Sudan University of Science and Technology

Concepts of programing Languages

Subprograms Design -II

Dr. Aghabi Nabil Abosaif

Topics

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Design Issues for Functions
- Overloaded Subprograms
- Closures
- Coroutines

Implementing Parameter-Passing Methods

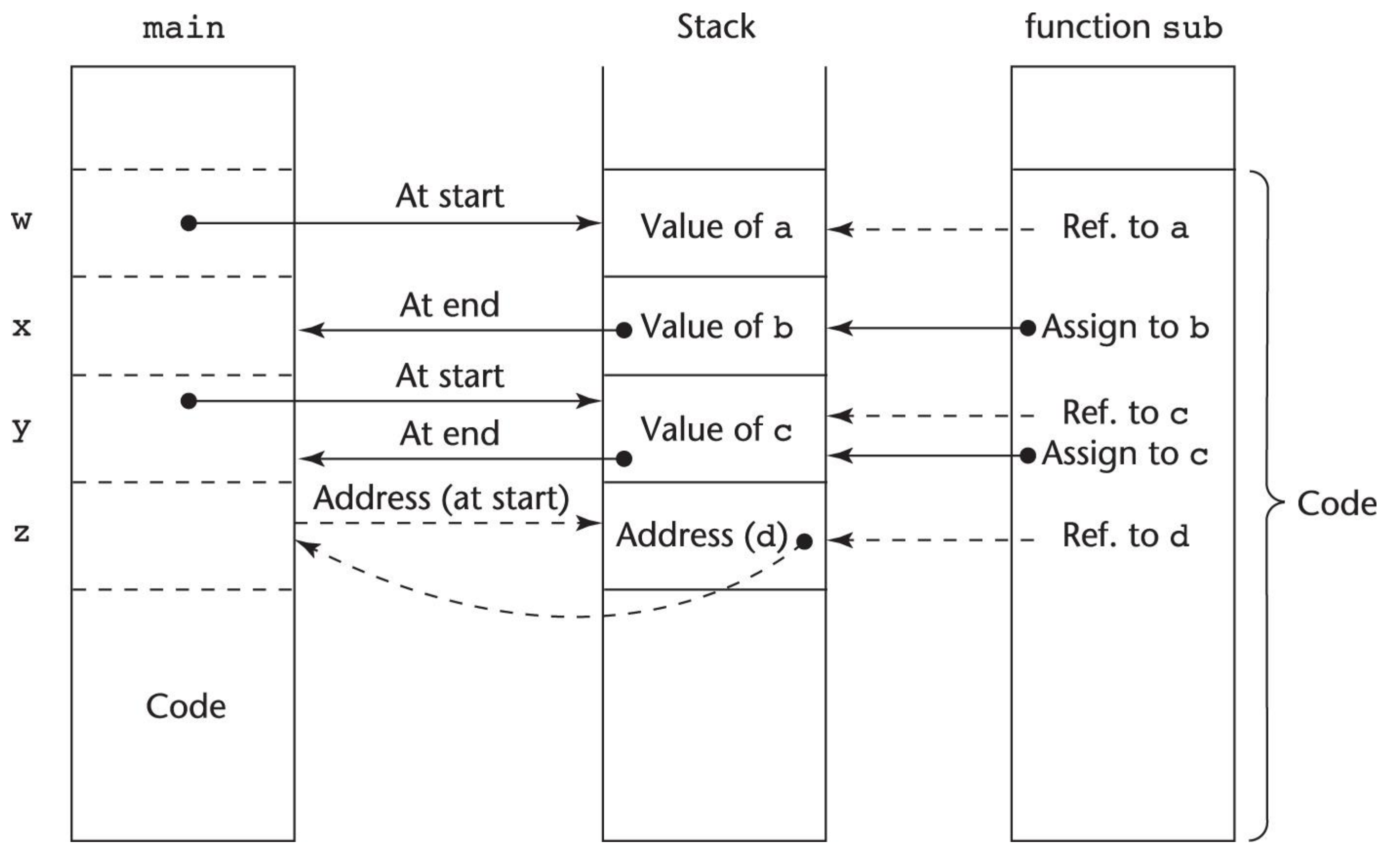
- In most languages **parameter communication** takes place thru **the run-time stack.**
- The four common **parameter-passing** methods:

1. **Pass-by-value:** The actual parameter's value is copied to the runtime stack, where it is stored as the formal parameter's value.
2. **Pass-by-result:** The parameter’s value is not copied immediately; instead, the result is stored on the stack, so that the calling program can retrieve it once the subprogram finishes.

Implementing Parameter-Passing Methods

3. **Pass-by-value-result**: This method combines pass-by-value and pass-by-result. The stack location is initialized by the calling program, used within the subprogram, and then the result is transferred back after execution.
4. **Pass-by-reference:** This method places the address (reference) of the actual parameter on the stack. The subprogram uses this reference to access and possibly modify the actual parameter.

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

Type Checking Parameters Need for Type Checking:

- Type checking ensures that the **types** of actual parameters **match** the **types** of formal parameters.
- This helps catch errors that could arise from mismatched types, preventing bugs that may be difficult to detect later.
- Considered very **important for reliability.**

Type Checking Parameters

- **FORTRAN 77 and original C**:
  - none
  - leading to potential bugs (e.g., passing an int to a function expecting a double).

```
double sin(x)
  double x;
  { . . . }
```

Using this form avoids type checking, thereby allowing calls such as

```
double value;
int count;
. . .
value = sin(count);
```

to be legal, although they are never correct.

Type Checking Parameters

- **C99 and C++:** Prototypes method

```
double sin(double x)
 { . . . }
```

  - The function's parameter types were specified, enabling type checking.
  - If there was a mismatch, the compiler would attempt to **force types** (e.g., converting an **int** to a **double** if needed(it is a widening coercion)).
  - If coercion wasn't possible or the number of parameters didn't match, **an error would occur**.

  - However, type checking can be avoided for some of the parameters by replacing the last part of the parameter list **with an ellipsis**, as in

```
int printf(const char* format_string, . . .);
```

Type Checking Parameters

- **Pascal and Java**: it is always required
- **C#**: Coercion and Reference Passing
  - if a **float** is passed to a **double** formal parameter, the value is automatically coerced (converted) from float to double if passed by value.
  - However, if passed by reference, type coercion isn't allowed. The actual and formal parameter types must match exactly to avoid issues like overflow when the value is returned.
- Relatively new languages **Perl, JavaScript, and PHP** do not require type checking
- In **Python and Ruby**, variables do not have types (objects do), so parameter type checking is not possible

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?
  1. What is the correct referencing environment for a subprogram that was sent as a parameter?

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

Referencing Environment-Example

```
function sub1() {
  var x;
  function sub2() {
    alert(x);  // Creates a dialog box with the value of x
    };
  function  sub3() {
    var  x;
    x = 3;
    sub4(sub2);
    };
  function  sub4(subx) {
    var  x;
    x = 4;
    subx();
    };
  x = 1;
  sub3();
  };
```

Referencing Environment- Example

- Consider the execution of sub2 when it is called in sub4.
- For ***shallow binding***, the referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4, and the output of the program is 4.
- For ***deep binding***, the referencing environment of sub2's execution is that of sub1, so the reference to x in sub2 is bound to the local x in sub1, and the output is 1.
- For ***ad hoc binding***, the binding is to the local x in sub3, and the output is 3.

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 know until execution** (e.g., event handling and GUIs)
- In C and C++, such calls are made through **function pointers.**

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 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**
  - Lua allows functions to return **multiple values**

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 <u>unique protocol (actual parameters passed in the function call and sometimes the return type.</u>

- C++, Java, C#, and Ada include predefined **overloaded subprograms**
  - For example, these languages have overloaded constructors **(constructors with different parameter types or numbers of parameters).**

Overloaded Subprograms
The Problem of Coercion

- Parameter coercion (automatic conversion of one type to another) can **complicate overloaded function calls**.

- When **no exact match** is found between the actual parameters and the parameter profile, the compiler may attempt to find the best match using coercions.

- The language designer must decide how to rank these coercions, which can be complex.
  - For instance, C++ has detailed rules for resolving such ambiguities.

Overloaded Subprograms

- In Ada, the **return type** of an overloaded function can be used to disambiguate calls.
- In **C++**, **Java**, and **C#**, the **return type** does not help the compiler decide which overloaded version to call, (e.g., one returns int and the other returns float), the call will result in a **compilation error** because the compiler cannot determine which version to use based on the return type alone.
- Overloaded subprograms **with default parameters** can also lead to **ambiguities**. For instance, in the C++ example:

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

Closures

- A closure is a subprogram (like a function or method) along with the referencing environment where it was defined.

- The referencing environment includes all variables that are **accessible** at the time the subprogram was created.

- The referencing environment **is needed if** the subprogram **can be called from any arbitrary** place in the program.

Closures

- A static-scoped language that does not permit nested subprograms doesn't need closures.

- Closures are only needed if a subprogram **can access variables in nesting scopes** and it can **be called from anywhere.**

- To support closures, an implementation may need to provide **unlimited extent to some variables** (because a subprogram may access a nonlocal variable that is normally no longer alive(deallocated))
  - Such variables are typically heap dynamic (allocated in the heap rather than the stack).

- Functional programming languages, scripting languages, and some imperative languages like **C#** support closures.

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`

The closure keeps a reference to the variable **x** even after the **makeAdder** function completes, and the lifetime of **x** must **extend for as long as the closure is in use.**

Coroutines

- A *coroutine* is a subprogram that has **multiple entries and controls them itself** – supported directly in Lua
- 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

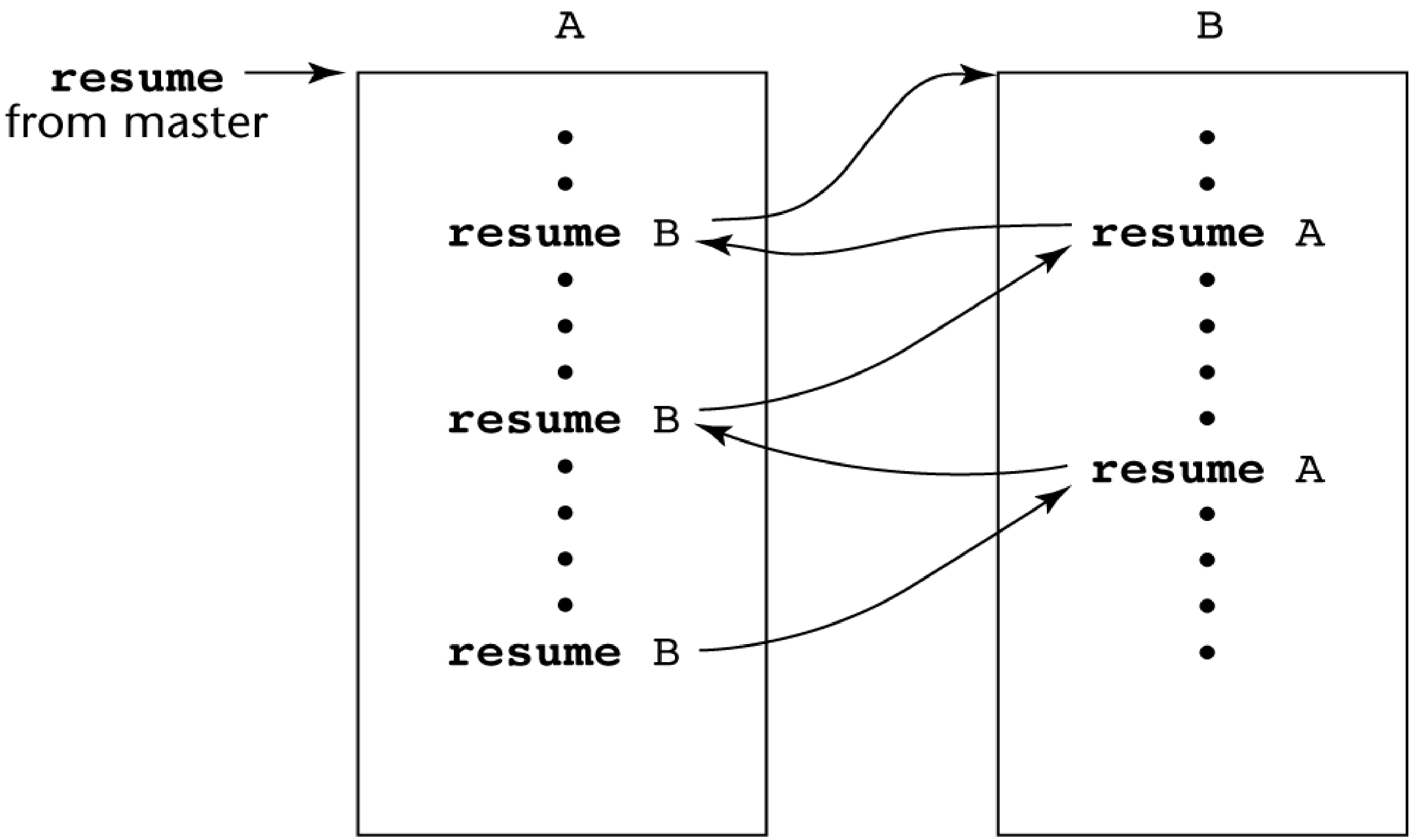
Coroutines

- Coroutines provide ***quasi-concurrent execution of*** **program units (the coroutines)**; their execution is interleaved, but not overlapped.
- **Quasi-Concurrency:** Coroutines share a single processor in a manner similar to how multiprogramming works, where multiple programs appear to be running concurrently even if only one is executing at a time.
- This is called quasi-concurrency, where the coroutines take turns running.

Example of - Coroutines

- **Card Game Simulation**: A card game with multiple players can be simulated using coroutines.
- A master program creates four player coroutines, each with their own hand of cards.
- The master program resumes each coroutine to simulate the players' turns.
- After a player finishes their turn, the control is passed to the next player’s coroutine, and so on until the game ends.

Coroutines Illustrated: Possible Execution Controls A starts B

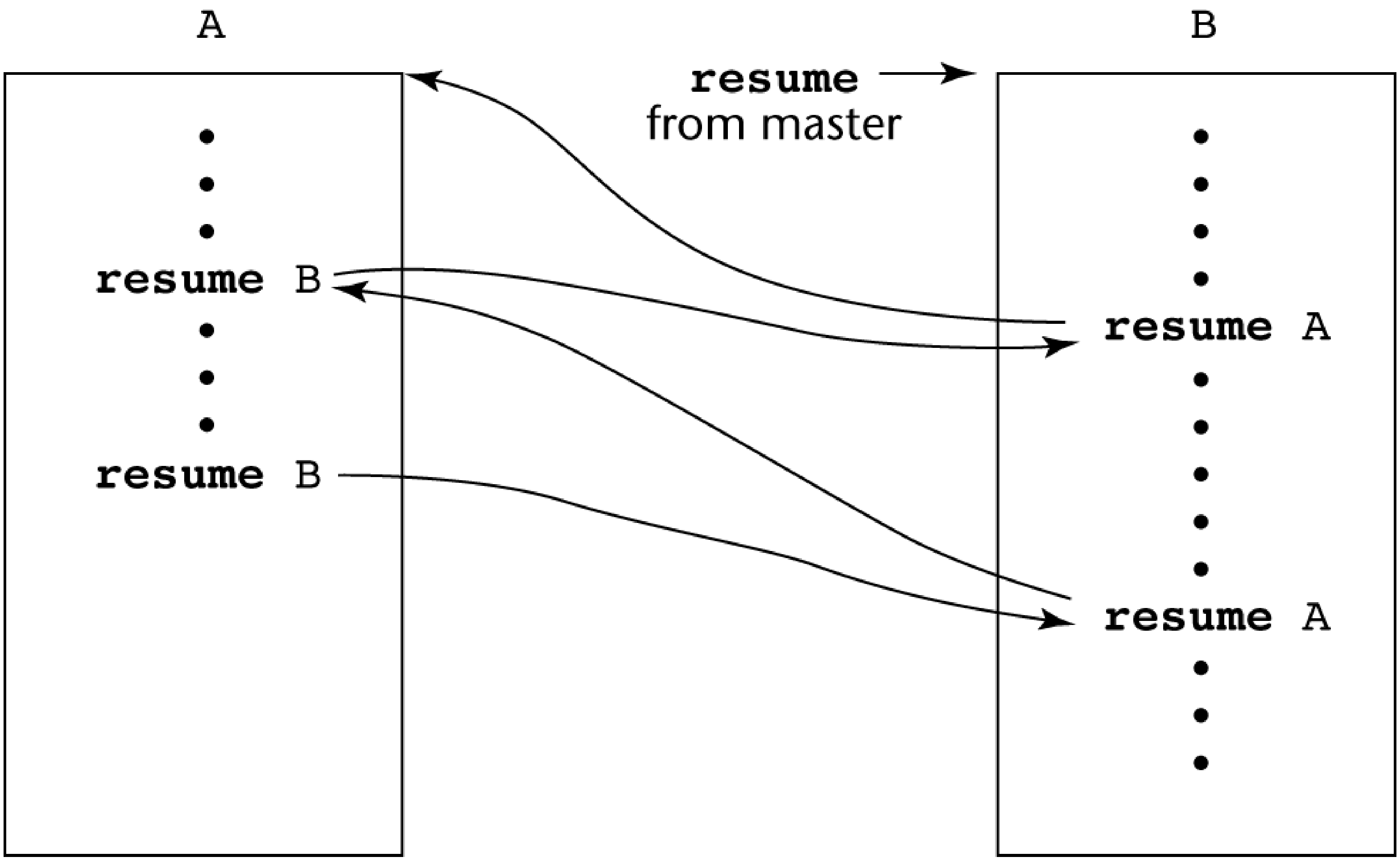


(a)

Coroutines Illustrated: Possible Execution Controls A starts B

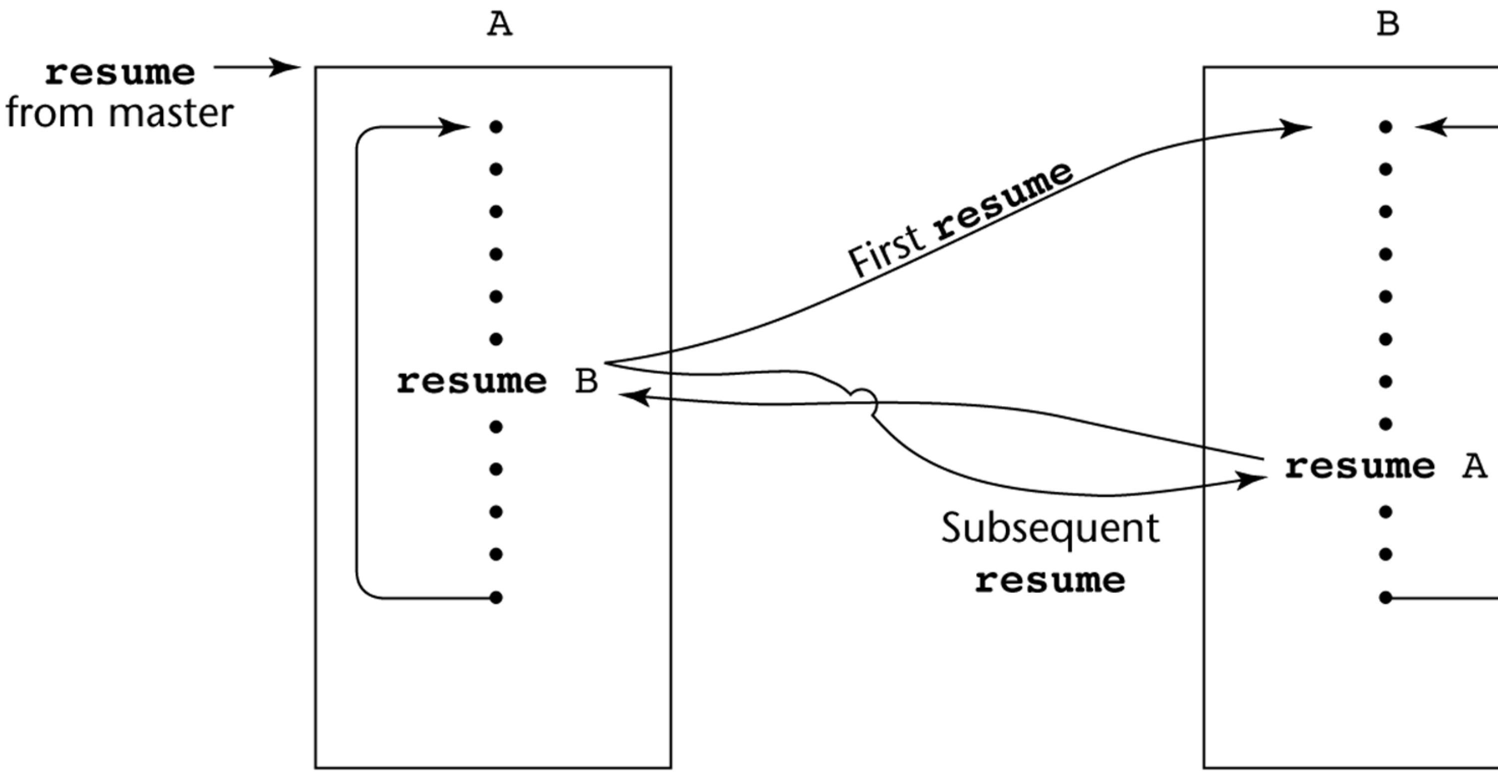
- The execution of coroutine A is started by the master unit.
- After some execution, A starts B.
- When coroutine B in first causes control to return to coroutine A, the semantics is that A continues from where it ended its last execution.
- In particular, its local variables have the values left them by the previous activation.

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**
- Local variables in subprograms can be **stack-dynamic or static**
- Three **models of parameter passing**: in mode, out mode, and inout mode
- Some languages allow operator **overloading**
- A **closure** is a subprogram and its ref. environment
- A **coroutine** is a special subprogram with multiple entries