Principles of Programming Languages

subprograms

Overview

- Fundamentals of Subprograms (Sec. 9.2 - 9.4)
- Parameter-Passing Methods (Sec. 9.5)
- Parameters That Are Subprograms (Sec. 9.6)
- Overloaded, Generic Subprograms (Sec. 9.7,9.8)
- Design Issues for Functions (Sec. 9.9)
- User-Defined Overloaded Operators (Sec. 9.10)
- Coroutines (Sec. 9.11)

Fundamentals of Subprograms

- Suppose in a program the same sequence of code appears in many different locations
  - Instead of many duplicated codes, we can use only one copy and jump to and back from that copy of code from those different locations in the program
- Subprogram:
  - Each subprogram has a single entry point
  - The calling subprogram is suspended during execution of the called subprogram
  - Control always returns to the caller when the called subprogram’s execution terminates

Fundamentals of Subprograms

- Suppose in a program the same sequence of code appears in many different locations, and they differ only in some variables used
  - We can use subprograms with parameters
- We can almost always substitute subprogram code to the location where it is called to understand what the code looks like there
- Concept of subprogram evolves to produce things like threads

Basic Definitions

- A subprogram definition describes the interface to and actions of the subprogram
  - A subprogram header is the 1st part of the definition, including name, subprogram kind, formal parameters

    ```
    void adder(parameters)
    ```

  - Number, order, types of formal parameters are called parameter profile (signature) of the subprogram
- A subprogram declaration provides parameter profile and return type, but not the body
  - Prototypes in C and C++; for compilers to see
- A subprogram call jumps to the subprogram

Parameters

- Two ways for a subprogram to gain access to the data for it to process
  - Direct access to non-local variables
  - Parameter passing
- A formal parameter is a dummy variable listed in subprogram header and used in the subprogram
  - Usually bound to storage when subprogram is called
- An actual parameter represents a value or address used in the subprogram call statement

Binding Actual to Formal Parameter

- Positional parameters
  - Binding by position: the first actual parameter is bound to the first formal parameter and so forth
- Keyword parameters: for long parameter list
  - Name of formal parameter to which an actual parameter is to be bound is specified

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

-- Python

  - Pros: Parameters can appear in any order
  - Cons: User must know the formal parameter's names

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated

```
float comput_pay (float income,
   float tax_rate, int exemptions = 1)
pay = comput_pay (20000.0, 0.15);
```

-- exemptions takes 1

- Variable numbers of parameters
  - `printf` in C

Two Categories of Subprograms

- Procedures: collection of statements that define parameterized computations
- Functions: structurally resemble procedures but are semantically modeled on math functions
  - Should be no side effects and return only one value to mimic mathematic functions
  - In practice, program functions have side effects
  - Functions define new user-defined operators

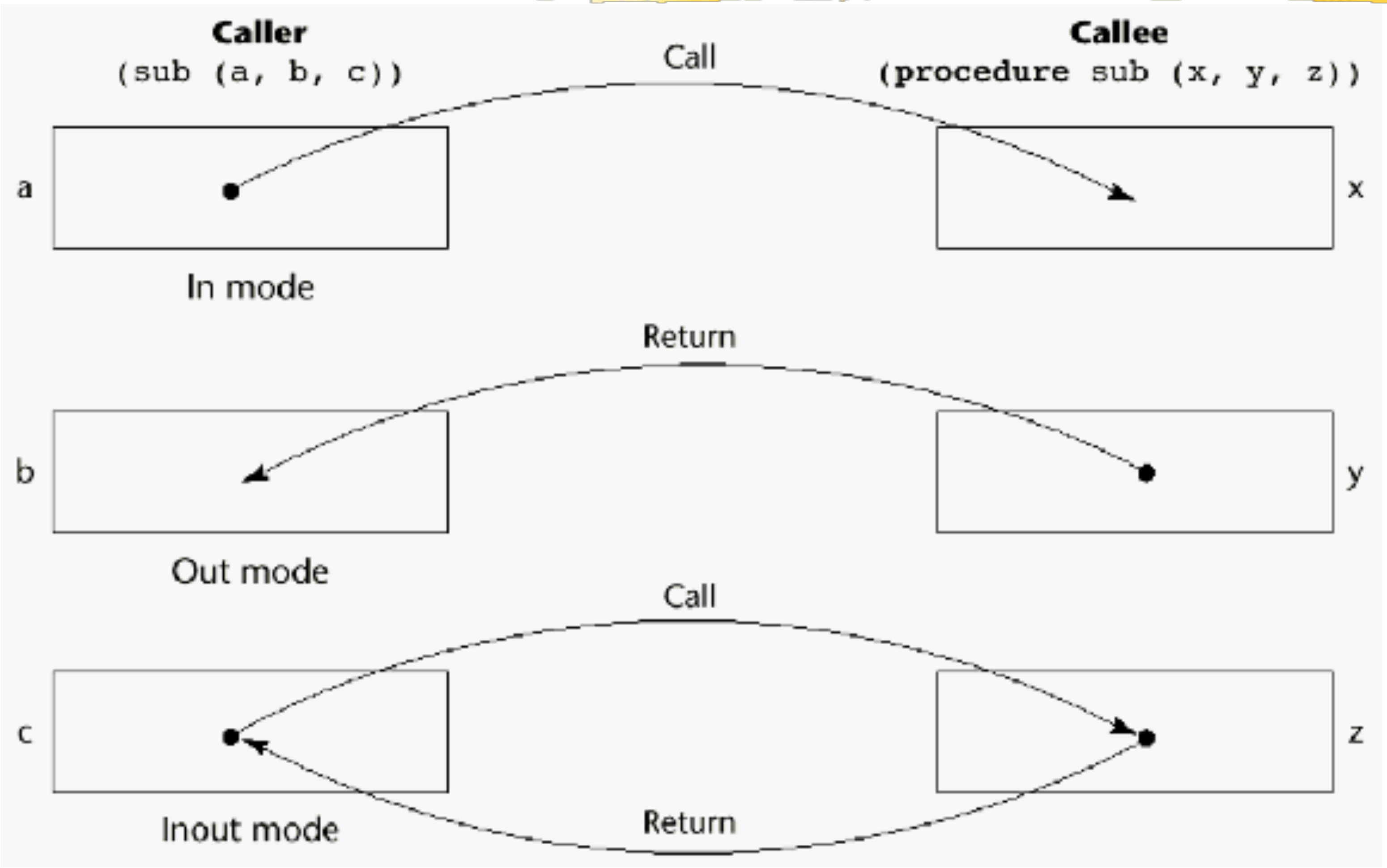
```
float power(float base, float exp)
result = 3.4 * power(10.0, x);
c.f. result = 3.4 * 10.0 ** x;
```

(Perl)

Overview

- Fundamentals of Subprograms (Sec. 9.2 – 9.4)
- Parameter-Passing Methods (Sec. 9.5)
- Parameters That Are Subprograms (Sec. 9.6)
- Overloaded, Generic Subprograms (Sec. 9.7,9.8)
- Design Issues for Functions (Sec. 9.9)
- User-Defined Overloaded Operators (Sec. 9.10)
- Coroutines (Sec. 9.11)

Semantic Models of Param 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)
  - If by access path: must write-protect in callee and accesses cost more (indirect addressing)

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, by physical moving/copying
  - Require extra storage location and copy operation
- Potential problem: **`sub(p1, p1)`**
  - With the two corresponding formal parameters having different names, whichever formal parameter is copied back last will represent current value of p1

Pass-by-Value-Result (Inout Mode)

- A combination of pass-by-value and pass-by-result, sometimes called pass-by-copy
  - The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable
- Disadvantages:
  - Those of pass-by-result and pass-by-value

Pass-by-Reference (Inout Mode)

- Pass an access path, also called pass-by-sharing
  - The called subprogram is allowed to access the actual parameter in the calling subprogram unit
- Advantage:
  - Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters due to indirect addressing
  - Potentials for unwanted changes to actual param.
  - Unwanted aliases (access to non-local)

Pass-by-Reference (Inout Mode)

- Aliases due to pass-by-reference:
  - Collisions between actual parameters, e.g., in C++

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

  - Collisions between array and array elements

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

  - Collisions between formal parameters and nonlocal variables that are visible

```
int *global;
void main() { ... sub(global); ...}
void sub(int *param) {...}
```

`global` and **`param`** are aliases inside sub()

Parameter Passing Methods

- In most languages, parameter communication takes place thru the run-time stack
  - Pass-by-reference are the simplest to implement; only an address is placed in stack
- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers
- Java
  - All parameters are passed by value
  - Object parameters are passed by reference

Type Checking Parameters

- Very important for reliability
- FORTRAN 77 and original C: none
- Pascal, FORTRAN 90, Java, Ada: required
- ANSI C and C++: choice is made by the user
    - Prototypes
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

Multidimensional Arrays as Param

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, compiler needs to know declared size of that array to build storage mapping function
  - A storage-mapping function for row-major matrices:
    address(mat[i,j])=
    address(mat[0,0]) + i * #_columns + j
    - Only needs to know #_columns

Multidimensional Arrays as Param

- For C and C++:
  - Programmer is required to include the declared sizes of all but the first subscript in the actual parameter

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

  - Disallows writing flexible subprograms
  - Solution: pass a pointer to the array and sizes of the dimensions as other parameters; user must include storage mapping function in terms of size parameters

Multidimensional Arrays as Param

- Java and C#
  - Arrays are objects; they are all single-dimensioned, but the elements can be arrays
  - Each array inherits a named constant (**`length`** in Java, **`Length`** in C#) that is set to the length of the array when the array object is created, e.g., in Java

```
float sumer(float mat[][]) {
  for(int row=0; row<mat.length; row++)
    for(int col=0; col<mat[row].length;
            col++)
      sum += mat[row][col];
```

Design Considerations

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

Overview

- Fundamentals of Subprograms (Sec. 9.2 – 9.4)
- Parameter-Passing Methods (Sec. 9.5)
- Parameters That Are Subprograms (Sec. 9.6)
- Overloaded, Generic Subprograms (Sec. 9.7,9.8)
- Design Issues for Functions (Sec. 9.9)
- User-Defined Overloaded Operators (Sec. 9.10)
- Coroutines (Sec. 9.11)

Subprogram Names as Parameters

- It is sometimes convenient to pass subprogram names as parameters
  → pass a computation to a subprogram
- Are parameter types checked?
  - C and C++: functions cannot be passed as parameters but pointers to functions can be passed and types of function pointers include the types of the parameters, so parameters can be type checked
  - Java does not allow method names to be passed as parameters

Referencing Environment

- For languages that allow nested subprograms, what referencing environment for executing the passed subprogram should be used?
  - 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 call statement that passed the subprogram as an actual parameter

Referencing Environment

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

Referencing env. of sub2():
- Shallow binding: sub4
  output = 4
- Deep binding: sub1
  output = 1
- Ad hoc binding: sub3
  output = 3

Overview

- Fundamentals of Subprograms (Sec. 9.2 – 9.4)
- Parameter-Passing Methods (Sec. 9.5)
- Parameters That Are Subprograms (Sec. 9.6)
- Overloaded, Generic Subprograms (Sec. 9.7,9.8)
- Design Issues for Functions (Sec. 9.9)
- User-Defined Overloaded Operators (Sec. 9.10)
- Coroutines (Sec. 9.11)

Overloaded Subprograms

- One that has the same name as another subprogram in same referencing environment
  - Every version of an overloaded subprogram has a unique protocol; meaning decided by actual param
- Problem with parameter coercion:
  - No method's parameter profile matches actual parameters, but several methods can match through coercion, then which method should be used?
- Problem with default parameters:

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

Generic Subprograms

- Software reuse → create one subprogram that works on different types of data, e.g., sorting on arrays of different element types
- A polymorphic subprogram takes parameters of different types on different activations
  - Overloaded subprograms provide ad hoc polymorphism
  - Parametric polymorphism: a subprogram that takes a generic (type-less) parameter that is used in a type expression that describes the type of the parameters

Generic Subprograms: C++

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

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

  - Can be instantiated with any type for which operator > is defined, e.g., **`int`**

```
int a,b,c;        c = max(a, b);
```

  - Template functions are instantiated implicitly either when the function is named in a call or when its address is taken with the **`&`** operator

Overview

- Fundamentals of Subprograms (Sec. 9.2 - 9.4)
- Parameter-Passing Methods (Sec. 9.5)
- Parameters That Are Subprograms (Sec. 9.6)
- Overloaded, Generic Subprograms (Sec. 9.7,9.8)
- Design Issues for Functions (Sec. 9.9)
- User-Defined Overloaded Operators (Sec. 9.10)
- Coroutines (Sec. 9.11)

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?

  Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Java and C# methods can return any type (but methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class

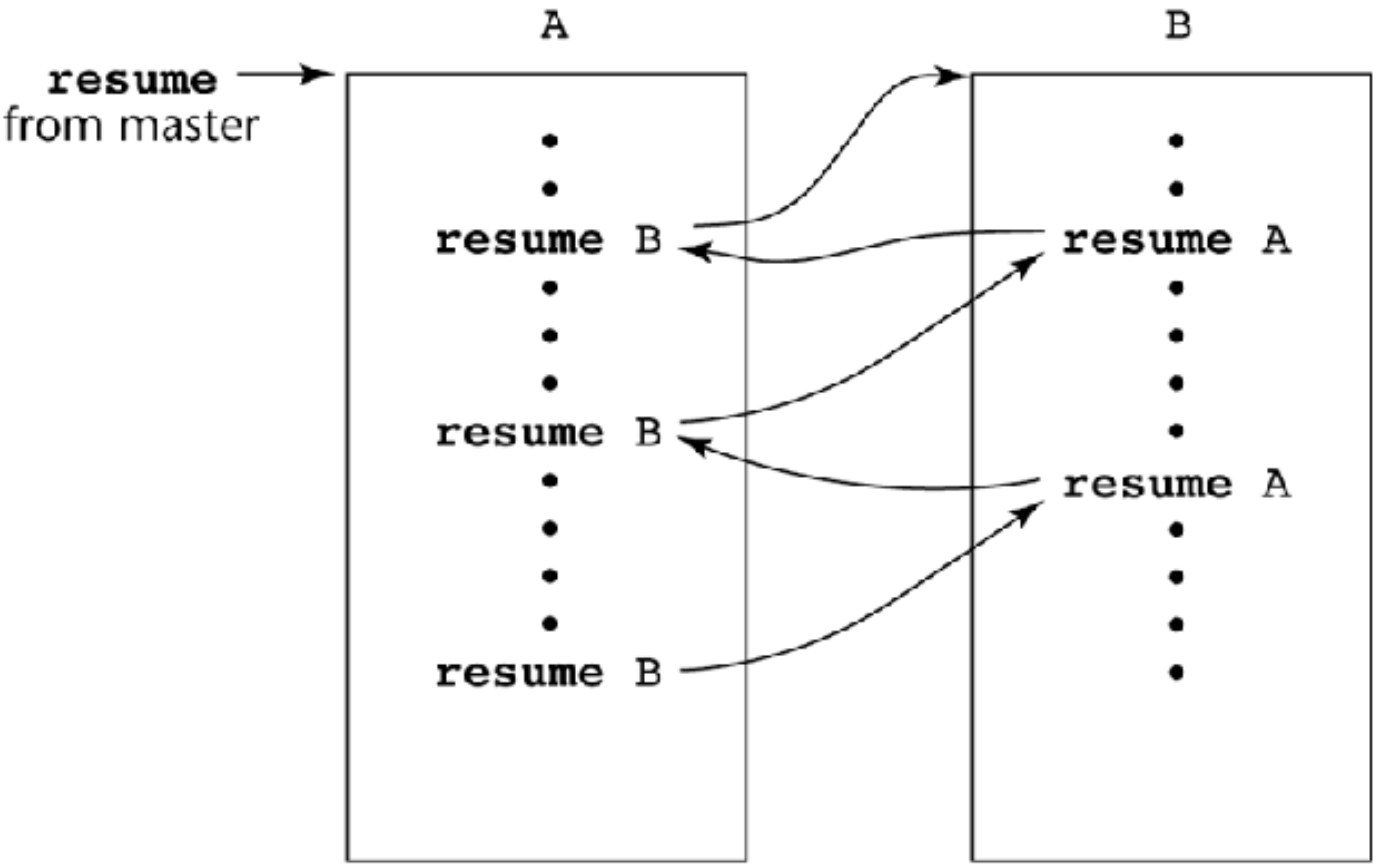
Overview

- Fundamentals of Subprograms (Sec. 9.2 – 9.4)
- Parameter-Passing Methods (Sec. 9.5)
- Parameters That Are Subprograms (Sec. 9.6)
- Overloaded, Generic Subprograms (Sec. 9.7,9.8)
- Design Issues for Functions (Sec. 9.9)
- User-Defined Overloaded Operators (Sec. 9.10)
- Coroutines (Sec. 9.11)

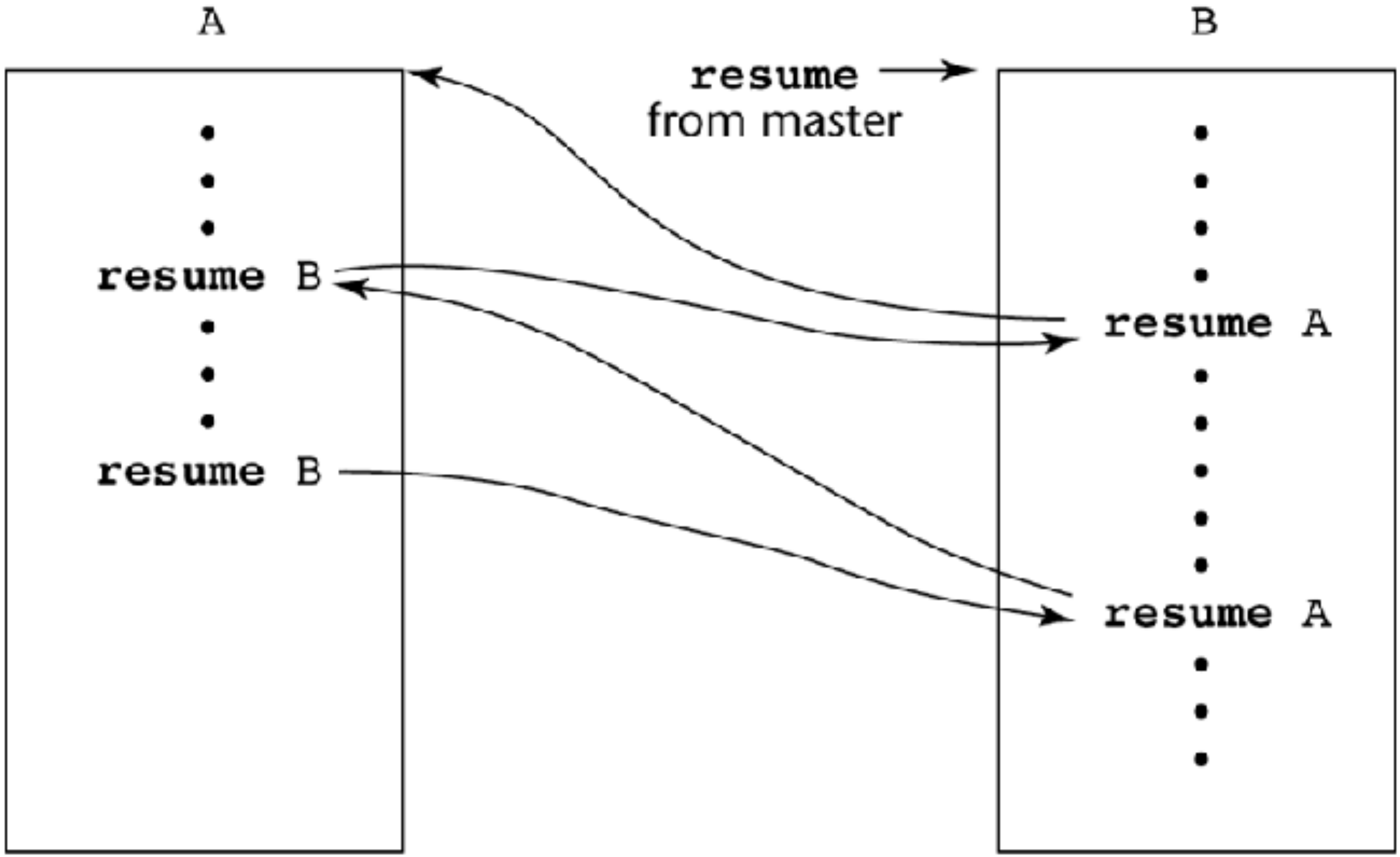
Coroutines

- A subprogram that has multiple entries and controls them itself – supported directly in Lua
  - Caller and called 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
  - Repeatedly resume each other, possibly forever
- Provide quasi-concurrent execution of program units; execution interleaved, but not overlapped

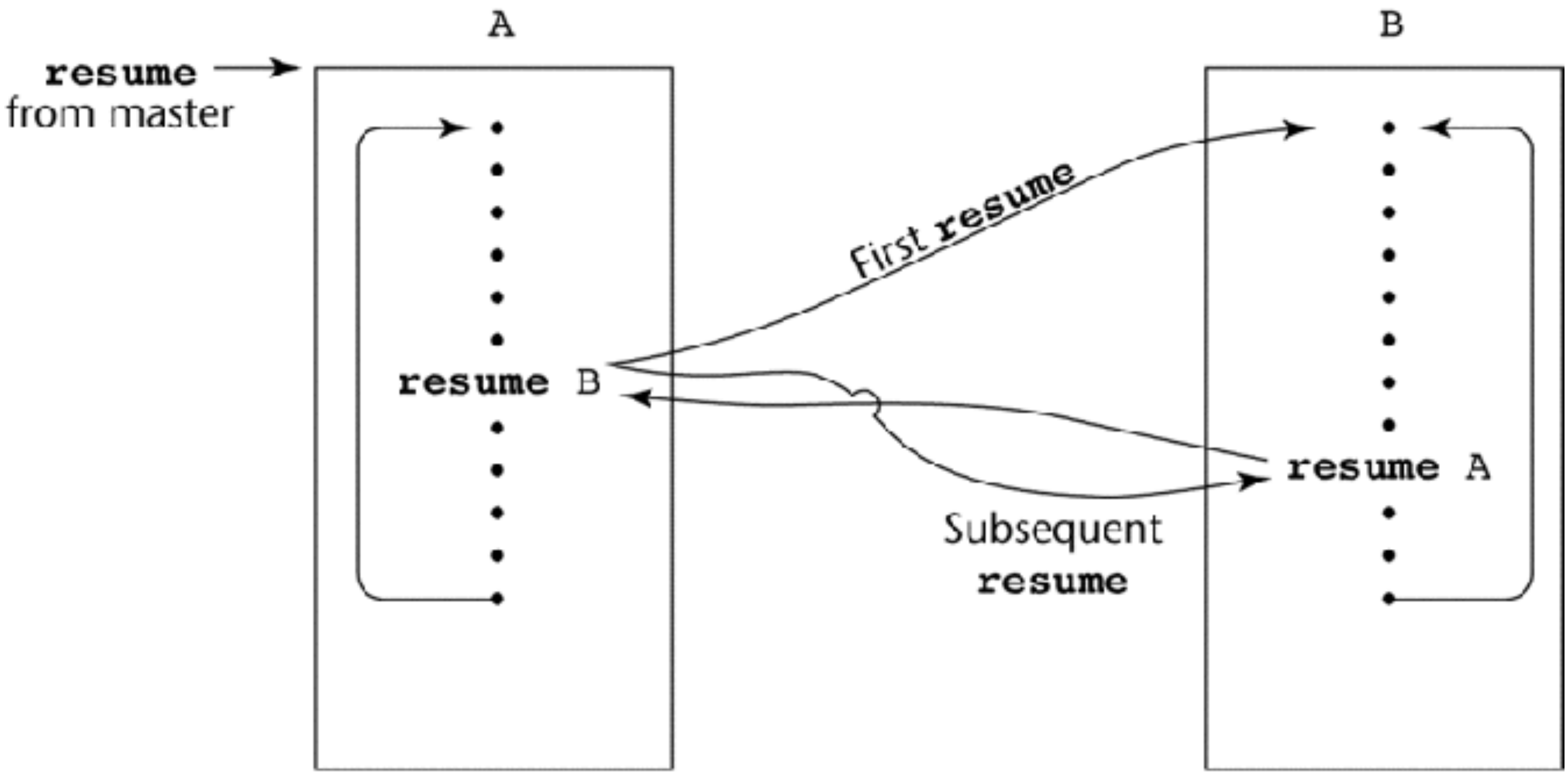
Possible Execution Controls



Possible Execution Controls



Possible Execution Controls with Loops



Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- A coroutine is a special subprogram with multiple entries