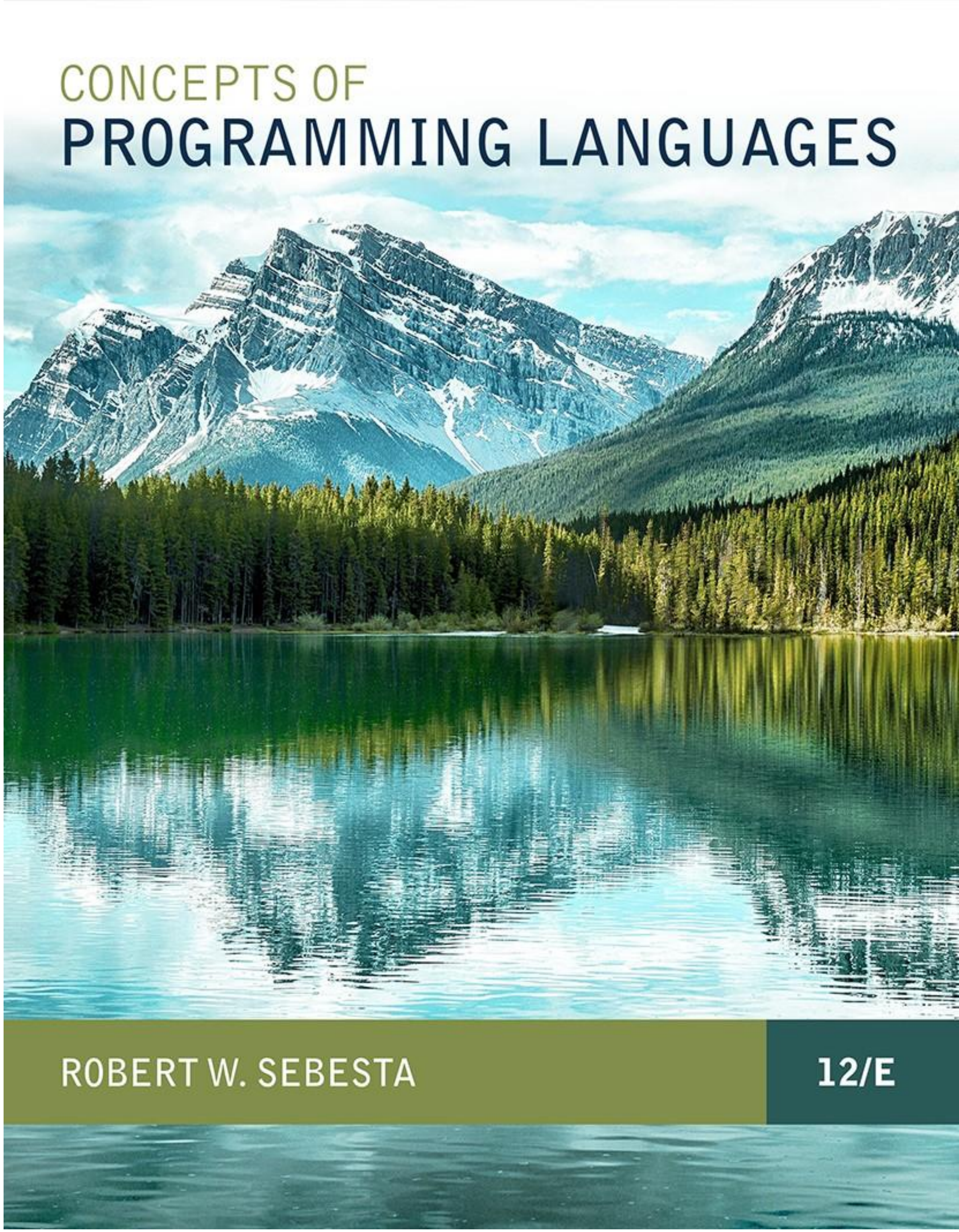
Chapter 9

Subprograms



ISBN 0-321-49362-1

Chapter 9 Topics

- Introduction
- Fundamentals of Subprograms
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded Operators
- Closures
- Coroutines

Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
    - Discussed in this chapter
  - Data abstraction
    - Emphasized in the1980s

Fundamentals of Subprograms

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram’s execution terminates

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type

Basic Definitions (continued)

- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
- A *formal parameter* (also called *parameter*) is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* (also called *argument*) represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

- Positional
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter's names

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed).

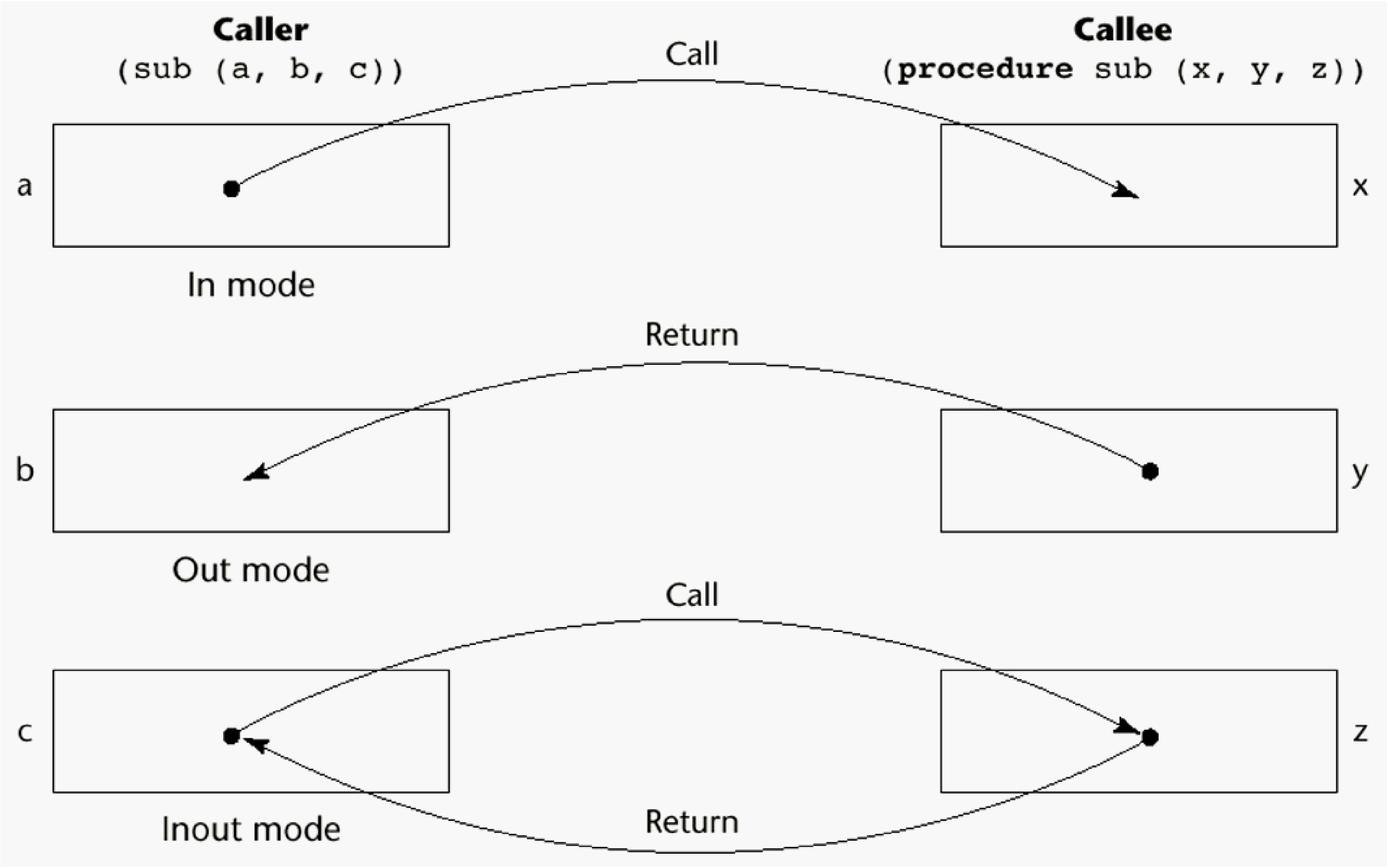
Procedures and Functions

- There are two categories of subprograms
  - *Procedures* are collection of statements that define parameterized computations
    - no return values
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
    - have return values

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by **copying**
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller
  - Require extra storage location and copy operation

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters

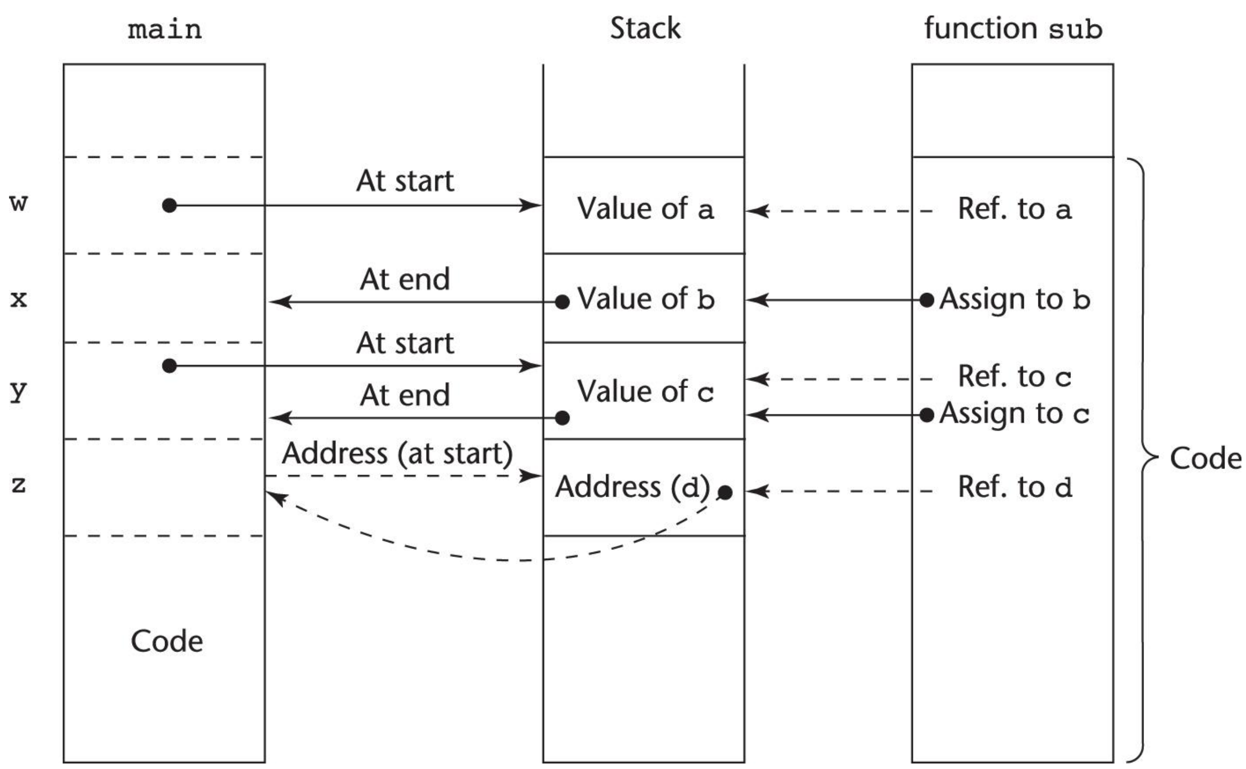
Pass-by-Name (Inout Mode)

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated

Implementing Parameter-Passing Methods

- In most languages, parameter communication takes place through the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack

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)

Parameter Passing Methods of Major Languages

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All non-object parameters are passed by value.
    - So, no method can change any of these parameters
  - Object parameters are passed by reference

Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters

Parameters that are Subprogram Names: Referencing Environment

- *Shallow binding*: The environment of the **call statement** that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- *Deep binding*: The environment of the **definition** of the passed subprogram
  - Most natural for static-scoped languages
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

Calling Subprograms Indirectly

- Usually when there are several possible subprograms to be called and the correct one on a particular run of the program is not known until execution (e.g., event handling and GUIs)
- In C and C++, such calls are made through function pointers

Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

Generic Subprograms

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- Overloaded subprograms provide *ad hoc polymorphism*
- *Subtype polymorphism* means that a variable of type T can access any object of type T or any type derived from T (OOP languages)
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*

Generic Subprograms (continued)

- C++
  - Generic subprograms are preceded by a **`template`** clause that lists the generic variables, which can be type names or class names

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

Generic Subprograms (continued)

- Java 5.0

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

  - The parameter is an array of generic elements (`T` is the name of the type)
  - A call:

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

Generic parameters can have bounds:

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

The generic type must be of a class that implements the `Comparable` interface

Generic Subprograms (continued)

- Java 5.0 (continued)
  - Wildcard types

    `Collection<?>` is a wildcard type for collection classes

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

  - Works for any collection class

User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- A Python example

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

Use: To compute `x + y`, `x.__add__(y)`

Closures

- A *closure* is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere

Closures (continued)

- A JavaScript closure:

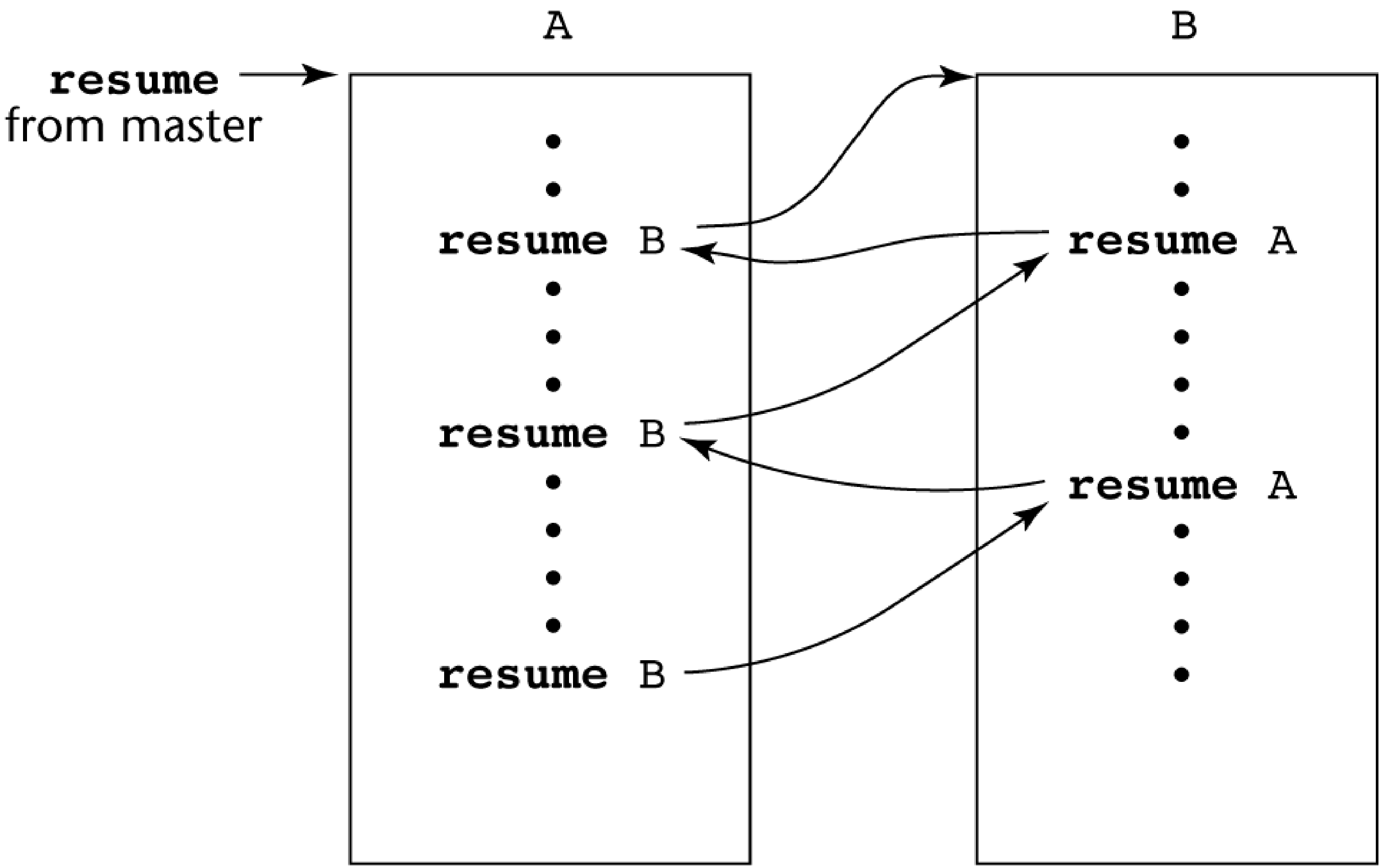
```
function makeAdder(x) {
  return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) +
               "<br />");
document.write("add 5 to 20: " + add5(20) +
               "<br />");
```

  – The closure is the anonymous function returned by `makeAdder`

Coroutines

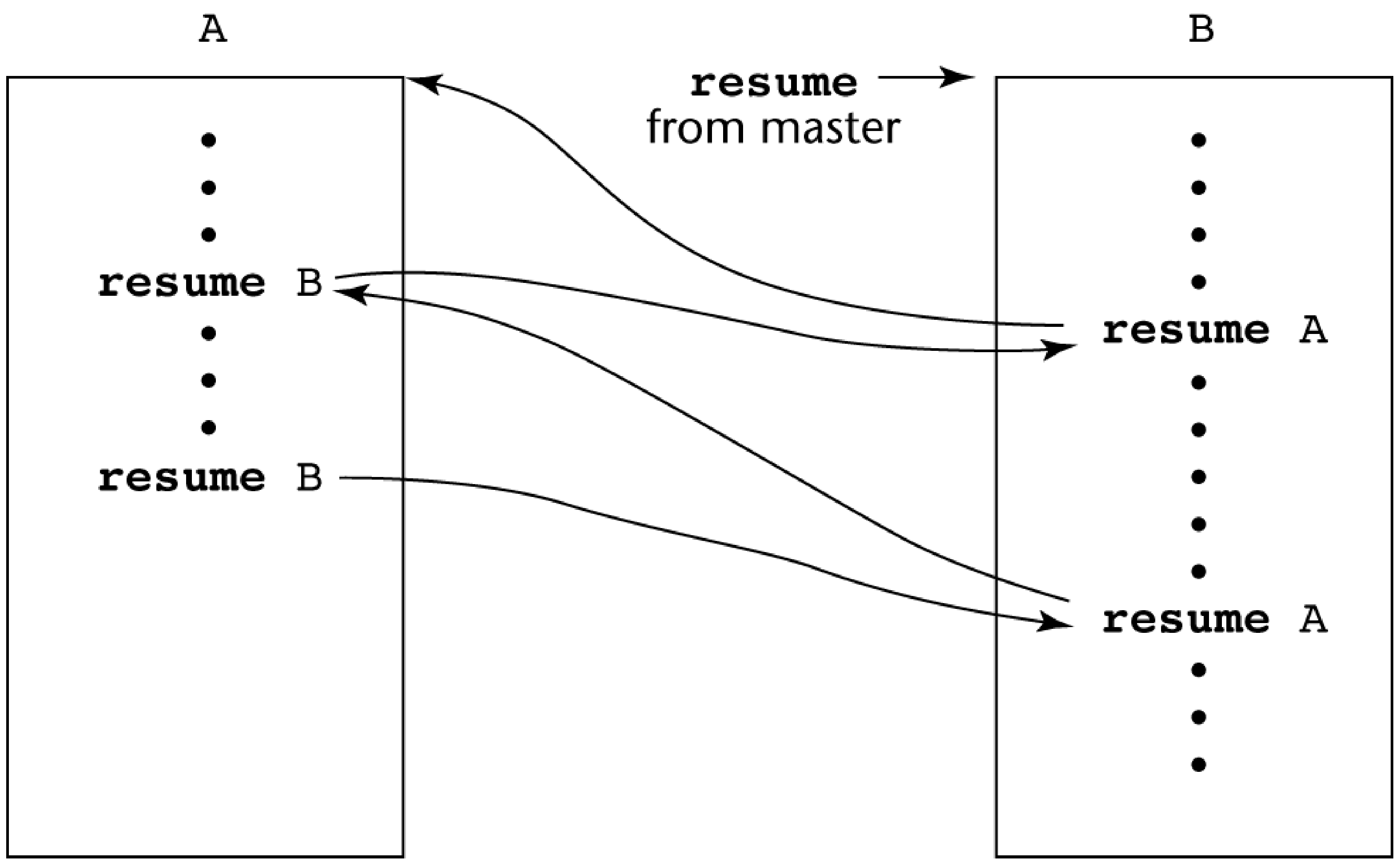
- A *coroutine* is a subprogram that has multiple entries and controls them itself
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

Coroutines Illustrated: Possible Execution Controls



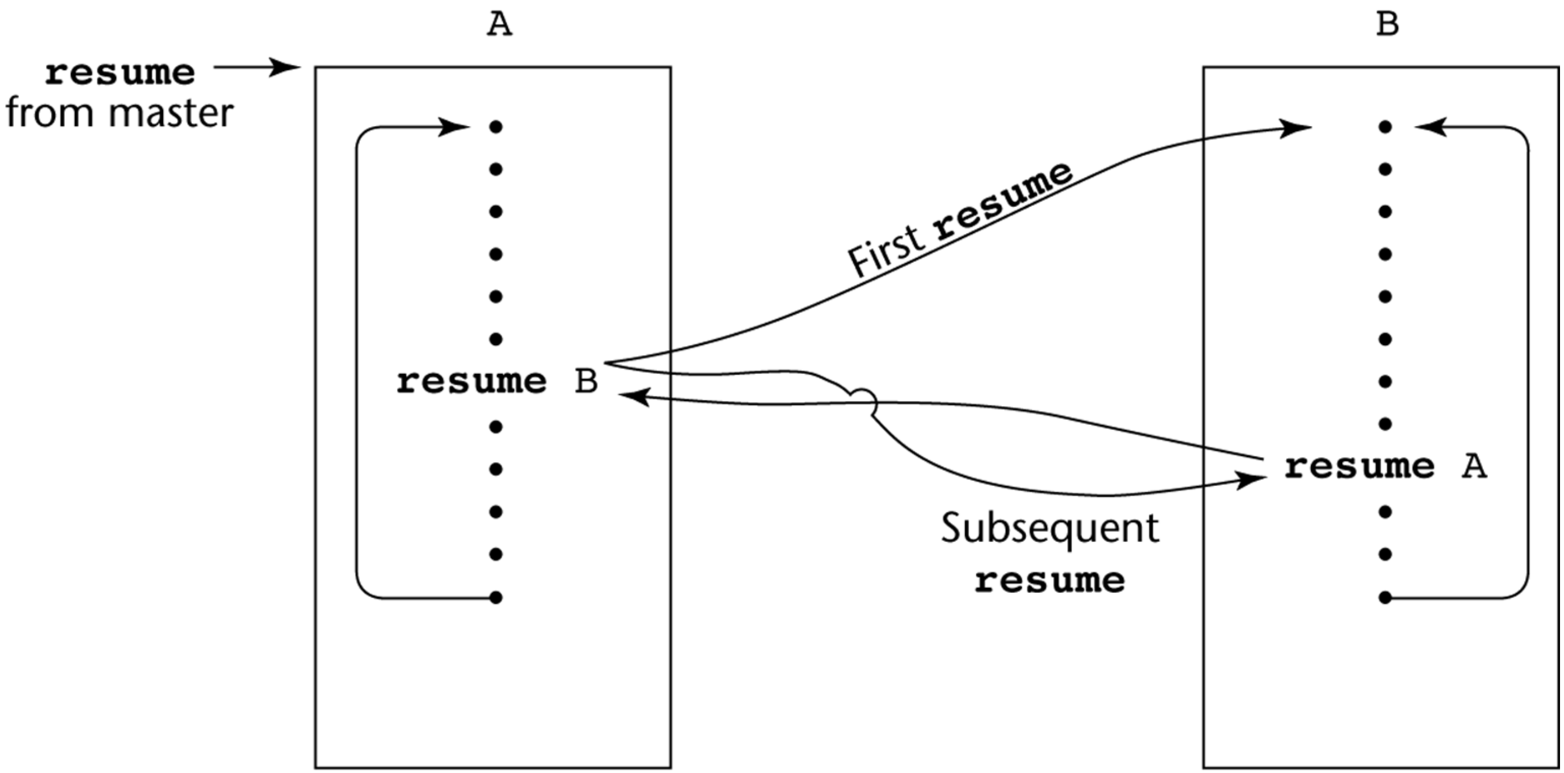
(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops



Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A closure is a subprogram and its referencing environment
- A coroutine is a special subprogram with multiple entries