Abstração

Prof. Hugo de Paula

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Departamento de Ciência da Computação

Sumário

Abstrações

Abstração

Entidade que incorpora uma computação. Remove qualquer dependência de aspectos específicos de implementação.

- Ideia: permitir expressar “o que” uma parte de um programa faz, sem que seja preciso dizer “como”
- Uma abstração produz encapsulamento
  - *visão interna*: visão do programador da abstração
  - *visão externa*: visão do usuário da abstração
- Dois modelos de abstração básicos:
  - Abstração de processo.
  - Abstração de dados.

Exemplo de abstrações de processo

- **Funções**: incorpora uma expressão que deve ser avaliada
  - Visão do usuário: mapeia argumentos em um resultado
  - Visão do programador: avalia corpo da função com parâmetros formais associados a seus respectivos argumentos
- **Procedimentos**: incorpora uma sequência de comandos
  - Visão do usuário: atualiza ambiente de execução
  - Visão do programador: executa corpo do procedimento com parâmetros formais associados a seus respectivos argumentos

Abstrações de subprogramas

Exemplo de abstração de função em Pascal

```
function potencia (x: real; n: integer): real
begin
    if n = 1
      then potencia:= x
      else potencia:= x * potencia (x, n-1)
end;
```

Crítica: identificador função denota duas entidades diferentes

Princípio da Abstração

Princípio da Abstração

Para cada construção sintática, semanticamente relevante na linguagem, deve existir um mecanismo de abstração.[a]

[a] Alfred John Cole, Ronald Morrison (1982)

Assim:

- para comandos existem procedimentos
- para expressões existem funções
- para tipos abstratos de dados existem classes e objetos

Extreme Programming (XP): *automatize cada processo que deve ser automatizado. Se uma tarefa é repetitiva, torne-a um script (abstração)*.

Subprogramas

- Cada subprograma deve ter um único ponto de entrada.
- A execução do programa que chama o subprograma é suspensa até que a execução do subprograma seja terminada (modelo síncrono).
- No modelo assíncrono, a execução do programa continua enquanto o subprograma é executado e um valor futuro é atribuído. Quando o valor futuro é necessário, o programa precisa aguardar explicitamente o término do subprograma.
- Controle sempre retorna para o ponto de chamada ao final da execução do subprograma.
- Uma chamada a um subprograma é uma requisição explícita de que o subprograma deve ser executado.

Definições básicas de subprogramas

- Uma **definição** de um subprograma descreve a interface e as ações da abstração de subprograma.
- O **cabeçalho** é a primeira parte da definição, e inclui: nome, tipo do subprograma e os parâmetros formais.
- A **assinatura** (ou perfil dos parâmetros) inclui: quantidade, ordem e tipos dos parâmetros.
- Um **protocolo** corresponde à assinatura do subprograma e seu valor de retorno, no caso de funções.
- **Declarações** de subprogramas provêm o protocolo, mas não o corpo do subprograma.

Registro de ativação

- Bloco de informação associado a cada chamada de subprograma.
- Elementos:
  - parâmetros e variáveis locais.
  - endereço de retorno.
  - estado dos registradores.
  - variáveis temporárias.
  - valor de retorno.
  - link estático ao “pai” (*parent*) do subprograma (ponto de chamada).
  - link dinâmico para o registro de ativação do escopo de chamada.

Parâmetros

Parâmetros em C

```
float area (float raio) {
        return pi * raio * raio;
}
area (5); ...        // chamada da função
area (a + b);
```

- **Argumento**: valor que é passado para a função. Exemplo: *5* e resultado da avaliação de *a+b*
- **Parâmetro de chamada ou parâmetro real**: expressão que produz o argumento. Exemplo: *5* e *a+b*
- **Parâmetro formal**: identificador que denota o argumento no interior da função. Exemplo: *raio*

Correspondência entre parâmetro real e formal

- **Posicional**:
  - A associação entre o parâmetro real e o formal é dada pela sua posição.
  - Seguro e eficiente.
- **Por palavra chave**:
  - O nome do parâmetro formal é especificado na chamada juntamente com o parâmetro real.
  - Vantagem: Parâmetros podem aparecer em qualquer ordem.
  - Desvantagem: deve-se conhecer o nome do parâmetro formal.

Valores *default*

- Algumas linguagens permitem que parâmetros formais tenham valores *default*. P. Ex.: C++, Python, Ruby, PHP.
- Se não for passado o parâmetro real, parâmetro formal assume valor *default*.
  - Em C++, parâmetros *default* devem aparecer no final, pois utiliza notação posicional.

Exemplo em C

```
void shulambs(int a, int b = 10) {
    // faz alguma coisa
}
int main() {
    shulambs(10);
    shulambs(10, 20); // sobrepoe o valor default
}
```

Número variável de parâmetros

- C# aceita número variável de parâmetros, desde que eles sejam do mesmo tipo. Parâmetro formal é um arranjo precedido pela palavra **params**.

Exemplo em C#

```
public static void ParamVariavel(params int[] lista)
{
    for (int i = 0; i < lista.Length; i++)
    {
        Console.Write(lista[i] + " ");
    }
    Console.WriteLine();
}
```

Número variável de parâmetros

Exemplo em Python

```
    def multiplica(x, y):
        print (x * y)
    multiplica(1, 2, 3)

# TypeError: multiplica() takes 2 positional arguments
# but 3 were given

    def multiplicaVar(*args):
        res = 1
        for num in args:
            res *= num
        print(res)
    multiplicaVar(2, 3, 4)
    multiplicaVar(2, 3, 4, 5)
```

Mecanismos de passagem de parâmetros

- **Mecanismos de cópia**: valores são copiados na entrada e/ou saída da abstração
- **Mecanismos de definição**: permite que um parâmetro formal *x* seja associado diretamente ao argumento

Mecanismos de cópia

- Valores são copiados na entrada e/ou saída da abstração
  - Passagem por valor
  - Passagem por resultado
  - Passagem por valor-resultado

Exemplo em ADA

```
procedure p (in x: integer, out y: real, in out z: real)
Chamada: p (10, a, b)
```

Modo de Passagem	Argumento	Efeito na Entrada	Efeito na Saída
Valor (in)	valor	x := 10	-
Resultado (out)	variável	-	a := y
Valor-Resultado (in out)	variável	z := b	b := z

Mecanismos de definição

- **Passagem por referência**: permite que um parâmetro formal *x* seja associado diretamente ao argumento
  - Parâmetro constante: **void** f (**const int** &x)
  - Parâmetro variável: **void** f (**int** &x)
- Qualquer utilização de *x* é na verdade uma utilização indireta do argumento (*x* é um *aliasing* para o argumento)
- Mecanismos disponíveis em diversas linguagens:
  - C: valor (passagem por referência simulada via ponteiros)
  - Pascal: valor e referência (palavra reservada var)
  - C++: valor e referência
  - Java e C#: valor para tipos primitivos e referência para objetos de classes
  - ADA: valor (in), resultado (out) e valor-resultado (in out)

Passagem de Subprogramas como Parâmetro

- Parametrizar um subprograma com um parâmetro que representa um outro subprograma (procedimento ou função).
  - Em C e C++: por meio de ponteiros

```
double integral(double (*fun)(double), double inf, double sup) {
   double soma = (*fun)(inf), dx = 0.00001;
   for (float x = inf + dx; x <= sup; x += dx)
      soma += (*fun)(x);
   return soma * dx;
}
double f(float x) {
   return x + 1;
}
void main() {
   cout << integral (f, 0, 3.1415926);
}
```

Exemplo

- Que valor será impresso, supondo passagem por:
- a) valor-resultado b) referência?

Exemplo em Pascal

```
1   program Shulambs(input, output);
2   var A: integer;
3   procedure qquer(var x : integer);
4   begin
5       x := 2;
6       A := 0;
7   end;
8   begin
9       A := 1;
10      qquer(A);
11      writeln(A)
12  end.
```

Mecanismos de definição

- **Passagem por nome**: o parâmetro formal recebe como argumento o próprio nome do parâmetro de chamada. Em cada instante da execução, o parâmetro é interpretado baseado no seu nome por ocasião da chamada.
  - parâmetro reflete as alterações de valor do argumento.
  - argumento é avaliado e pode se modificar durante a execução do procedimento.
- utilizado para implementar avaliação tardia (*lazy evaluation*)
- Exemplos de linguagens:
  - Algol-60
  - LISP: através de *quotation*

Ordem de Avaliação

- Quando uma função é chamada, em que momento o parâmetro real (ou de chamada) é avaliado ?
- Duas alternativas:
  - Avaliação de Ordem Aplicativa (*Eager Evaluation*)
  - Avaliação de Ordem Normal (*Lazy Evaluation*)
- Avaliação de Ordem Aplicativa:
  - Parâmetro real avaliado uma única vez, no momento da chamada
  - Exemplos: Algol-68, Pascal, C, C++, Java, C#, LISP, etc
- Avaliação de Ordem Normal:
  - Parâmetro real avaliado toda vez que argumento for utilizado no interior da função.
  - Exemplo: Algol-60 (passagem de parâmetros por nome), LISP (quote/eval), Haskell, Scheme, JavaScript, Ruby, etc

Ordem de Avaliação

- Exemplo: Para n = 2 e t = 2, qual será o resultado da chamada cand (n > 0 , t/n > 0.5) ? E se n = 0?

Exemplo em ML

```
fun cand (b1: bool, b2: bool) =
    if b1 then b2 else false
```

- Função *strict* (rigorosa) = uma chamada à função só pode ser avaliada se todos os seus argumentos puderem ser avaliados.
- Função *nonstrict* = uma função é dita ser *nonstrict* com relação a um argumento *n*, se uma chamada à função puder ser avaliada, mesmo que o *n*-ésimo argumento não possa ser avaliado

Exemplo de Avaliação Tardia

Exemplo em JavaScript

```
function uns() {
    return new Stream( 1, uns );
}
function numerosNaturais() {
    return new Stream(
        1,
        function () {
            return uns().add( numerosNaturais() );
        }
    );
}
numerosNaturais().take( 5 ).print(); // imprime 1, 2, 3, 4, 5
```

Exemplo de Avaliação Tardia

Exemplo em JavaScript

```
uns
={ 1, uns }
={ 1, { 1, uns } }
= ...
={ 1, { 1, { 1, ... // infinitamente!

nat
={ 1, uns+nat }
={ 1, { 1, uns } + { 1, uns+nat } } = { 1, { 1+1, uns+uns+nat } }
={ 1, { 2, { 1, uns } + { 1, uns } + { 1, nat } } }
= ...
={ 1, { 2, { 3, ... // e assim por diante.
```

Exemplo de Avaliação Tardia

Exemplo em C#

```
public int Soma()
{
    int x = 0;
    int y = 0;
    Lazy<int> s = new Lazy<int>(() => x + y);
    x = 10;
    y = 20;
    return s.Value; // resposta 30
}
```

Exemplo de Avaliação Tardia

Exemplo em C#: Fibonacci infinito

```
public static IEnumerable<long> Serie() {
  long n1 = 1;
  long n2 = 1;

  yield return n1;
  yield return n2;

  while (true) {
     long temp = n1 + n2;
     n1 = n2;
     n2 = temp;
     yield return temp;
  }
}
var primeiros10elementos = Fibonnaci.Serie().Take(10)
```

Exemplo de Avaliação Tardia

Exemplo em Haskell: Quicksort lazy

```
qSort [] = []
qSort (x:xs) =
   qSort (filter (< x) xs) ++ [x] ++ qSort (filter (>= x) xs)
```

Se quisermos encontrar o menor elemento de uma lista, não precisamos ordenar a lista inteira:

```
menor lista = head (qSort lista)
```

Closure

- Closure é um subprograma e seu ambiente de referenciação em que foi definido.
- O ambiente de referenciação é necessário se o subprograma for chamado de qualquer lugar do programa.
- Uma linguagem com escopo estático que não permite subprogramas aninhados não precisa de Closures.
- Closures só são necessários se um subprograma pode acessar variáveis em escopos aninhados e pode ser chamado de qualquer lugar.
- Para suportar closures, uma LP pode precisar prover extensão de tempo de vida de uma variável, pois um subprograma pode acessar uma variável não local que normalmente já não estaria alocada.

Exemplo de Closure

1. O Closure no JavaScript é a função anônima retornada pelo somador.

Exemplo em JavaScript

```
 function somador(x) {
  return function(y) {return x + y;}
}
...
var soma10 = somador(10);
var soma5 = somador(5);
document.write("Soma 10 a 20: " + soma10(20) + "<br />");
document.write("Soma 5 a 20: " + soma5(20) + "<br />");
```

Exemplo de Closure

- Podemos escrever um Closure em C# usando um *delegate* anônimo.
- Func<**int**, **int**> (tipo de retorno) especifica um *delegate* que recebe um inteiro como parâmetro e retorna outro inteiro.

Exemplo em C#

```
 static Func<int, int> somador(int x) {
   return delegate(int y) {return x + y;};
}
...
Func<int, int> soma10 = somador(10);
Func<int, int> soma5 = somador(5);
Console.WriteLine("Soma 10 a 20: {0}", soma10(20));
Console.WriteLine("Soma 5 a 20: {0}", soma5(20));
```

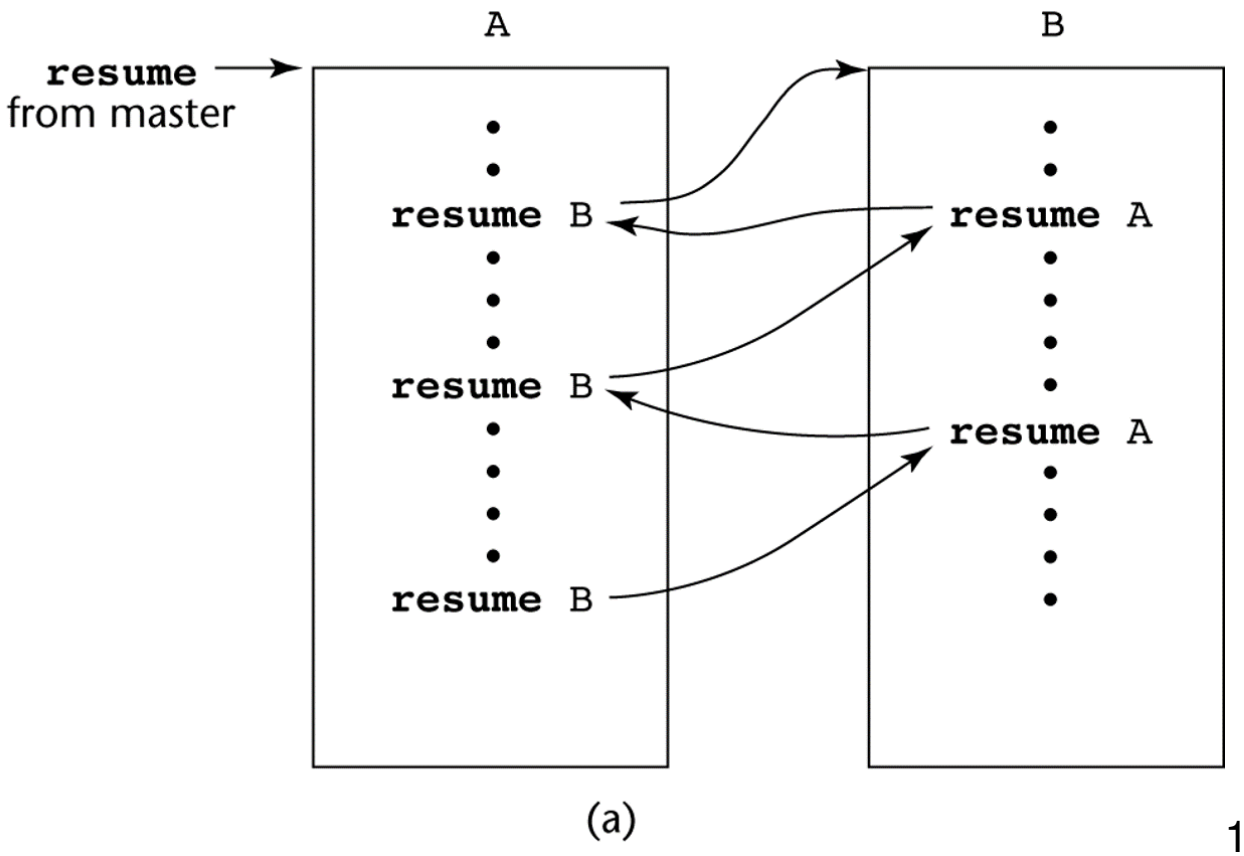
Co-rotinas

Corotinas

São subprogramas que possuem múltiplas entradas e se auto controlam.

- Uma *chamada* a uma co-rotina é denominada *resume*.
- O primeiro *resume* inicia a co-rotina, mas as *chamadas* subsequentes entram nos pontos logo após a última instrução executada.
- Co-rotinas podem se *resumir* eternamente.
- Co-rotinas provêm execução "*quasi-concorrente*" de blocos de programa: execuções são intercaladas, mas não sobrepostas.

Exemplo de execução de co-rotinas



[1]

[1] SEBESTA (2019). *Conceitos de Linguagens de Programação. 11ed*, Fig. 9.3

Exemplo de execução de co-rotinas

- Principal versão de co-rotina é o modelo produtor/consumidor.
- Consumidor transfere controle ao produtor quanto precisa do próximo valor gerado pelo produtor.
- Produtor retorna o controle ao consumidor com o valor produzido.
- Exemplo:
  - Parser do compilador solicita o próximo *token* ao analisador léxico.
  - Analisador léxico produz *tokens* sob demanda.

Exemplo de co-rotinas

Exemplo em Python

```
def rangeOnDemand(st, to):
  i = st
  while (i < to):
    yield i
    i += 1 # (resume) Próxima execução continua deste ponto

for i in rangeOnDemand(1, 5):
  print("{0}a exec de randOnDemand".format(i))
```

Saída

```
1a exec de randOnDemand
2a exec de randOnDemand
3a exec de randOnDemand
4a exec de randOnDemand
```