COSC252: Programming Languages:

Basic Semantics: Functions

Jeremy Bolton, PhD
Asst Teaching Professor

GEORGETOWN UNIVERSITY

Topics

- 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
- Generic Subprograms
- User-Defined Overloaded Operators
- Closures
- Coroutines

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* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* (aka 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
  - Simple and effective
  - Ex:
    - Def: f(int x, int y)
    - Invocation: f(1, 1+4)
    - Binds x to 1 and y to 5, in scope associated with function f
- 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
  - Example, Invocation:
    - f(“x”, 4, “y” 1+4)

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)

  – In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)

- Variable numbers of parameters
  – C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**

Procedures and Functions

- There are two** categories of subprograms
  - *Procedures* are collection of statements
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
    - Map input to output
    - They are expected to produce no side effects
    - However, in practice, program functions have side effects

Design Issues for Subprograms

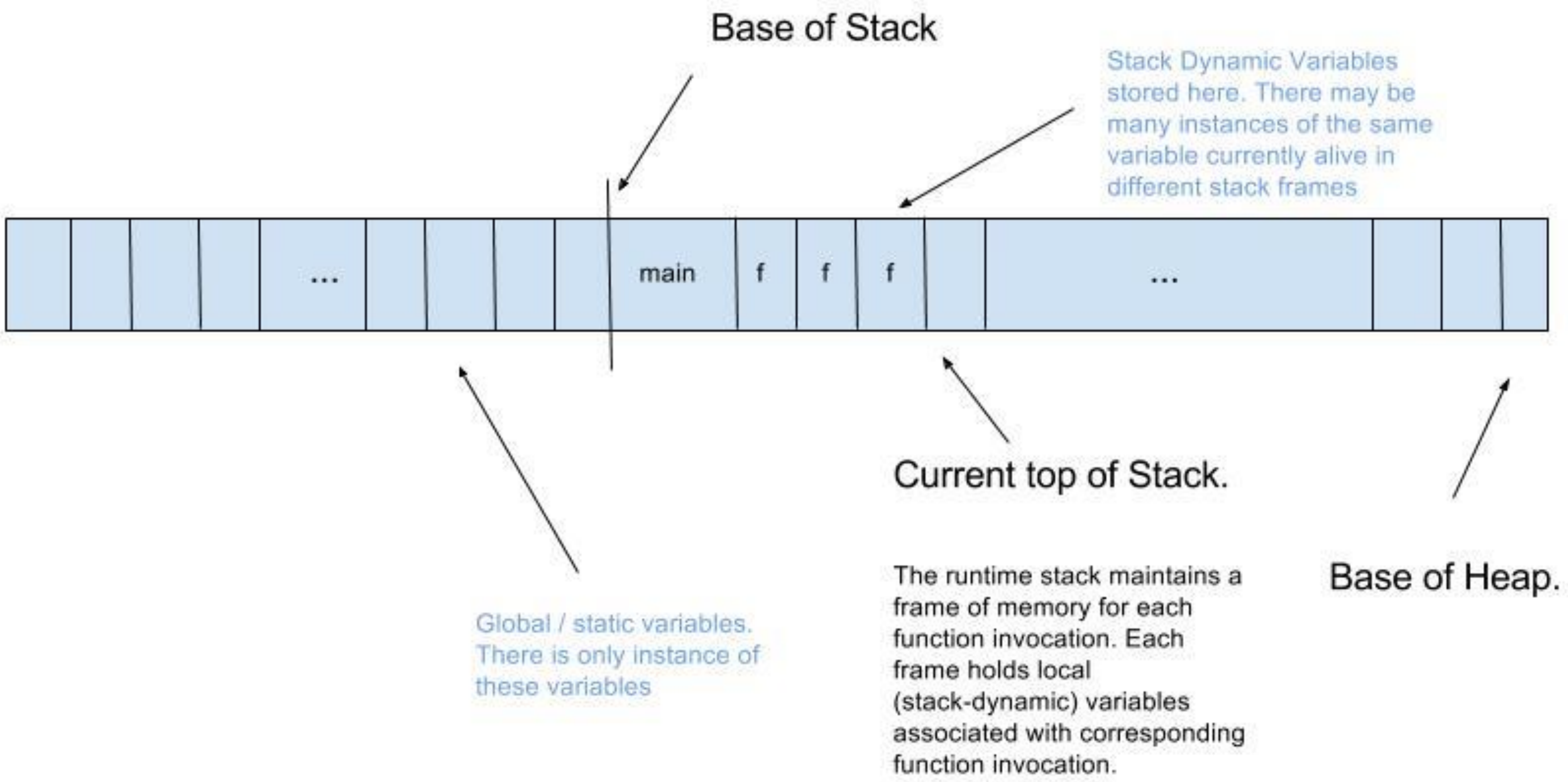
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types and how many values can be returned from functions?
- Can subprograms be overloaded?
  - How are overloaded functions resolved?
- Can subprogram be generic / template?

Local Referencing Environments

- Local variables can be stack-dynamic
  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages
    - Allocation/de-allocation, initialization time
    - Indirect addressing

- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables
  - Does not permit recursion, since there is only one instance of a local variable (not multiple instances as needed for recursion)

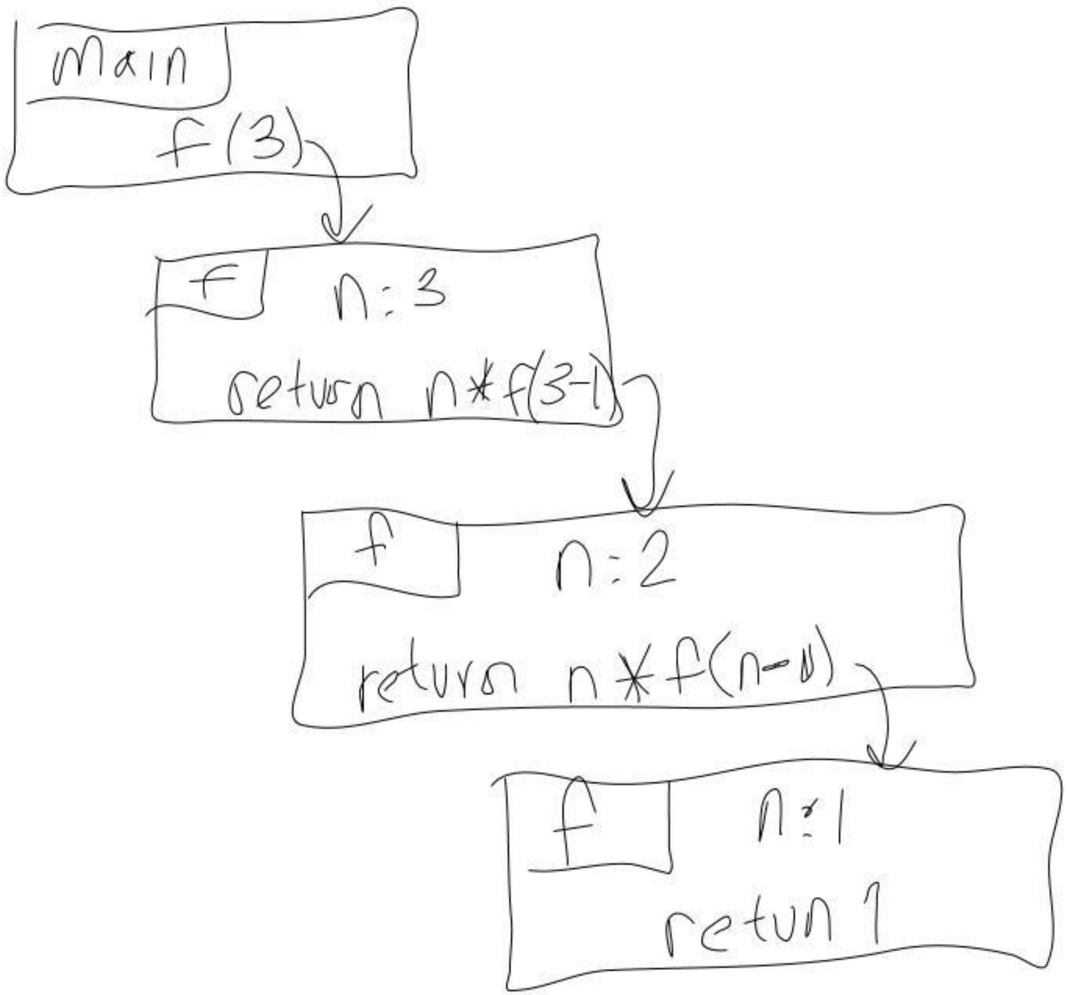
Stack Dynamic Allocation and Recursion



Draw Scope (function chain) diagrams to trace the execution of a recursive function.

```
int f(int n)
        {
        // Assumes n is non-negative
        int val = 1;
        if (n == 0 || n == 1) // Base case -- stop
repetition
                return 1;
        else  // recursive case -- continue
recursive call
        return n * f(n - 1);
        }

void main()
{f(3);}
```

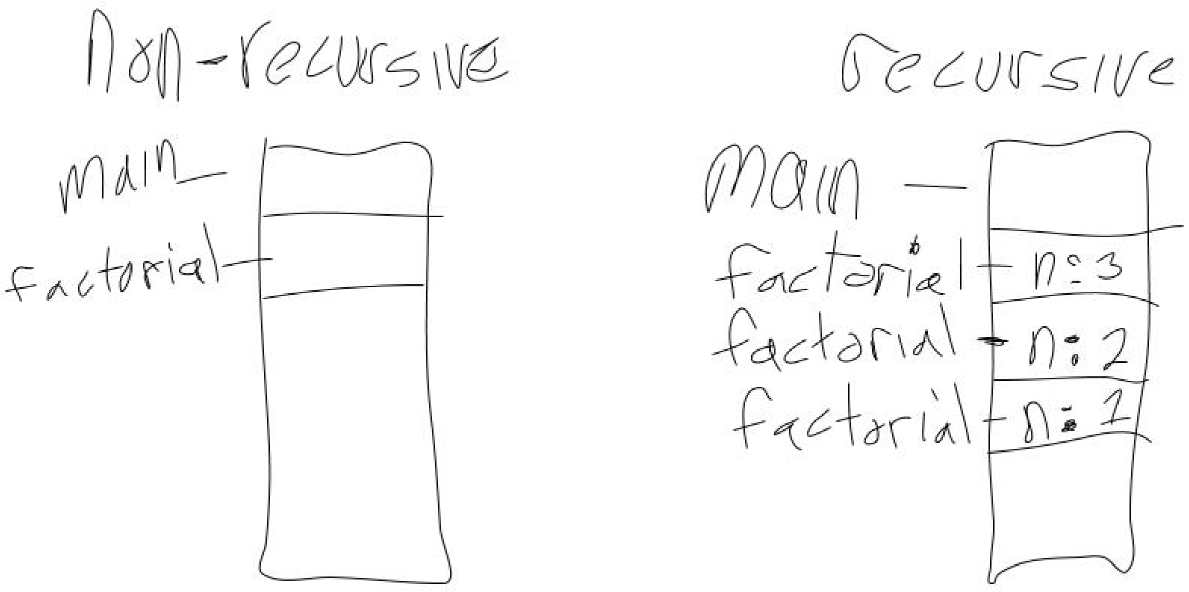


Recursion (and non-recursion) and the Stack

```
int factorial(int n)
    {
    //Assumes non-negative n
    int val = 1;
    for (int i = n; i > 1; i--;) //
repeatedly take product of values between 1
and n
        val = val * i;

    return val;
    }
```

```
int factorial(int n)
    {
    // Assumes n is non-negative
    int val = 1;
    if (n == 0 || n == 1) // Base case --
stop repetition
        return 1;
    else  // recursive case -- continue
recursive call
    return n * factorial(n - 1);
    }
```



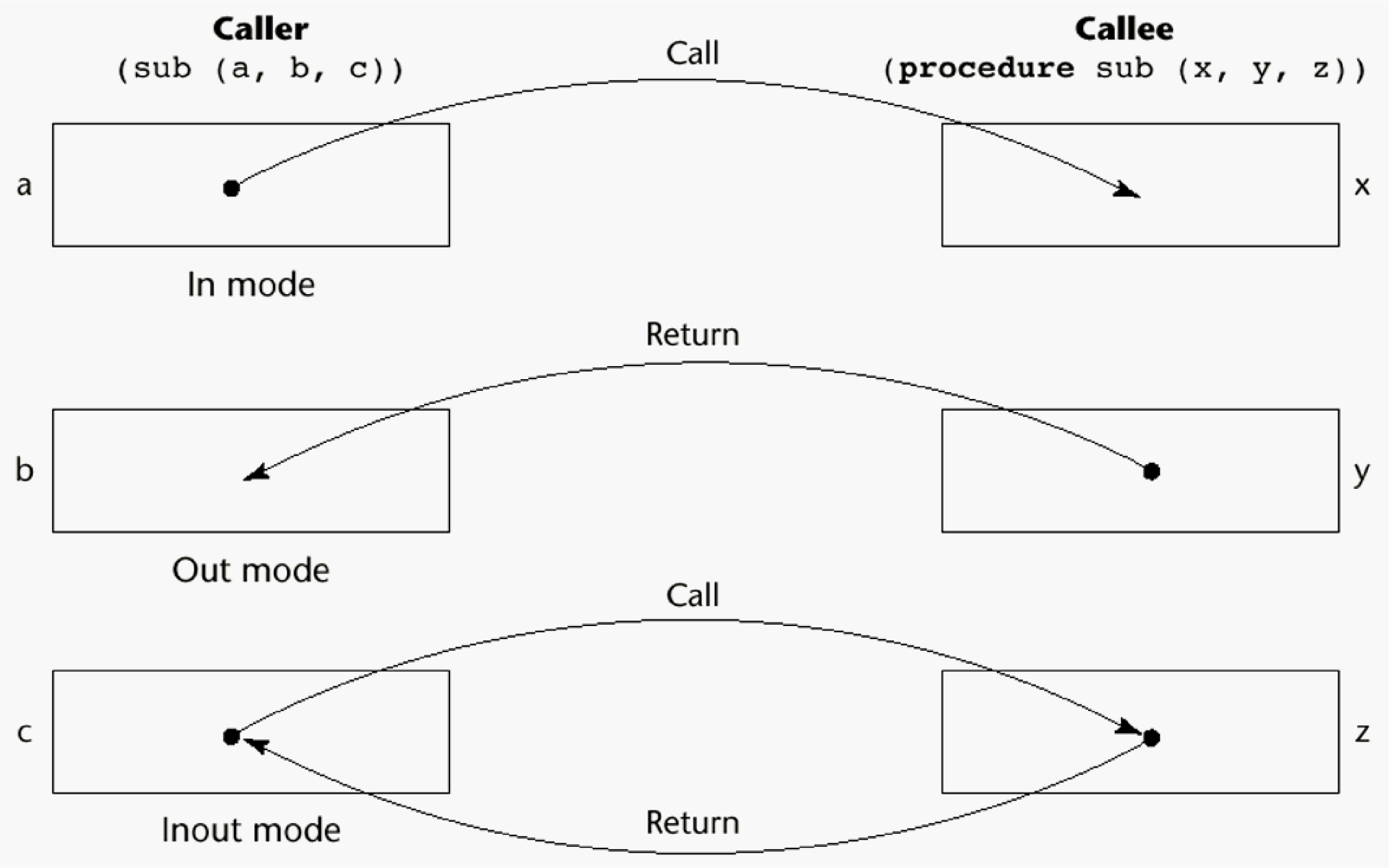
Local Referencing Environments: Examples

- In most contemporary languages, locals are stack dynamic
- In C, locals are by default stack dynamic, but can be declared **`static`**
- The methods of C++, Java, Python, and C# only have stack dynamic locals
- In Lua, all implicitly declared variables are global; local variables are declared with **`local`** and are stack dynamic

Semantic Models of Parameter Passing

- In mode (input)
- Out mode (output)
  - Simply a container for a return value
- Inout mode (input and output)
  - EG pass-by-reference

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
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move**): additional storage** is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram 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 move
  - Require extra storage location and copy operation
- Potential problems:
  - `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?

GEORGETOWN UNIVERSITY

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
  - Potentials for unwanted side effects
  - **Unwanted aliases**
  - `fun(total, total);  fun(list[i], list[j];  fun(list[i], i);`

GEORGETOWN UNIVERSITY

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
- Used in Algol

Pass by Name Example

Procedure Definition:

```
procedure swap (a, b);
integer a, b, temp;
begin
  temp := a;
  a := b;
  b:= temp
end;
```

Execute: swap(x, y):

```
  temp := x;
  x := y;
  y := temp
```

Execute: swap(i, x[i]):

```
  temp := i;
  i := x[i];
  x[i] := temp
```

Textually x and y do not dependent, so the late binding does not affect the expected result

Note that the i and x[i] parameters are ***not*** evaluated before function execution. Instead they are textually substituted into the function definition and replace a and b correspondingly. The parameters are finally evaluated when referenced or assigned.

i: 2

x:								
	4	5	6	7	8	9	1	2

i: 6

x:								
	1	2	3	4	5	2	7	8

GEORGETOWN UNIVERSITY

Implementing Parameter-Passing Methods

- In most languages parameter communication takes place thru the run-time stack

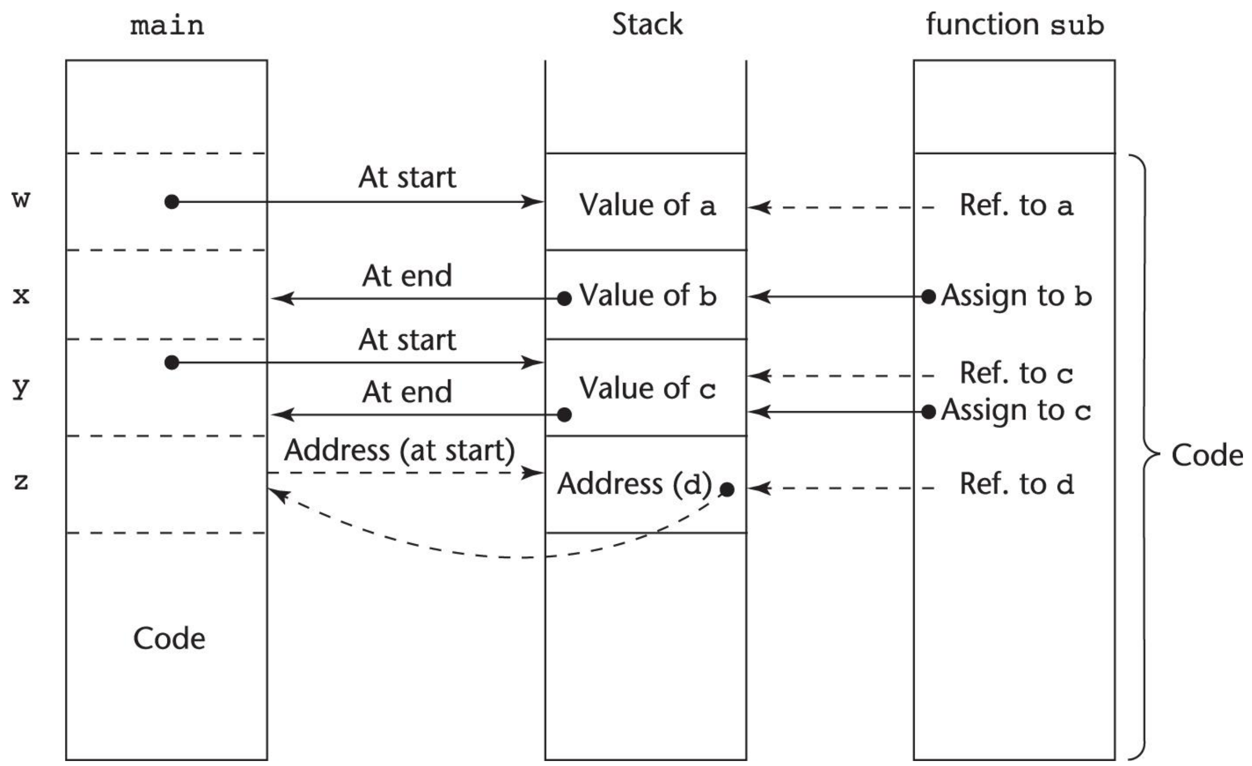
- Pass-by-reference are the simplest to implement; only an address is placed in the stack

Implementing 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 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 primitive parameters are passed are passed by value
  - Object parameters are passed by reference

Parameter Passing Methods of Major Languages (continued)

- Fortran 95+
  - Parameters can be declared to be in, out, or inout mode
- C#
  - Default method: pass-by-value
    - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with `ref`
- PHP: very similar to C#, except that either the actual or the formal parameter can specify ref
- Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal

GEORGETOWN UNIVERSITY

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: it is always required
- ANSI C and C++: choice is made by the user
- Some 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

Multidimensional Arrays as Parameters: C++

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function
  - Programmer is required to include the declared sizes of all but the first subscript in the actual parameter

- In General, stack allocation concerns*

- Remedies
  - Arrays are by default pass by reference in C++*
  - Use/pass pointer and pass dimensions as arg

Multidimensional Arrays as Parameters: 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

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

- *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

Bindings

```
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();    // calling the passed subprogram which is a parameter.
     };
     x = 1;
     sub3();
  };
```

Javascript Example from text:

- sub1 calls sub3 which calls sub4 by the call statement, sub4(sub2).
- sub4() subsequently calls sub2().
- The environment of the execution of sub2() in this case can be one of the following three:
  1. that of sub4(): Shallow Binding.
     x=4
  2. that of sub1(): Deep Binding.
     x=1
  3. that of sub3(): Ad Hoc Binding.
     x=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

C++ Example: functions as arguments

```cpp
// Note our user-defined comparison is the third parameter
void selectionSort(int *array, int size, bool (*comparisonFcn)(int, int))
{
    // Step through each element of the array
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // bestIndex is the index of the smallest/largest element we've encountered so far.
        int bestIndex = startIndex;
        // Look for smallest/largest element remaining in the array (starting at startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // If the current element is smaller/larger than our previously found smallest
            if (comparisonFcn(array[bestIndex], array[currentIndex])) // COMPARISON DONE HERE
                // This is the new smallest/largest number for this iteration
                bestIndex = currentIndex;
        }
        // Swap our start element with our smallest/largest element
        std::swap(array[startIndex], array[bestIndex]);
    }
}
```

```cpp
bool ascending(int x, int y)
{
    return x > y; // swap if the first element is greater than the second
}

// Here is a comparison function that sorts in descending order
bool descending(int x, int y)
{
    return x < y; // swap if the second element is greater than the first
}

// This function prints out the values in the array
void printArray(int *array, int size)
{
    for (int index=0; index < size; ++index)
        std::cout << array[index] << " ";
    std::cout << '\n';
}

int main()
{
    int array[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };

    // Sort the array in descending order using the descending() function
    selectionSort(array, 9, descending);
    printArray(array, 9);

    // Sort the array in ascending order using the ascending() function
    selectionSort(array, 9, ascending);
    printArray(array, 9);

    return 0;
}
```

Example taken from learncpp

GEORGETOWN UNIVERSITY

Design Issues for Functions

- Are side effects allowed?
  - Should parameters 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 unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- 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. Three categories

1. Overloaded subprograms provide *ad hoc polymorphism*

2. *Subtype polymorphism* means that a variable of type T can access any object of type T or any type derived from T (OOP languages)

3. 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*
 - A cheap compile-time substitute for dynamic binding

Generic Subprograms (continued)

- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - 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;
  }
```

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)`

GEORGETOWN UNIVERSITY

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
  - 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)

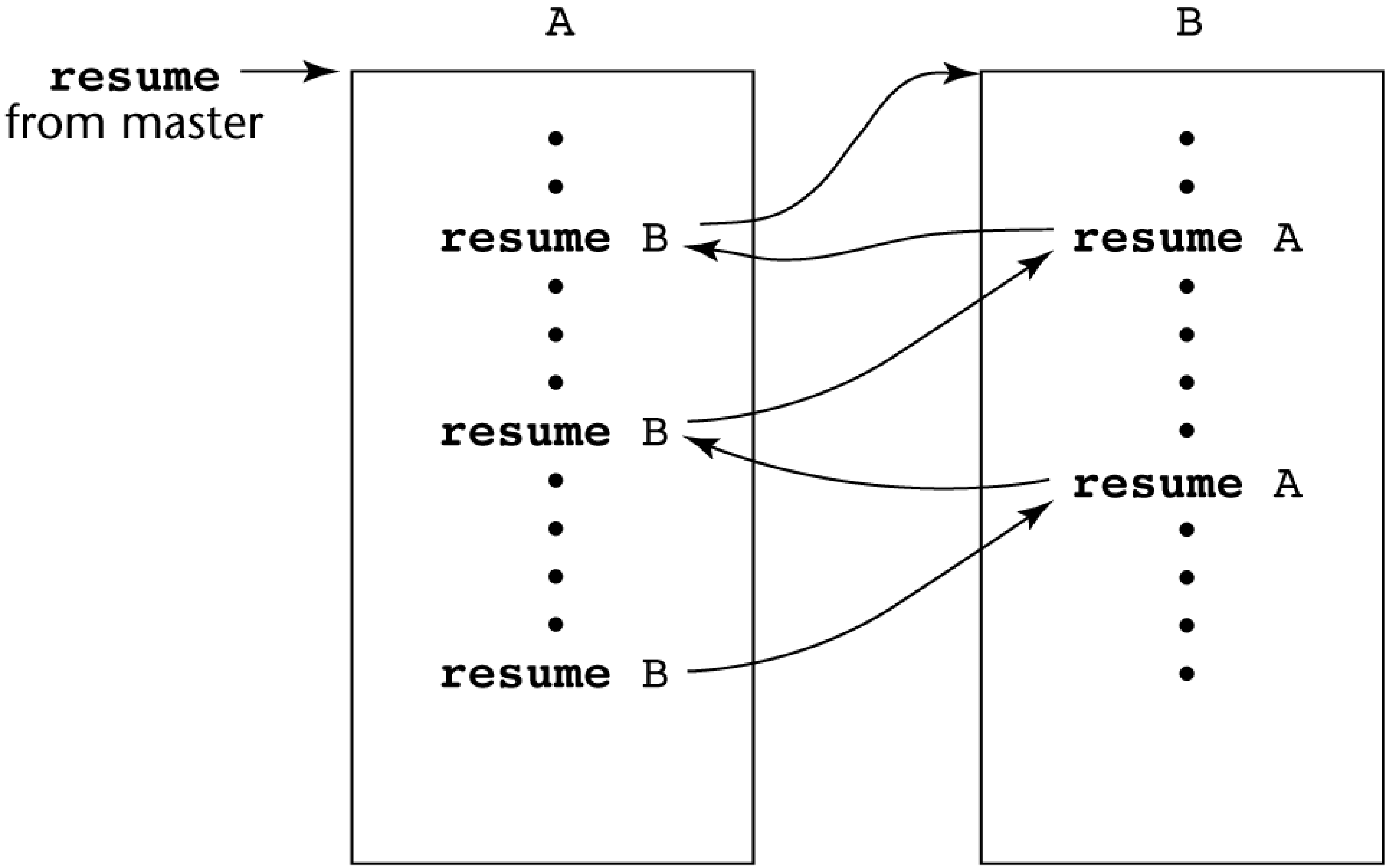
Summary

- 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
- Subprograms can be generic
- A closure is a subprogram and its ref. environment
- A coroutine is a special subprogram with multiple entries

Coroutines

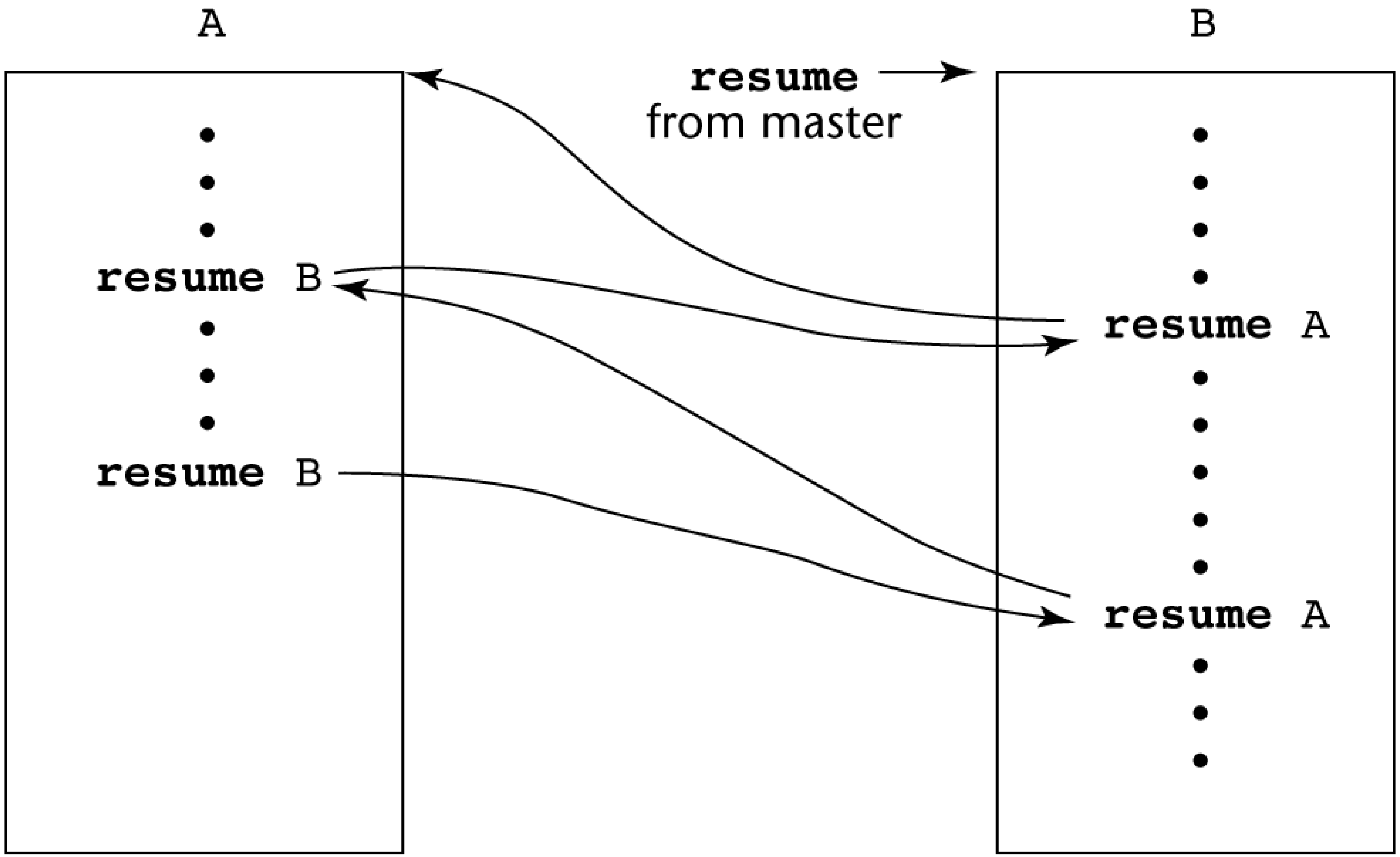
- Only Lua fully supports co-routines
- A *coroutine* is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a 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 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