max(a, b);
  f = max(d, e);
  // instanciação explícita
  float x = max<float>(1.2, 3);
  // erro de compilação
  float y = max(1.2, 3);
  // erro de compilação, o operador > não foi definido
  // para o tipo P
  i = max(g, h);
}
```

Subprogramas genéricos - Exemplo em C++

```
template <typename T>
void generic_sort(T list[], int n) {
  for (int top = 0; top < n - 2; top++) {
    for (int bottom = top + 1; bottom < n - 1; bottom++) {
      if (list[top] > list[bottom]) {
        T temp = list[top];
        list[top] = list[bottom];
        list[bottom] = temp;
      }
    }
  }
}
struct P {};
void test_generic_sort() {
  int is[] = {10, 5, 6, 3};
  generic_sort(is, 4);
  // erro de compilação, o operador > não foi definido
  // para o tipo P
  P ps[] = {P(), P(), P()};
  generic_sort(ps, 3);
}
```

Subprogramas genéricos

- Java
  - Adicionado ao Java 5.0
  - As variáveis genéricas são especificadas entre $< >$ antes do tipo de retorno do método
  - Diferenças entre C++/Ada e Java
    - Os parâmetros genéricos precisam ser classes
    - Apenas uma cópia do método genérico para todas as instaciações
    - É possível especificar restrições sobre as classes que podem ser utilizadas como parâmetros genéricos
    - Parâmetro genéricos do tipo wildcard (curinga)

Subprogramas genéricos

- Java
  - Adicionado ao Java 5.0
  - As variáveis genéricas são especificadas entre $< >$ antes do tipo de retorno do método
  - Diferenças entre C++/Ada e Java
    - Os parâmetros genéricos precisam ser classes
    - Apenas uma cópia do método genérico para todas as instaciações
    - É possível especificar restrições sobre as classes que podem ser utilizadas como parâmetros genéricos
    - Parâmetro genéricos do tipo wildcard (curinga)
- C#
  - Adicionado ao C# 2005
  - Semelhante ao Java
  - Não tem suporte a tipo wildcard
  - Uma versão para cada tipo primitivo

Questões de projeto referentes a funções

Questões de projeto referentes a funções

- Efeitos colaterais são permitidos?

Questões de projeto referentes a funções

- Efeitos colaterais são permitidos?
  - Funções em Ada podem ter apenas parâmetros in mode, o que diminui as formas de efeitos colaterais

Questões de projeto referentes a funções

- Efeitos colaterais são permitidos?
  - Funções em Ada podem ter apenas parâmetros in mode, o que diminui as formas de efeitos colaterais
- Qual tipo de valores podem ser retornados?

Questões de projeto referentes a funções

- Efeitos colaterais são permitidos?
  - Funções em Ada podem ter apenas parâmetros in mode, o que diminui as formas de efeitos colaterais
- Qual tipo de valores podem ser retornados?
  - C/C++ não permite o retorno de arranjos e funções (ponteiros para arranjos e função são permitidos)
  - Java, C#, Ada, Python, Ruby, Lua permitem o retorno de qualquer tipo
  - Ada não permite o retorno de funções, por que função não tem tipo. Ponteiros para funções tem tipo e podem ser retornados
  - Java e C#: métodos não tem tipos

Questões de projeto referentes a funções

- Efeitos colaterais são permitidos?
  - Funções em Ada podem ter apenas parâmetros in mode, o que diminui as formas de efeitos colaterais
- Qual tipo de valores podem ser retornados?
  - C/C++ não permite o retorno de arranjos e funções (ponteiros para arranjos e função são permitidos)
  - Java, C#, Ada, Python, Ruby, Lua permitem o retorno de qualquer tipo
  - Ada não permite o retorno de funções, por que função não tem tipo. Ponteiros para funções tem tipo e podem ser retornados
  - Java e C#: métodos não tem tipos
- Quantos valores podem ser retornados?

Questões de projeto referentes a funções

- Efeitos colaterais são permitidos?
  - Funções em Ada podem ter apenas parâmetros in mode, o que diminui as formas de efeitos colaterais
- Qual tipo de valores podem ser retornados?
  - C/C++ não permite o retorno de arranjos e funções (ponteiros para arranjos e função são permitidos)
  - Java, C#, Ada, Python, Ruby, Lua permitem o retorno de qualquer tipo
  - Ada não permite o retorno de funções, por que função não tem tipo. Ponteiros para funções tem tipo e podem ser retornados
  - Java e C#: métodos não tem tipos
- Quantos valores podem ser retornados?
  - A maioria das linguagens permitem apenas um valor de retorno
  - Python, Ruby e Lua permitem o retorno de mais de um valor

Operadores sobrecarregados definidos pelo usuário

Operadores sobrecarregados definidos pelo usuário

- Operadores podem ser sobrecarregados pelo usuário em Ada, C++, Python e Ruby

Operadores sobrecarregados definidos pelo usuário

- Operadores podem ser sobrecarregados pelo usuário em Ada, C++, Python e Ruby
- Exemplos (produto escalar de dois arranjos)
  - Ada

```
function "*"(A, B : in Vector_Type)
             return Integer is
  Sum : Integer := 0;
  begin for Index in A'range loop
    Sum := Sum + A(Index) * B(Index);
  end loop;
  return Sum;
end "*";
```

Operadores sobrecarregados definidos pelo usuário

- Exemplos (produto escalar de dois arranjos)
  - C++

```
int operator *(vector<int> &A, vector<int> &B) {
  int sum = 0;
  for (int i = 0; i < A.size(); i++) {
    sum += A[i] * B[i];
  }
  return sum;
}
```

Fechamentos

Fechamentos

- Um **fechamento** (*closure* em inglês) é um subprograma e o ambiente de referenciamento onde ele foi definido
- O ambiente de referenciamento é necessário pois o subprograma pode ser chamado em qualquer local

Fechamentos

- Exemplo python

```
def somador(x):
    def soma(n):
      return x + n
    return soma

>>> soma1 = somador(1)
>>> soma1(5)
6
>>> soma5 = somador(5)
>>> soma5(3)
8
>>> soma1.func_closure[0].cell_contents
1
>>> soma5.func_closure[0].cell_contents
5
```

Referências

Referências

- Robert Sebesta, Concepts of programming languages, 9ª edição. Capítulo 9.