Subprograms

BBM 301 – Programming Languages

Fundamentals of Subprograms

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
  - Therefore, only one subprogram is in execution at a given time
- 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
  - In Python, function definitions are executable; in all other languages, they are non-executable

```
if ….
  def fun1(…);
else
  def fun2(…);
   …
```

Basic Definitions

- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters

  FORTRAN example:

  ```
  SUBROUTINE name (parameters)
  ```

  C example:

  ```
  void adder(parameters)
  ```

- A *subprogram call* is an explicit request that the subprogram be executed

Basic Definitions (cont’d.)

- 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 (cont'd.)

- 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* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* 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

Parameters

Example in Ada,

```
SUMER (LENGTH => 10,
         LIST => ARR,
         SUM => ARR_SUM);
```

Formal parameters: `LENGTH, LIST, SUM`.

Actual parameters: `10, ARR, ARR_SUM`.

The programmer doesn't have to know the order of the formal parameters.

But, must know the names of the formal parameters.

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

Ada example:

```
function Comp_Pay (Income: Float;
                   Examptions: Integer := 1;
                   Tax_Rate: Float) return Float;
```

Therefore, the call doesn't have to provide values for all parameters.

A sample call may be

```
Pay := Comp_Pay (2000.0, Tax_Rate => 0.23);
```

Formal Parameters

- C# allows methods to accept a variable number of parameters, as long as they are of the same type.
- The call can send either an array or a list of expressions, whose values are placed in an array by the compiler

```
public void DisplayList(params int[] list) {
       foreach (int next in list) {
              Console.WriteLine("Next value {0}", next);
       }
}
Myclass myObject = new Myclass;
int[] myList = new int[6] {2, 4, 6, 8, 10, 12};
```

DisplayList could be called with either of the following:

```
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3 * x - 1, 17);
```

Formal Parameters

- Lua uses a simple mechanism for supporting a variable number of parameters—represented by an ellipsis (. . .).
- This ellipsis can be treated as an array or as a list of values that can be assigned to a list of variables.

```
function multiply (. . .)
      local product = 1
      for i, next in ipairs{. . .} do
            product = product * next
      end
      return sum
end
```

Procedures and Functions

- There are two categories of subprograms
  - *Procedures* are collection of statements that define parameterized computations, they do not return values
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions, they return values
    - They are expected to produce no side effects
    - In practice, program functions have side effects
- In most languages that do not include procedures as a separate form of subprogram, functions can be defined not to return values and they can be used as procedures.

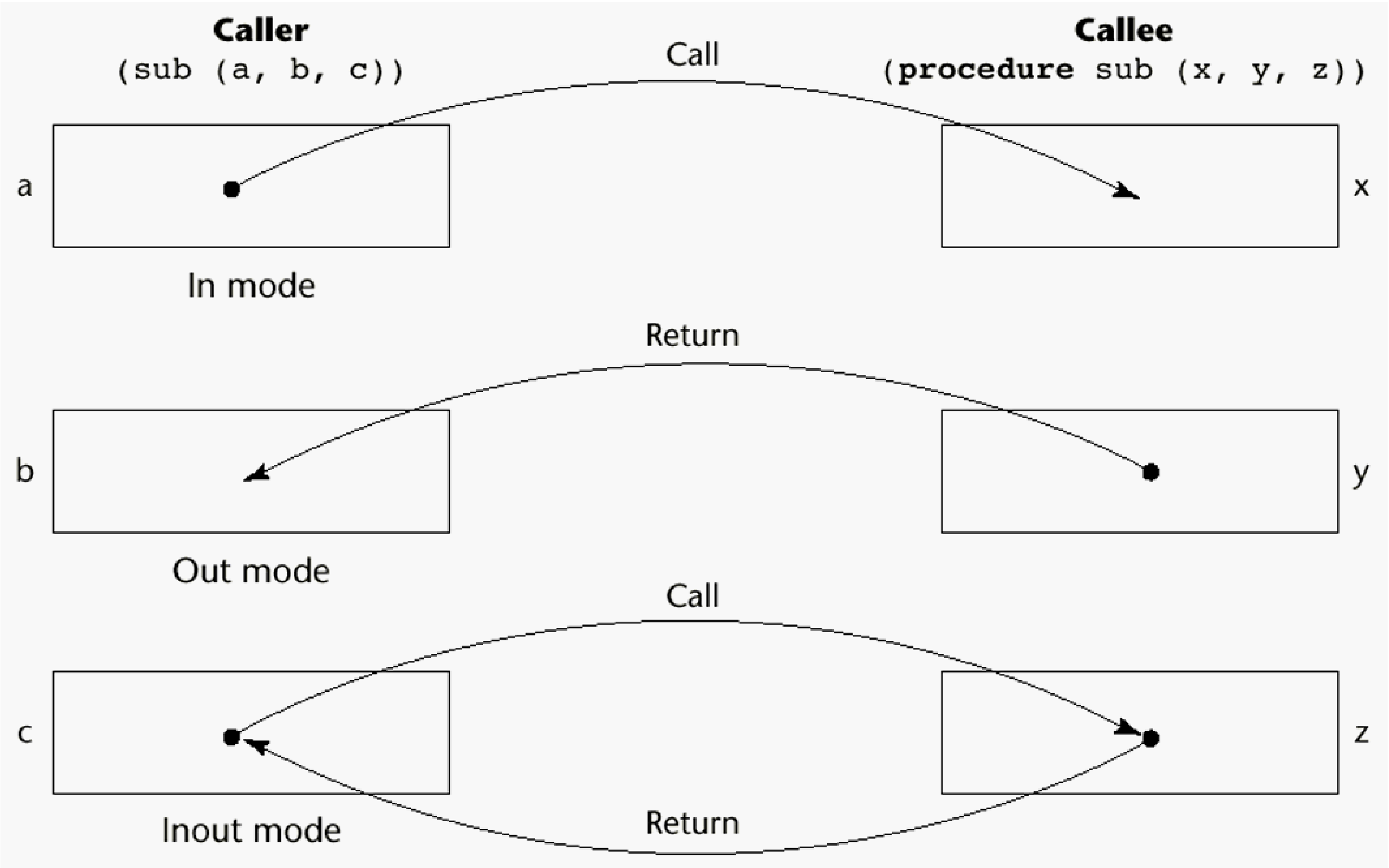
Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprogram be generic?

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Parameter Passing Methods

- Ways in which parameters are transmitted to and/or from called subprograms
  - Pass-by-value
  - Pass-by-result
  - Pass-by-value-result
  - Pass-by-reference
  - Pass-by-name

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Advantages: Actual variable is protected.*
  - *Disadvantages*: additional storage is required (stored twice) and the actual move can be costly (for large parameters – such as arrays)

Pass-by-Result (Out Mode)

- 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
- Potential problems: Parameter collision
  - `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of p1
  - `sub(list[sub], sub);` Compute address of list[sub] at the beginning of the subprogram or end?

Pass-by-Result (Out Mode)

Problem: Actual parameter collision definition:

```
subprogram sub(x, y) { x <- 3 ; y <-5;}
call:
sub(p, p)
```

what is the value of p here ? (3 or 5?)

- The values of x and y will be copied back to p. Which ever is assigned last will determine the value of p.
- The order is important
- The order is implementation dependent ⇒ Portability problems.

Pass-by-Result (Out Mode)

Problem: Time to evaluate the address of the actual parameter

- at the time of the call
- at the time of the return

- The decision is up to the implemention.

Definition:

```
subprogram sub(x)
i <-5                  is changed as a global variable here
x <- ..
call:
i <- 3
sub(A[i])
```

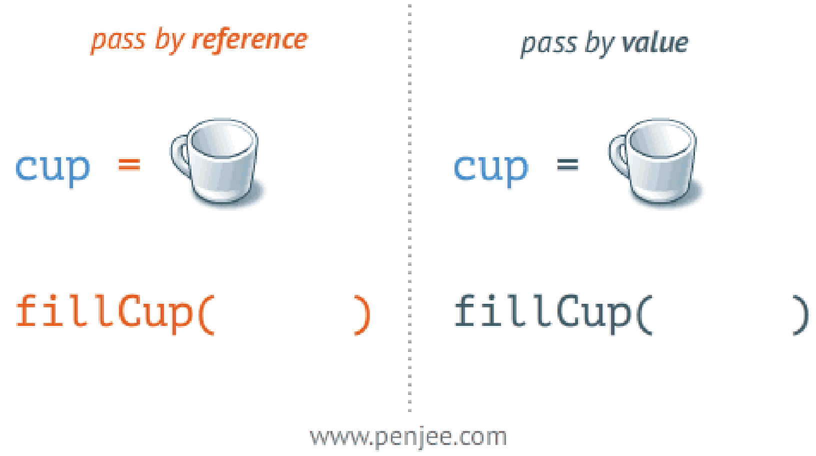
Is A[3] or A[5] is changed?

The decision is up to the implemention.

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
- The value of the actual parameter is used to initialize the corresponding formal parameter
  - the formal parameter acts as a local parameter
  - At termination, the value of the formal parameter is copied back.

Pass-by-Reference (Inout Mode)



- Pass an access path
- Also called pass-by-sharing
- Advantage:
  - Passing process is efficient
  - no copying
  - no duplicated storage
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

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

Pass-by-Reference

Dangerous: Actual parameter may be modified unintentionally.

Aliasing:

definition: `subprogram sub(x, y)`

call: `sub(p, p)`

Here, `x` and `y` in the subprogram are aliases.

- Another way of aliasing:

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

Here, `global` and `param` in the subprogram are aliases.

Pass by : Example

Example of call by value versus value-result versus reference

The following examples are in an Algol-like language.

```
begin
integer n;
procedure p(k: integer);
    begin
    n := n+1;
    k := k+4;
    print(n);
    end;
n := 0;
p(n);
print(n);
end;
```

OUTPUT:
call by value: 1 1
call by value-result: 1 4
call by reference: 5 5

An Example: pass-by-value-result vs. pass-by-reference

```
program foo;
var x: int;
    procedure p(y: int);
    begin
       y := y + 1;
       y := y * x;
    end
begin
    x := 2;
    p(x);
    print(x);
end
```

	pass-by-value-result		pass-by-reference	
	x	y	x	y
(entry to p)				
(after y:= y + 1)				
(at p's return)				

Pass-by-Name (Inout Mode)

- By textual substitution: Actual parameter is textually substituted for the corresponding formal parameter in all occurrences in the subprogram.
- Late binding: actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.
- Allows flexibility in late binding

Pass-by-name

- If the actual parameter is a scalar variable, then it is equivalent to pass-by-reference.
- If the actual parameter is a constant expression, then it is equivalent to pass-by-value.
- Advantage: flexibility
- Disadvantage: slow execution, difficult to implement, confusing.

Pass-by-name

```
procedure BIG;
integer GLOBAL;
integer array LIST[1:2];
procedure SUB (P: integer);
begin
P := 3;
GLOBAL := GLOBAL + 1;
P := 5;
end;
begin
LIST[1] := 2;
LIST[2] := 2;
GLOBAL := 1;
SUB(LIST[GLOBAL]);
end.
```

```
LIST[GLOBAL] :=3
GLOBAL := GLOBAL + 1
LIST[GLOBAL] :=5
```

Execution:

```
    LIST[1] :=3
    GLOBAL := 1 + 1
    LIST[2] :=5
```

Pass-by-Name Elegance: Jensen's Device

- Passing expressions into a routine so they can be repeatedly evaluated has some valuable applications.
- Consider calculations of the form: "sum *xi* × *i* for all *i* from 1 to *n*." How could a routine Sum be written so that we could express this as

```
sum(i, 1, n, x[i]*i) ?
```

- Using pass-by-reference or pass-by-value, we cannot do this because we would be passing in only a single value of x[i]*i, not an expression which can be repeatedly evaluated as "i" changes.
- Using pass-by-name, the expression x[i]*i is passed in without evaluation.

Pass-by-Name Elegance: Jensen's Device

```
real procedure Sum(j, lo, hi, Ej);
value lo, hi;
integer j, lo, hi;
real Ej;
begin
  real S; S := 0;
  for j := lo step 1 until hi do
      S := S + Ej;
  Sum := S
end;
```

- Each time through the loop, evaluation of Ej is actually the evaluation of the expression x[i]*i = x[j]*j.

Pass-By-Name Security Problem (Severe)

- A sample program:
  - procedure swap (a, b);
  - integer a, b, temp;
  - begin temp := a; a := b; b:= temp end;
- Effect of the call swap(x, y):
  - temp := x; x := y; y := temp
- Effect of the call swap(i, x[i]):
  - temp := i; i := x[i]; x[i] := temp
  - It doesn't work! For example:

Before call:	`i = 2`	`x[2] = 5`	
After call:	`i = 5`	`x[2] = 5`	`x[5] = 2`

- It is very difficult to write a correct swap procedure in Algol.

Pass by : Example

Example of call by value versus call by name

```
begin
integer n;
procedure p(k: integer);
    begin
    print(k);
    n := n+10;
    print(k);
    n := n+5;
    print(k);
    end;
n := 0;
p(n+1);
end;
```

OUTPUT
call by value: 1 1 1
call by name: 1 11 16

Pass by : Example

Example of call by reference versus call by name

This example illustrates assigning into a parameter that is passed by reference or by name

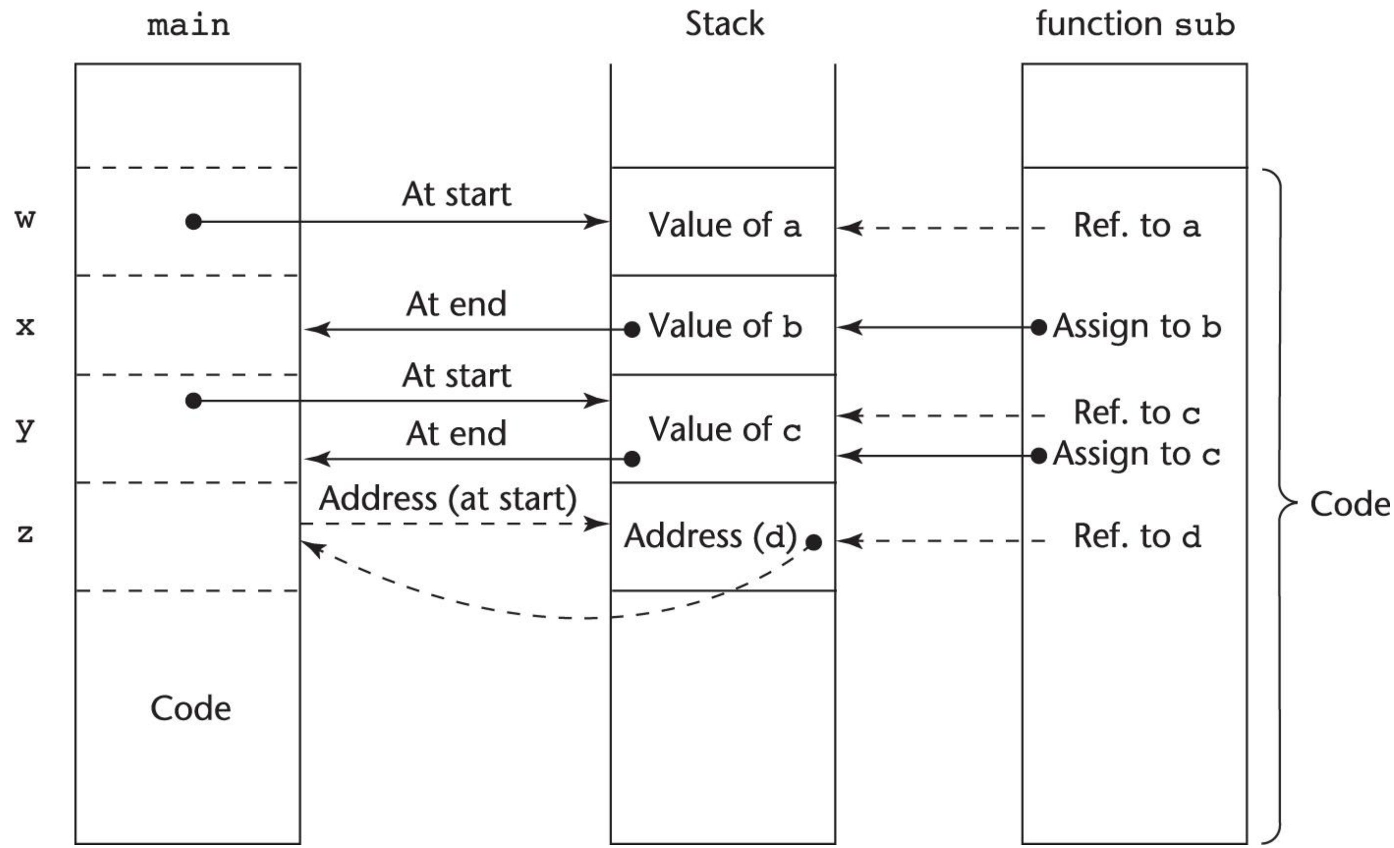
```
begin
array a[1..10] of integer;
integer n;
procedure p(b: integer);
    begin
    print(b);
    n := n+1;
    print(b);
    b := b+5;
    end;
a[1] := 10;
a[2] := 20;
a[3] := 30;
a[4] := 40;
n := 1;
p(a[n+2]);
new_line;
print(a);
end;
```

OUTPUT
call by reference: 30 30 10 20 35 40
call by name: 30 40 10 20 30 45

Implementing Parameter-Passing Methods

- In most language parameter communication takes place thru the run-time stack
- Pass-by-reference is the simplest to implement; only an address is placed in the stack
- A subtle but fatal error can occur with pass-by-reference and pass-by-value-result:
  - A formal parameter corresponding to a constant can mistakenly be changed

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 parameters are passed are passed by value
  - Object parameters are passed by reference
- Ada

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

  - Three semantics modes of parameter transmission: `in`, `out`, `in out`; `in` is the default mode
  - Formal parameters declared `out` can be assigned but not referenced; those declared `in` can be referenced but not assigned; `in out` parameters can be referenced and assigned

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
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Referencing Environment

- Q: *What is the referencing environment for executing the passed subprogram? (For non local variables)*

- *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

Parameters that are Subprograms

```
Example:
function sub1() {
  var x;             2:declared in
  function sub2() {
    window.status = x;
  } // sub2
  function sub3(){
    var x;
    x = 3;
    sub4(sub2);       3: passed in
  } // sub3
  function sub4(subx) {
    var x;
    x = 1;
    subx();          1:called by
  } // sub4
  x = 2;
  sub3();
} // sub1
```

```
Passed subprogram S2
Output:
is called by S4
is declared in S1
is passed in S3
```

1- Shallow binding

2- Deep binding

3- Ad hoc binding

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 (different number of arguments,etc)
  - The correct meaning (the correct code) to be invoked is determined by the actual parameter list.
  - In case of functions, the return type may be used to distinguish.
- 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
- *The same formal parameter can get values of different types.*
- *Ada and C++ provide Generic (Polymorphic) Subprograms*

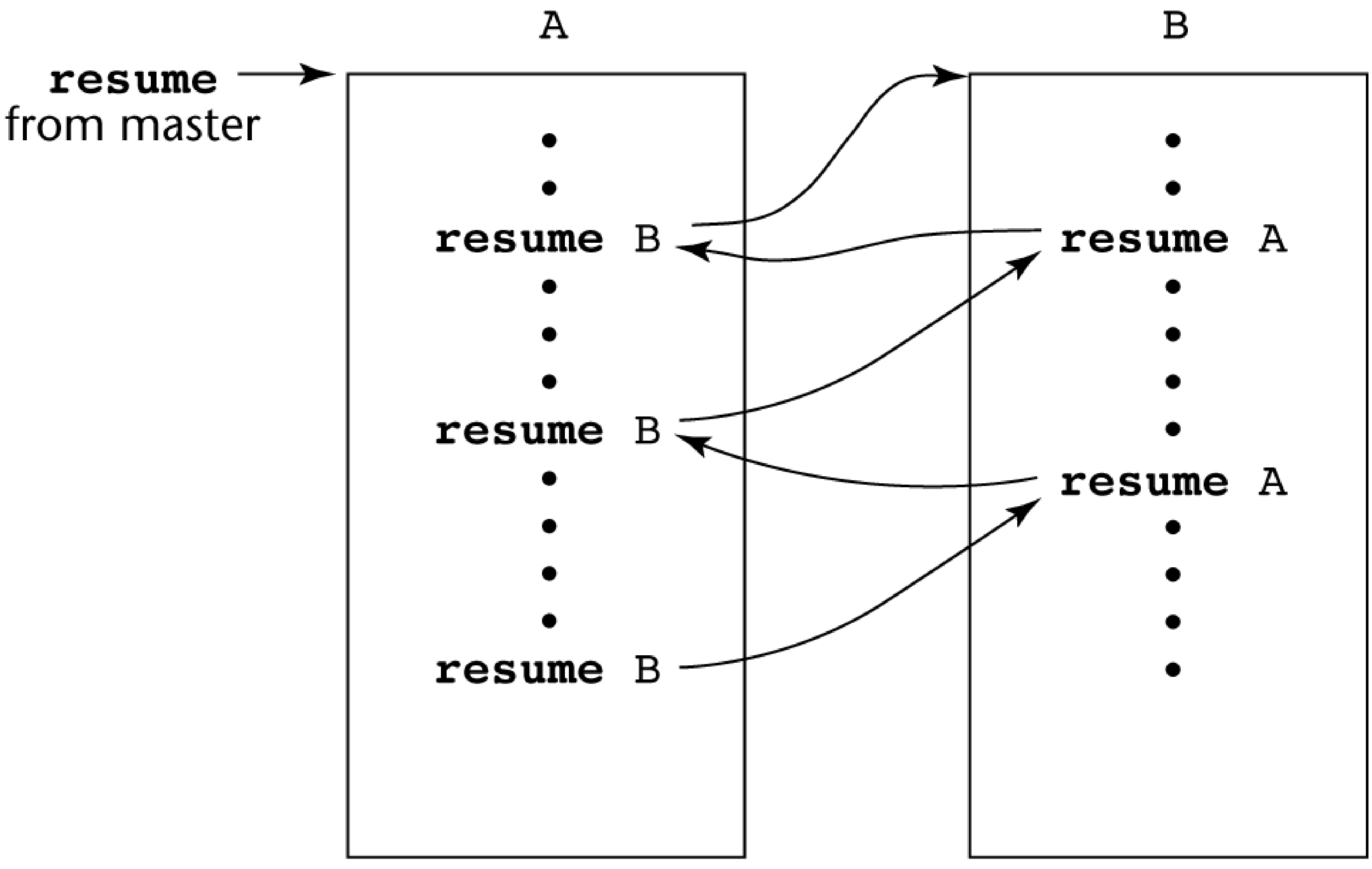
Design Issues for Functions

- 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
  - Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
  - Java and C# methods can return any type (but because 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

Coroutines

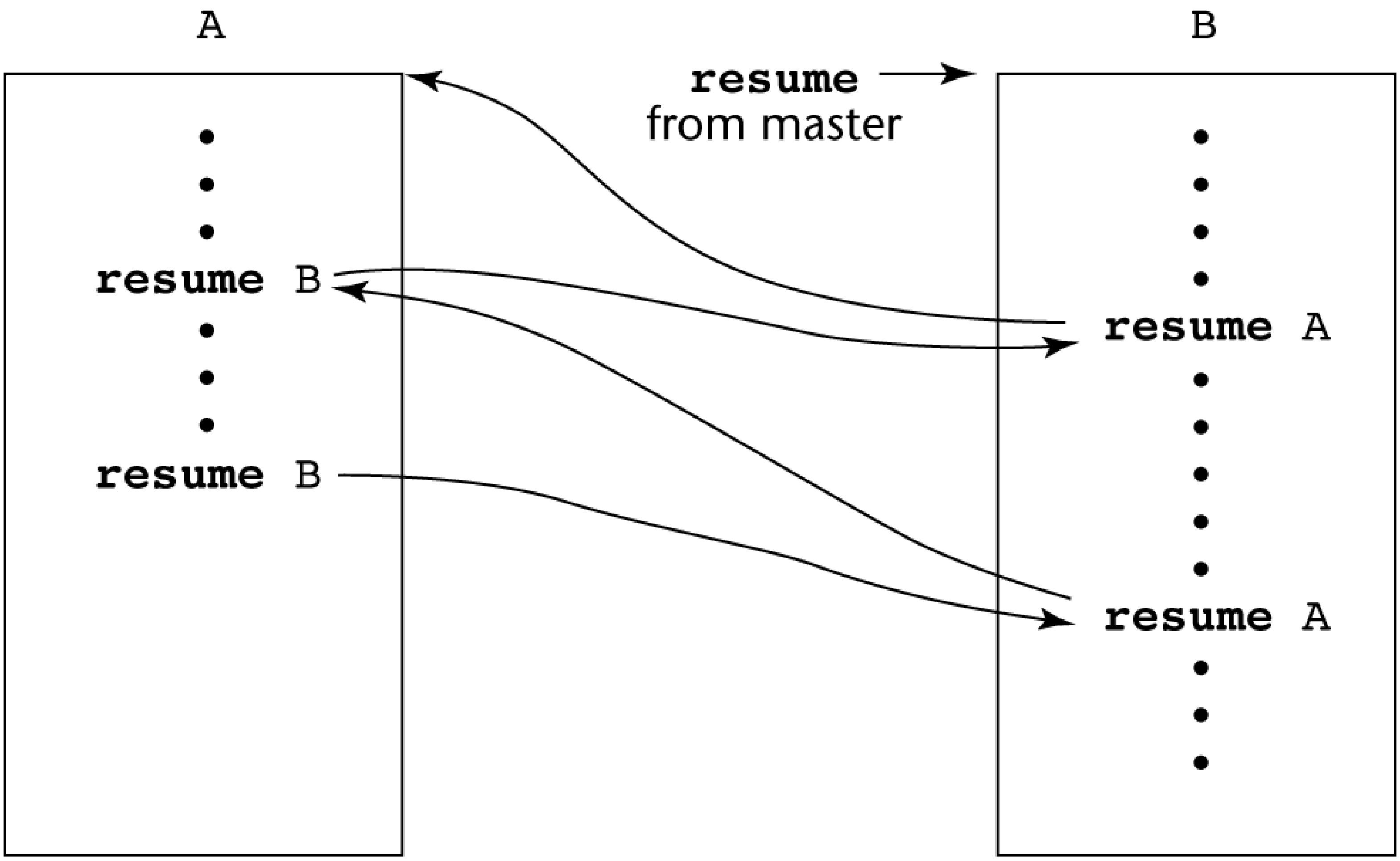
- A *coroutine* is a special kind of a subprogram that has multiple entries and controls them itself
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- *Coroutines call is a resume*. The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped
- Coroutines are history sensitive, thus they have static variables.
- Couroutines are created in an application by a special unit called master unit, which is not a coroutine.
- A coroutine may have an initialization code that is executed only when it is created.
- Only one coroutine executes at a given time.

Coroutines Illustrated: Possible Execution Controls



(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops