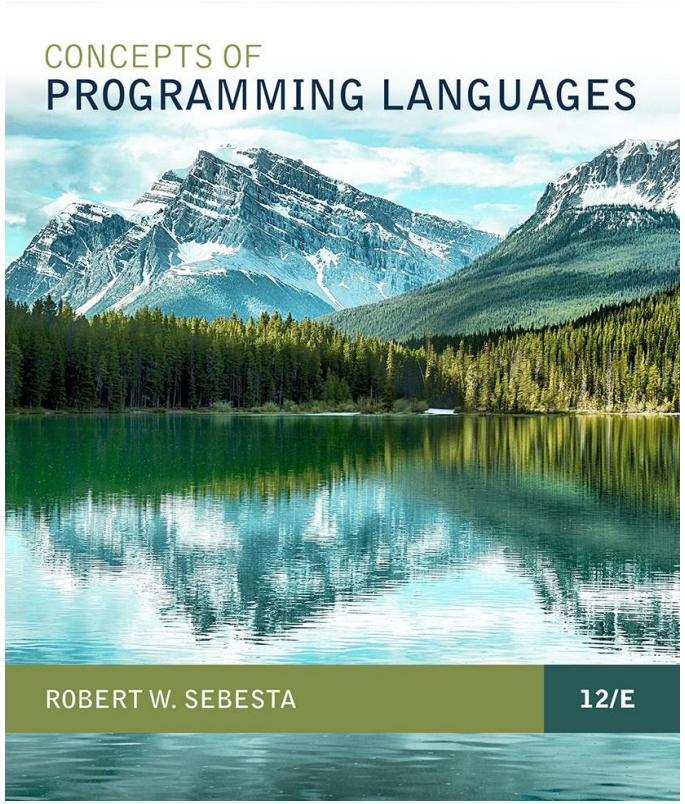
CAPÍTULO 9: SUBPROGRAMAS

Estructura de los lenguajes

Dr. Christian von Lücken

Tópicos

- Introducción
- Fundamentos de Subprogramas
- Cuestiones de diseño de Subprogramas
- Entornos Locales de Referencia
- Métodos de Paso de Parámetros
- Parámetros que son Subprogramas
- Llamando subprogramas indirectamente
- Cuestiones de diseño de funciones
- Subprogramas Sobrecargados
- Subprogramas Genéricos
- Sobrecarga de operadores definidos por el usuario
- Closures
- Corutinas



AJUSTADO A LAS PRESENTACIONES DEL LIBRO "CONCEPTS OF PROGRAMMING LANGUAGES", ROBERT SEBESTA, 12/E. PEARSON. 2018. ISBN 0-321-49362-1

Introducción

Dos facilidades de abstracción fundamentales

- Abstracción de procesos
  - Separación del código en funciones
  - Agrupación de funciones en librerías
  - Enfatizado desde el primer día
- Abstracción de datos
  - Tipos Abstractos de Datos
  - Estructuras de datos: registros, pilas, colas, árboles, grafos
  - Enfatizados en los 80s

Fundamentos de Subprogramas

- Cada subprograma tiene un único punto de entrada
- El programa que llama se suspende dirante la ejecución del subprograma llamado
- El control siempre retorna al llamador cuando la ejecución del subprograma llamado termina

Definiciones Básicas

Subprogram definition: describe la interfaz y las acciones del subprograma.

- En Python, las definiciones de funciones son ejecutables; en todos los otros lenguajes, son no-ejecutables
- En Ruby, las definiciones de funciones pueden aparecer ya sea dentro o fuera de las definiciones de clase. Si afuera, son métodos de Object. Ellos pueden ser llamados sin un objeto, como una función
- En Lua, todas las funciones son anónimas

Subprogram call: Es un pedido explícito de ejecución del subprograma.

Subprogram header: Es la primera parte de la definición del subprograma, incluye el nombre, el tipo de subprograma y los parámetros formales.

Parameter profile (signature): Es el número, orden y tipos de los parámetros del subprograma.

Protocol: Es el profile del subprograma, y si es una función el tipo de retorno.

Definiciones básicas (cont.)

Las declaraciones de las funciones en C y C++ son denominadas usualmente ***prototipos***.

Una ***declaración de subprograma*** provee el protocolo, pero no el cuerpo del subprograma.

Un ***parámetro formal*** es una variable tonta listada en la cabecera del subprograma y usada en el subprograma.

Los ***parámetros actuales*** representan un valor o dirección de memoria utilizado en una sentencia de llamada al subprograma.

Correspondencia Actual/Formal de Parámetros

Posicional

- El enlace de los parámetros actuales a formales es por posición: el primer parámetro actual es enlazado con el primer parámetro formal.
- Seguro y efectivo.

Por claves

- El nombre del parámetro formal al cual el parámetro actual será enlazado es especificado con el parámetro actual.
- Ventaja: Los Parámetros pueden aparecer en cualquier orden.
- Desventaja: El usuario debe conocer el nombre de los parámetros formales

Parámetros Formales. Valores por defecto

Ciertos lenguajes admiten valores por defecto para parámetros formales, cuando no se especifica un parámetro actual (ej: C++, Ada, Python, Ruby, PHP)

- En C++, los parámetros por defecto deben aparecer al final ya que los parámetros son asociados posicionalmente (no se utilizan keywords).

Número variable de parámetros

- Los métodos en C# pueden aceptar un número variable de parámetros siempre que estos sean del mismo tipo—el parámetro formal correspondiente es un array precedido por **`params`**

- En Ruby, los parámetros actuales se envían como elementos de un literal hash y el parámetro formal correspondiente es precedido por un asterisco (https://www.tutorialspoint.com/ruby/ruby_methods.htm)

- En Python, el parámetro actual es una lista de valores y el parámetro formal correspondiente es un nombre con un asterisco (https://www.tutorialspoint.com/python/python_functions.htm)

Procedimientos y Funciones

Existen dos categorías de subprogramas

- ***Procedimientos:*** son una colección de sentencias que definen cómputos parametrizados.
  - No especifican un valor de retorno.
- ***Funciones:*** son estructuralmente equivalentes a procedimientos pero semánticamente son modelados de funciones matemáticas.
  - Especifican un valor de retorno.
  - Se espera que no produzcan efectos colaterales (en la práctica pueden tener).

Diseño de Subprogramas Consideraciones

- Las variables locales son estáticas o dinámicas?
- Pueden las definiciones de los subprogramas aparecer en otras definiciones de subprogramas?
- Qué métodos de paso de parámetros se proveen?
- Se verifica el tipo de los parámetros?
- Si los subprogramas pueden ser pasados como parámetros y los subprogramas pueden ser anidados, cual es el entorno de referencias de un subprograma pasado?
- Se permiten los efectos colaterales funcionales?
- Qué tipo de valores pueden ser retornados de las funciones?
- Cuántos valores pueden retornarse de las funciones?
- Pueden ser sobrecargados los subprogramas?
- Pueden los subprogramas ser genéricos?
- Si el lenguaje permite subprogramas anidados, se soportan las closures?

Ambientes de Referencias Locales

Las variables locales pueden ser dinámicas (ligado al almacenamiento)

- Ventajas
  - Soporte para recursión.
  - El almacenamiento para locales puede compartirse entre algunos subprogramas
- Desventajas
  - Allocation/de-allocation, Tiempo de inicialización
  - Direccionamiento indirecto
  - Subprogramas no pueden ser *history sensitive*

Variables Locales pueden ser estáticas

- Más eficientes (no hay indirección)
- Sobrecarga en tiempo de ejecución
- No soporta recursión

Ambientes de Referencias Locales: Ejemplos

En la mayoría de los lenguajes contemporáneos, las locales son stack dynamic

En los lenguajes basados en C, las variables locales son por defecto stack dynamic, pero pueden ser declaradas **`static`**

Los métodos de C++, Java, Python, y C# solo tienen locales stack dynamic

Métodos de Pasos de Parámetros

Formas de paso de parámetros:

- Paso por valor (Modo In)
- Paso por resultado (Modo Out)
- Paso por valor – resultado (Modo In-Out)
- Paso por referencia
- Paso por nombre

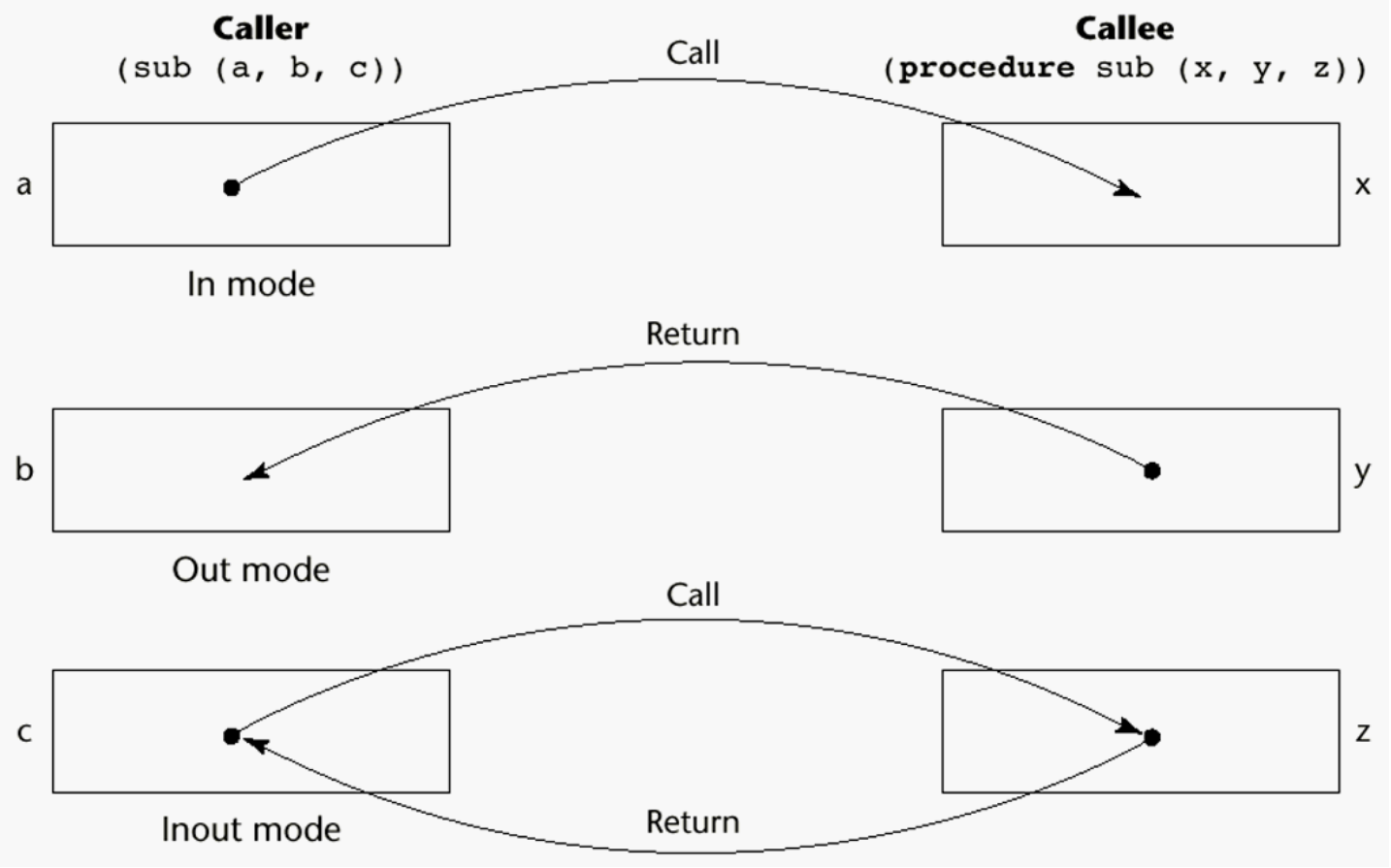
Modelos semánticos de paso de parámetros

In mode

Out mode

Inout mode

Modelos de paso de parámetros



Modelos conceptuales de transferencia

Mover físicamente un valor

Mover un camino de acceso al valor

Paso por Valor (Modo In)

El valor del parámetro actual es utilizado para inicializar el correspondiente parámetro formal

- Normalmente implementado por copia
- Puede ser implementado transmitiendo un camino de acceso pero no recomendado (forzar la protección contra escritura no es una tarea sencilla)
- Desventajas (si se mueve físicamente): almacenamiento adicional es requerido (se almacena dos veces) y mover puede ser costoso (para parámetros grandes).
- Desventajas (si se provee el camino): debe ser protegido de escritura en el programa llamado y el costo de acceso es mayor (acceso indirecto)

Paso por Resultado (Modo Out)

Cuando un parámetro es pasado por resultado, no se transmite ningún valor al subprograma

El correspondiente parámetro formal actúa como variable local; su valor es transmitido al parámetro actual del invocador cuando el control es retornado al invocador por medio de un movimiento físico: requiere almacenamiento extra y la operación de copia

Problema potencial:

`sub(p1, p1);` cualquiera de los parámetros formales representará el valor actual de p1

`sub(list[sub], sub);` Computar la dirección de list[sub] al comiendo del subprograma o al final?

Paso por Valor-Resultado (Modo InOut)

Una combinación de paso por valor y paso por resultado.

A veces denominado paso por copia.

Los parámetros formales tienen almacenamiento local.

Desventajas:

- Las de paso por resultado.
- Las de paso por valor.

Paso por Referencia (Modo InOut)

Se pasa el camino de acceso

Denominado paso compartido (pass-by-sharing)

Ventaja: El proceso de paso es eficiente (no hay copia ni almacenamiento duplicado)

Desventajas

- Acceso más lento (comparado con el paso por valor) a los parámetros formales.
- Propenso a efectos colaterales no deseados.
- Alias no deseados (Acceso múltiple)

```
fun(total, total);  fun(list[i], list[j];  fun(list[i], i);
```

Paso por Referencia (Modo InOut) - II

Otra cuestión:

Pueden cambiar las referencias pasadas en el subprograma llamado?

- En C, es posible

- En otros lenguajes, como Pascal y C++, los parámetros formales que son direcciones son dereferenciadas implícitamente, lo que evita tales cambios

Paso por Nombre (Inout Mode)

Por sustitución textual

Los parámetros formales son enlazados a un método de acceso al momento de la invocación, pero el enlace actual a un valor o referencia sucede en la sentencia de referencia o en la asignación.

Permite flexibilidad en el enlazado tardío (late binding)

La implementación requiere que el entorno de referencia dedl llamador sea pasado con el parámetro, de tal forma que la dirección del parámetro actual pueda ser calculado

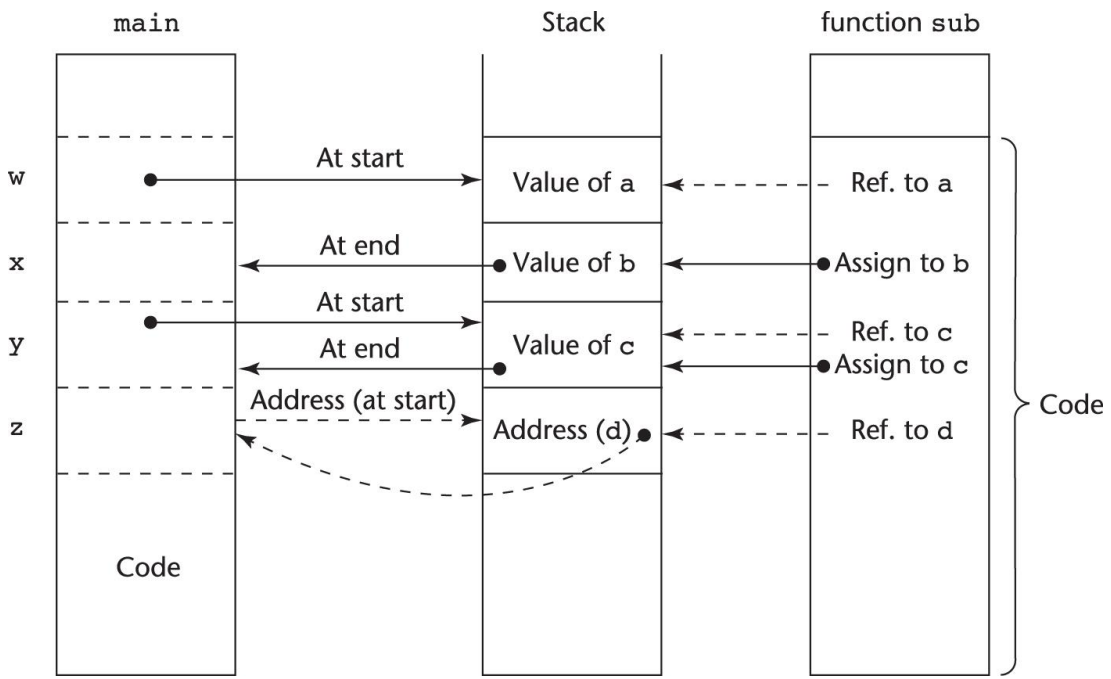
Implementación de los métodos de paso de parámetros

En la mayoría de los lenguajes el paso de parámetros ocurre a través de la pila de ejecución

El paso por referencia es el más simple, y solo se coloca la dirección de memoria en la pila

En los pasos por valor y por resultado se reserva almacenamiento tanto para los parámetros formales en el subprograma y los parámetros actuales en el invocador

Implementing Parameter-Passing Methods



Function header: `void sub(int a, int b, int c, int d)`

Function call in main: `sub(w, x, y, z)`

(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)

Paso de parámetros en algunos lenguajes

Fortran
- Modelo semántico inout
- Antes de Fortran 77: paso por referencia
- Fortran 77 y posteriores: variables escalares son pasadas por valor-resultado
- Fortran 95+
  - Los parámetros pueden ser declarados para que sean in, out, o inout

C
- Paso por valor
- Paso por referencia, a través de punteros como parámetros

C++
- Paso por referencia usando un puntero especial (para implementar pass-by-reference)

Java
- Todos los parámetros que no son objetos son pasados por valor (ningúnn método puede cambiarlos)
- Objetos se pasan por referencia

Paso de parámetros en algunos lenguajes (cont.)

Python y Ruby usa pass-by-assignment (todos los valores de datos son objetos); el actual se asigna al formal

Ada

- Admite tres modelos semánticos de transmisión de parámetros: `in`, `out`, `in out`; `in` es por defecto.
- Parámetros formales declarados como `out` pueden ser asignados pero no referenciados; los declarados como `in` pueden ser referenciados pero no asignados; los `in out` pueden ser referenciados y asignados.

C#

- Método por defecto: Paso por valor
- El paso por referencia se debe especificar en el parámetro formal y en el actual anteponiendo `ref`

PHP: muy similar a C#, excepto que ya sea el parámetro actual o formal puede ser especificado ref

Perl: todos los parámetros actuales son implícitamente colocados en un array predefinido denominado `@_`

Verificación de tipos de parámetros

- Importante para dar confiabilidad
- FORTRAN 77 y C original: no controlan
- Pascal, FORTRAN 90, Java, and Ada: siempre controlan
- ANSI C and C++: el usuario elige la alternativa: Prototipos
- Lenguajes relativamente nuevos Perl, JavaScript, y PHP no requieren verificación de tipos
- En Python y Ruby, las variables no tienen tipos (los objetos si), entonces el chequeo de tipos de los parámetros no es posible

Arrays multidimensionales como parámetros

Si se pasa un array multidimensional como parámetro y el subprograma es compilado por separado, el compilador necesita conocer el tamaño declarado del array para construir la función de mapeo para almacenamiento

Arrays multidimensionales como parámetros: C y C++

- El programador necesita incluir los tamaños declarados de cada dimensión del array, excepto la primera, en la definición del parámetro formal en el subprograma.
- No posibilita la escritura de subprogramas flexibles.
- Solución: Pasar un puntero al array y las dimensiones del mismo como otros parámetros, y el usuario define la función de mapeo para almacenamiento en términos de los parámetros que indican los tamaños.

Multidimensional Arrays as Parameters: Pascal and Ada

Pascal

- No es problema; el tamaño declarado es parte de la definición del array

Ada

- Arrays declarado con constantes - como Pascal
- Arrays no declarados con constantes – el tamaño declarado es parte de la declaración del objeto.

Arrays multidimensionales como parámetros: Java and C#

Similar a Ada

Arrays son objetos; son todos de 1 dimensión pero los elementos pueden ser arrays

Cada aray hereda una constante con nombre (`length` en Java, `Length` en C#) a la que se asigna el valor de la longitud del array cuando se crea el objeto

Consideraciones de diseño - Paso de parámetros

Dos consideraciones importantes

- Eficiencia
- Transferencia de datos de 1 vía o 2 vías

Consideraciones conflictivas

- La buena programación sugiere acceso limitado a las variables, lo cual implica 1 vía siempre que sea posible
- El paso por referencia es más eficiente para pasar estructuras de tamaño significativo

Nombres de Subprogramas como parámetros

En algunas ocasiones es conveniente pasar un subprograma como parámetro.

Consideraciones:

1. Se verifican los tipos de los parámetros?
2. Cuál es el entorno de referencia correcto para un subprograma que fue pasado como parámetro?

Paso de subprogramas como parámetro: Verificación de tipos

C y C++: Las funciones no se pueden pasar como parámetros pero punteros a funciones si pueden ser pasados; se pueden verificar los tipos

FORTRAN 95 verifica tipos

Las últimas versiones de Pascal y Ada no permiten subprogramas como parámetros; una alternativa smilar es proveida a través de la facilidad genérica de Ada.

Ambientes de referencia. Subprogramas como parámetros

Shallow binding: El ambiente de referencia de la sentencia que activa el subprogama pasado como parámetro.

- Más natural para lenguajes de alcance dinámico

Deep binding: El ambiente de la definición del subprograma pasado como parámetro.

- Más natural para lenguajes de alcance estático

Ad hoc binding: El ambiente de la sentencia de invocación que pasa el subprograma como parámetro.

Ejemplo Shallow / Deep / Ad-hoc binding

```
1  def sub1():
2    x = 1
3    def sub2():
4      print x
5    def sub3():
6      x = 3
7      sub4(sub2)
8    def sub4(f):
9      x = 4
10     f()
11   sub3()
```

Si se llama a **sub1()**, tres casos diferentes se tendran para los tres tipos de binding:

Deep binding : Toma el entorno de la función padre. En el ejemplo, no importa cual subfunción sea llamada el valor impreso de x será **1**.

Shallow binding : Toma el entorno de la función llamadora "final". Aquí la la última función que llama a **sub2** es **sub4**, entonces **sub2** debería tomar el valore de **x** inicializado en **sub4** e imprimiría **4**.

Ad-hoc binding : Toma el entorno de la función llamada. Aquí, para entender la diferencia, al mirar el código, la función que llama en **sub1** es **sub3,** y en **sub3** la función **"final"** que llama es **sub4**, por lo que en Shallow binding el valor usado es el puesto en **sub4** , mientras que en ad-hoc binding el valor de x utilizado es el asignado en **sub3** , por tanto, será impreso (en caso de ad-hoc binding) **3.**

Python soporta deep binding, por lo que la respuesta a este código será 1.

Ejemplo de: https://www.quora.com/What-are-the-differences-between-shallow-binding-deep-binding-and-ad-hoc-binding

Llamando a Subprogramas Indirectamente

Usualmente cuando existen múltiples subprogramas posibles a ser llamados y el correcto de una ejecución particular del programa no se sabe hasta la ejecución (ej. Manejo de eventos y GUIs)

En C y C++, tales llamadas se hacen utilizando punteros de funciones

Llamando a Subprogramas Indirectamente (II)

En C#, punteros a métodos son implementados como objetos llamados delegates

- Una declaración de delegate :

```
public delegate int Change(int x);
```

- Este tipo delegate, llamado `Change`, puede ser instanciado con cualquier método que toma un parámetro **`int`** y retorna un valor **`int`**

Un método: `static int fun1(int x) { … }`

Instanciado: `Change chgfun1 = new Change(fun1);`

Puede ser llamado con: `chgfun1(12);`

- Un delegate puede almacenar más que úna dirección, lo que se llama un delegado multicast (*multicast delegate)*

Cuestiones de diseño para funciones

Se permiten los side-effects?

- Los parámetros deberías ser siempre de in-mode para reducir side-effect (como Ada)

Qué tipos de valor de retorno son permitidos?

- La mayor parte de los lenguajes imperativos restringen los tipos de retorno
- C permite cualquier tipo excepto arrays y funciones
- C++ es como C pero también permite tipos definidos por el usuario
- Los métodos de Java y C# pueden retornar cualquier tipo (pero dedbido a que los métodos no sono tipos, estos no pueden ser retornados)
- Python y Ruby tratan a los métodos como objetos de primera clase, entonces estos pueden ser retornados, así como cualquier otra clase

Subprogramas Sobrecargados

Un subprograma sobrecargado es uno que tiene el mismo nombre que otro y en el mismo ambiente de referencia

- Cada versión del subprograma sobrecargado debe tener un protocolo único

C++, Java, C#, y Ada incluyen subprogramas sobrecargados predefinidos.

En Ada, el tipo de retorno de funciones sobrecargadas puede ser utilizado para resolver ambigüedades (entonces 2 funciones sobrecargadas pueden tener los mismos parámetros)

Ada, Java, C++, y C# permiten al usuario escribir múltiples versiones de subprogramas con el mismo nombre

Subprogramas Genéricos

Un subprograma genérico o polimórfico toma parámetros de diferentes tipos en diferentes activaciones.

Los subprogramas sobrecargados proveen polimorfismo *ad hoc*

Polimorfismo de subtipo (Subtype polymorphism) significa que una variable de tipo T puede tener acceso a cualquier objeto de tipo T o cualquier tipo derivado de T (lenguajes OOP)

Un subprograma que toma un parámetro genérico que es utilizado en una expresión de tipos que describe el tipo específico del parámetro provee polimorfismo paramétrico

- Un sustituto económico de tiempo de compilación para el enlace dinámico

Subprogramas Genéricos (II)

C++

- Se crean versiones de un subprograma genérico de manera implícita cuando el subprograma es nombrado en una llamada o cuando su dirección es tomada con el portador &

- Los subprogramas genéricos son precedidos por una clausula **`template`** que lista las variables genéricas, las cuales pueden ser nombres de tipos o nombres de clases

```
template <class Type>
  Type max(Type first, Type second) {
  return first > second ? first : second;
  }
```

Subprogramas Genéricos (III)

Java 5.0

- Existen diferencias entre genéricos de Java 5.0 y los de C++:

1. Los parámetros genéricos de Java 5.0 deben ser clases

2. Los métodos genéricos de Java 5.0 son instanciados sólo una vez como métodos genéricos reales

3. Se pueden aplicar restricciones a los rangos de las clases que pueden ser pasadas a los métodos genéricos como parámetros genéricos

4. Parámetros genéricos utilizan un tipo comodín

Subprogramas Genéricos (IV)

Java 5.0 (continuación)

```
public static <T> T doIt(T[] list) { … }
```

- El parámetro es un array de elementos genéricos (`T` es el nombre del tipo)
- Una llamada:

```
doIt<String>(myList);
```

Los parámetros genéricos pueden tener límites:

```
public static <T extends Comparable> T
  doIt(T[] list) { … }
```

El tipo genérico debe ser ded una clase que implemena la interface `Comparable`

Subprogramas Genéricos (V)

Java 5.0 (continuación)

- Tipos comodín (Wildcard types)

`Collection<?>` es un tipo comodín para clases Collection

```
void printCollection(Collection<?> c) {
    for (Object e: c) {
      System.out.println(e);
    }
  }
```

- Funciona para cualquier clase Collection

Subprogramas Genéricos (VI)

C# 2005

- Soporta métodos genéricos que son similares a aquellos que utiliza Java 5.0
- Una diferencia: los tipos de los parámetros actuales en una llamada pueden ser omitidas si el compilador puede inferir el tipo no especificados
  - Otra – C# 2005 no soporta comodines

Subprogramas Genéricos (VII)

F#

- Infiere un tipo genérico si no puede determinar el tipo de un parámetro de una función – *automatic generalization*
- Tales tipos se denotan con un apostrofe y una letra, ej., `'a`
- Se pueden definir funciones para tener parámetros genéricos

```
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

- `%A` es un código de formato para cualquier tipo
- Estos parámetros no estas restringidos por tipo

Subprogramas Genéricos (VIII)

F# (continuación)

- Si los par;ametros de una función son utilizados con operadores aritméticos, estos son restringidos de tipo, aún si se especifica que los parámetros sean genéricos
- Debido a la inferencia de tipo y la falta de coerciones de tipo, las funciones genéricas de F# son mucho menos útiles que en C++, Java 5.0+, y C# 2005+

Operadores sobrecargados definidos por el usuario

En Ada, C++, Python, y Ruby se pueden sobrecargar los operadores.

Ejemplo en Ada

```
Function "*"(A,B: in Vec_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 "*";
…
c = a * b; -- a, b, and c are of type Vec_Type
```

Operadores sobrecargados definidos por el usuario

Un ejemplo en Python

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

Uso: Para computar `x + y`, `x.__add__(y)`

Closures

Una cerradura (*closure*) es un subprograma y el entorno de referencia donde esta fue definido

- El entorno de referencia se necesita si el subprograma puede ser llamado desde cualquier punto arbitrario en el programa
- Un lenguaje con alcance estático que no permite subprogramas anidados no necesita cerraduras
- Los Closures se requieren sólo si un subprogrma pueded acceder a variables en los alcances anidados y este puede ser llamado de cualquier parte
- Para soportar closures, una implementación puede necesitar proveer un acceso ilimitado a algunas variables (debido a que un subprograma puede acceder a una variable nolocal que normalmente no se encuentra viva)

Closures (continuación)

Una cerradura JavaScript :

```
function makeAdder(x) {
    return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) +
               "<br />");
document.write("add 5 to 20: " + add5(20) +
               "<br />");
```

- La cerradura es la función anónima retornada por `makeAdder`

Closures (continuación)

C#

- Se puede escribir la misma cerradura en C# usando un delegado anónimo anidado
- `Func<int, int>` (el tipo de retorno) especificica un delegate que toma un **`int`** como parametro y retorna un **`int`**

```
static Func<int, int> makeAdder(int x) {
    return delegate(int y) {return x + y;};
}
...
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

Co-rutinas

Una co-rutina es un subprograma **que puede tener múltiples puntos de acceso y controla el flujo internamente** – Soportado de forma directa en Lua

También se lo llama control simétrico: llamador y llamado actúan más sobre la base de igualdad

Una invocación a una co-rutina se denomina *resume*

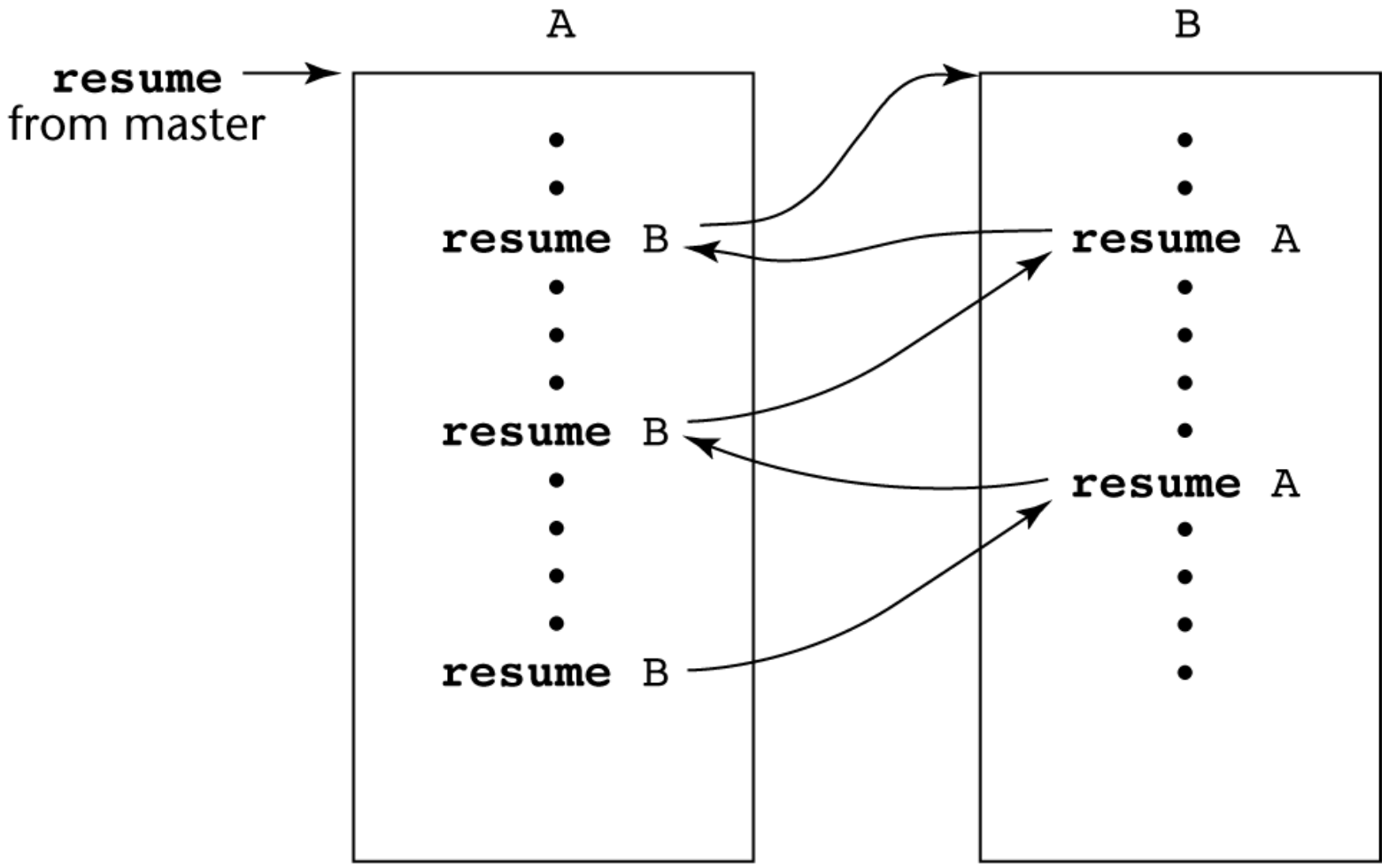
La primera llamada *resume* de una co-rutina es también su comienzo, pero siguientes llamadas entran en la sentencia posterior a la última sentencia ejecutada en la co-rutina.

Las corrutinas de forma repetida se invocan unas a otras, posiblemente por siempre

Las co-rutinas proveen *quasi-concurrent execution* de unidades del programa, su ejecución es entrecortada pero no solapada.

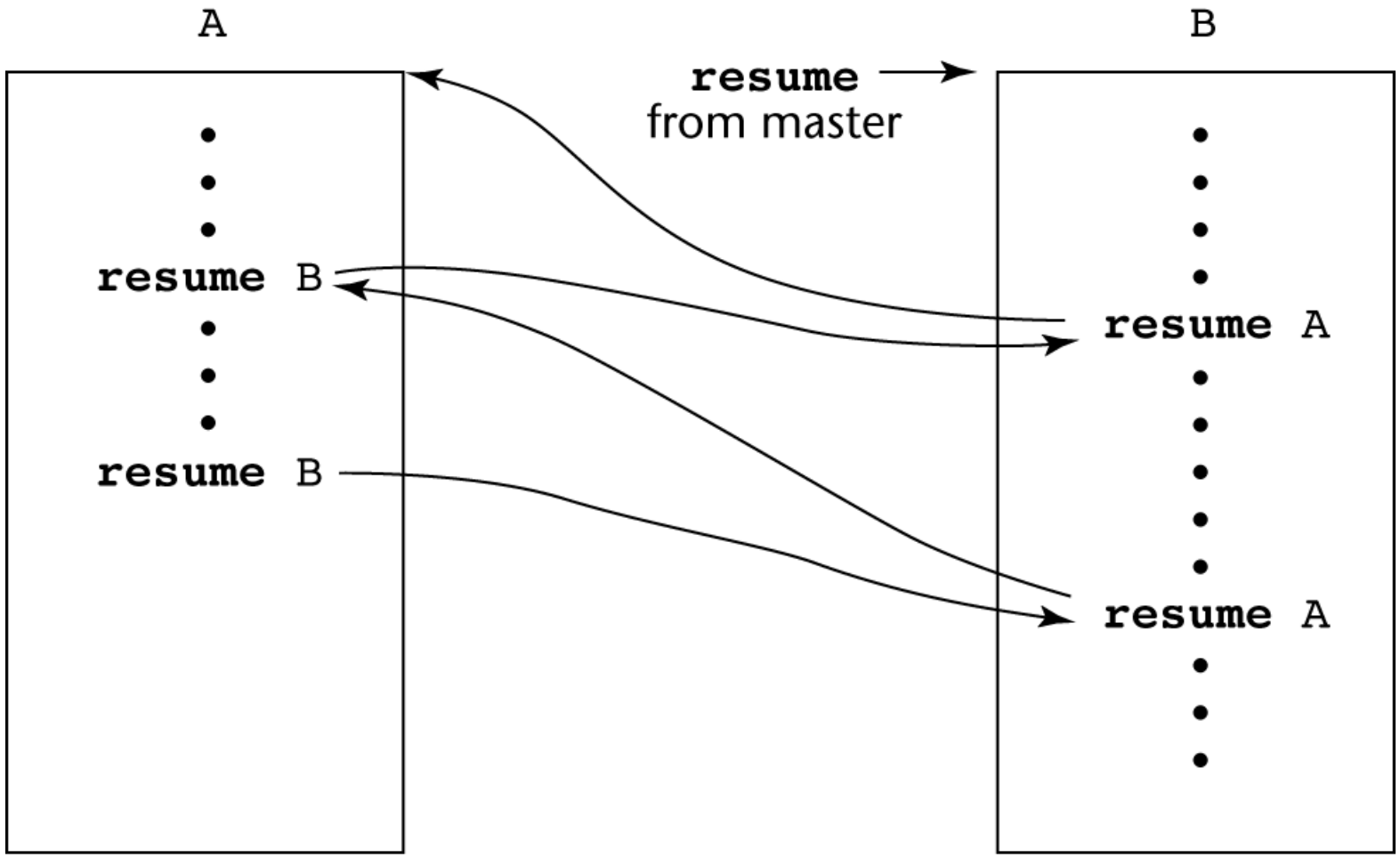
https://www.tutorialspoint.com/lua/lua_coroutines.htm

Co-rutinas: Posible Control de Ejecución



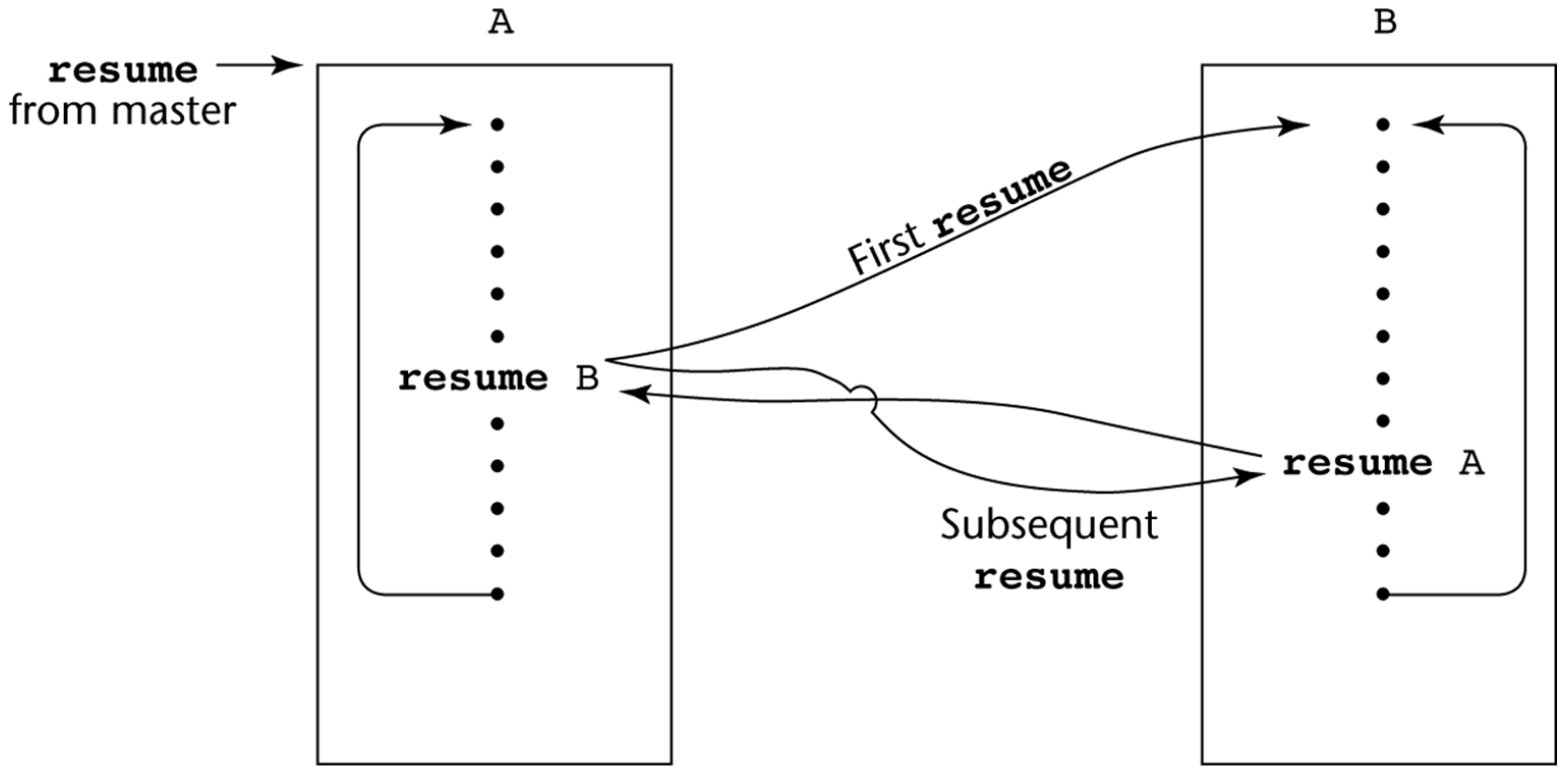
(a)

Co-rutinas: Posible Control de Ejecución



(b)

Co-rutinas: Posible Control de Ejecución con ciclos



Resumen

- Una definición de un subprograma define acciones representadas por el subprograma.
- Pueden ser funciones o procedimientos
- Variables locales en subprogramas pueden ser dinámicas mediante pila o estáticas
- Modelos de paso de parámetros: Modos In, Out, InOut
- Algunos lenguajes permiten sobrecarga de operadores
- Una cerradura es un subprograma más su entorno de referencia
- Los subprogramas pueden ser genéricos: con poliformismo ad hoc o parametrizable
- Una co-rutina es un subprograma especial con múltiples puntos de acceso