UNIT -3
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

UNIT-3
Fundamentals of Subprograms

R.JEEVITHA
Asst. Professor, Department of CSE
Narsimha Reddy Engineering College (Autonomous)
Secunderabad, Telangana State, India-500100.
Ph. No: 09959845657

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

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* 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, 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

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`**
- In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.
- In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
- In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods

Ruby Blocks

- Ruby includes a number of iterator functions, which are often used to process the elements of arrays
- Iterators are implemented with blocks, which can also be defined by applications
- Blocks are attached methods calls; they can have parameters (in vertical bars); they are executed when the method executes a **`yield`** statement

```
def fibonacci(last)
   first, second = 1, 1
   while first <= last
     yield first
     first, second = second, first + second
   end
end

puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "} puts
```

Procedures and Functions

There are two categories of subprograms

- *Procedures* are collection of statements that define parameterized computations
- *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
  - They are expected to produce no side effects
  - In practice, program functions have side effects

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?

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
  - Subprograms cannot be history sensitive

Local variables can be static

- Advantages and disadvantages are the opposite of those for stack-dynamic local variables

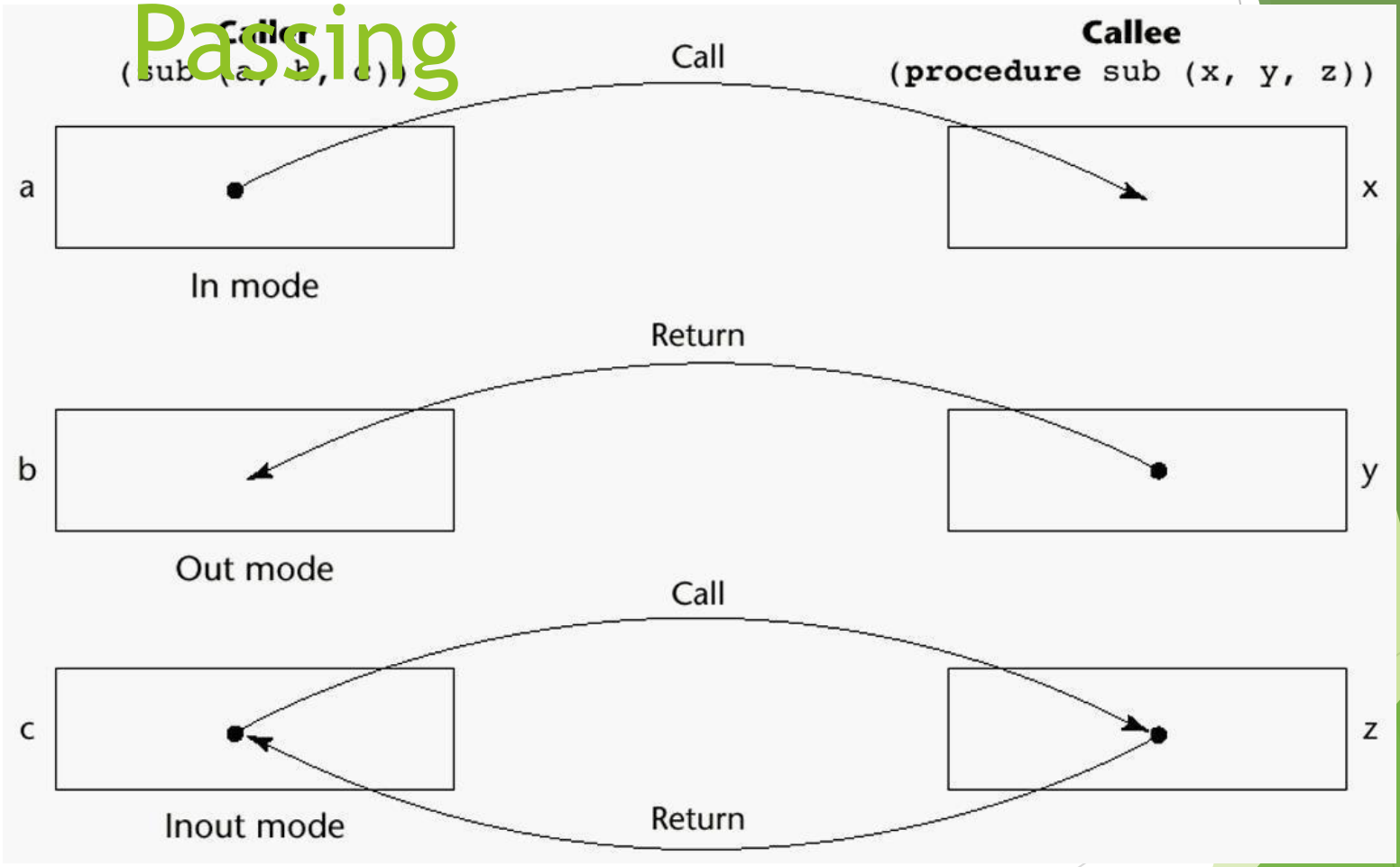
Semantic Models of Parameter Passing

In mode

Out mode

Inout mode

Models of Parameter Passing



Conceptual Models of Transfer

Physically move a path

Move an access path

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 problem: `sub(p1, p1);`
whichever formal parameter is copied back will represent the 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

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
- Potentials for unwanted side effects (collisions)
- Unwanted aliases (access broadened)

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

Implementing Parameter-Passing Methods

In most language 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

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

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

- 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

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#

Perl: all actual parameters are implicitly placed in a predefined array named `@_`

Python and Ruby use pass-by-assignment (all data values are objects)

Type Checking Parameters

- Considered very important for reliability FORTRAN 77 and original C: none
- Pascal, FORTRAN 90, Java, and Ada: it is always required ANSI C and C++: choice is made by the user
  - – Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking

Multidimensional Arrays as Parameters

- 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

Multidimensional Arrays as Parameters: C and C++

Programmer is required to include the declared sizes of all but the first subscript in the actual parameter

Disallows writing flexible subprograms

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

Multidimensional Arrays as Parameters: Ada

Ada – not a problem

- Constrained arrays – size is part of the array's type
- Unconstrained arrays - declared size is part of the object declaration

Multidimensional Arrays as Parameters: Fortran

Formal parameter that are arrays have a declaration after the header

- For single-dimension arrays, the subscript is irrelevant
- For multidimensional arrays, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function

Multidimensional Arrays as Parameters: Java and C#

Similar to Ada

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:

Are parameter types checked?

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

Parameters that are Subprogram Names: Parameter Type Checking

C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked

FORTRAN 95 type checks

Ada does not allow subprogram parameters; an alternative is provided via Ada's generic facility

Java does not allow method names to be passed 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

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

Overloaded subprograms provide *ad hoc polymorphism*

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)

Ada

- Versions of a generic subprogram are created by the compiler when explicitly instantiated by a declaration statement
- Generic subprograms are preceded by a `generic` clause that lists the generic variables, which can be types or other subprograms

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

Generic Subprograms (continued)

Java 5.0

- Differences between generics in Java 5.0 and those of C++ and Ada:

1. Generic parameters in Java 5.0 must be classes

Java 5.0 generic methods are instantiated just once as truly generic methods

3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters 4. Wildcard types of generic parameters

Generic Subprograms (continued)

C# 2005

Supports generic methods that are similar to those of Java 5.0

One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type

Examples of parametric polymorphism: C++

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

The above template can be instantiated for any type for which operator > is defined

```
int max (int first, int second) { return
  first > second? first : second;
}
```

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
- Lua allows functions to return multiple values

User-Defined Overloaded Operators

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

An Ada example

```
function "*" (A,B: in Vec_Type): return
  Integer is
  Sum: Integer := 0;
  begin
  for Index in A'range loop
   Sum := Sum + A(Index) * B(Index)
  end loop
  return sum;
end "*";
…
c = a * b; -- a, b, and c are of type Vec_Type
```

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

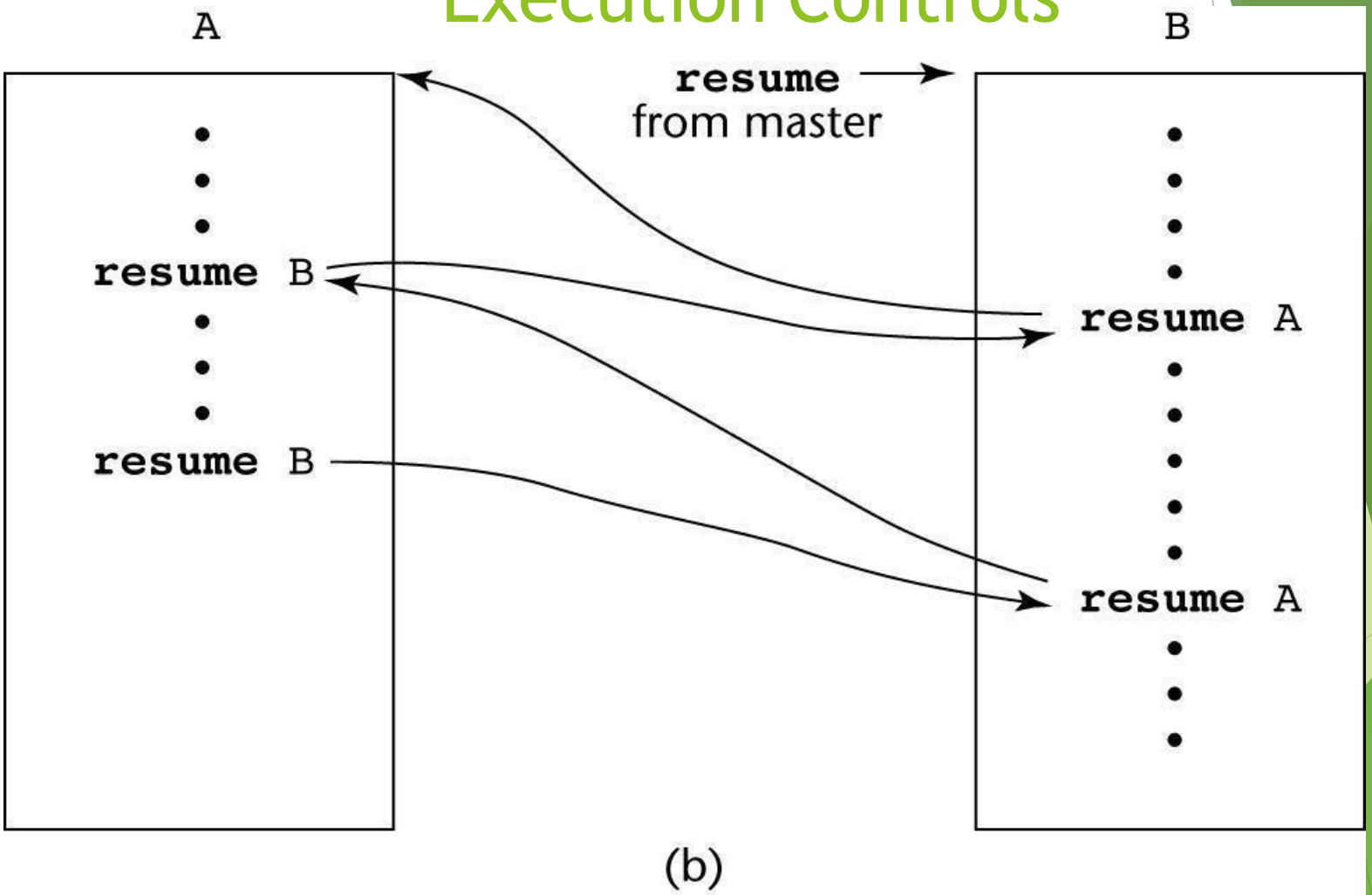
A coroutine call is named a *resume*

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

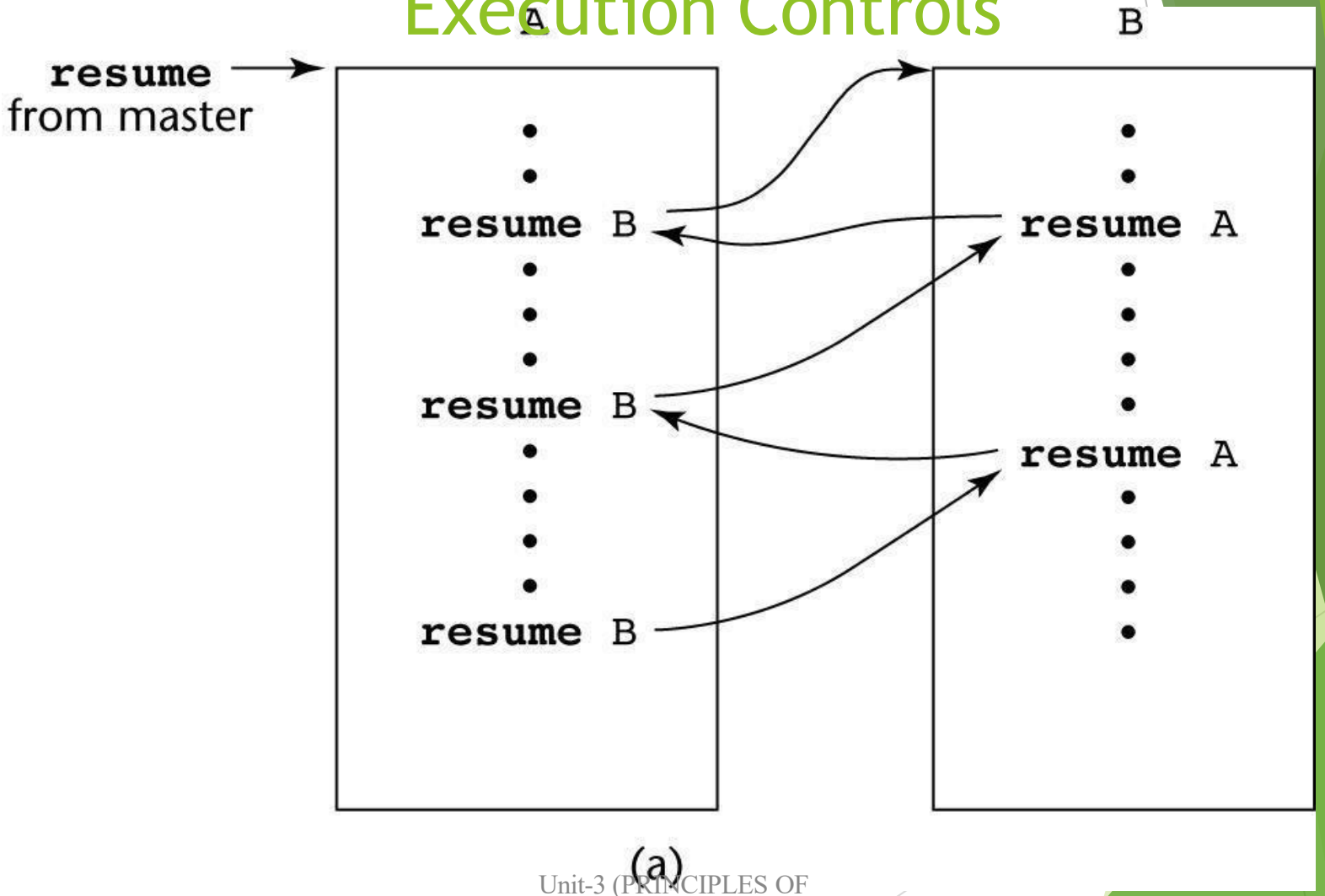
Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

Coroutines Illustrated: Possible Execution Controls



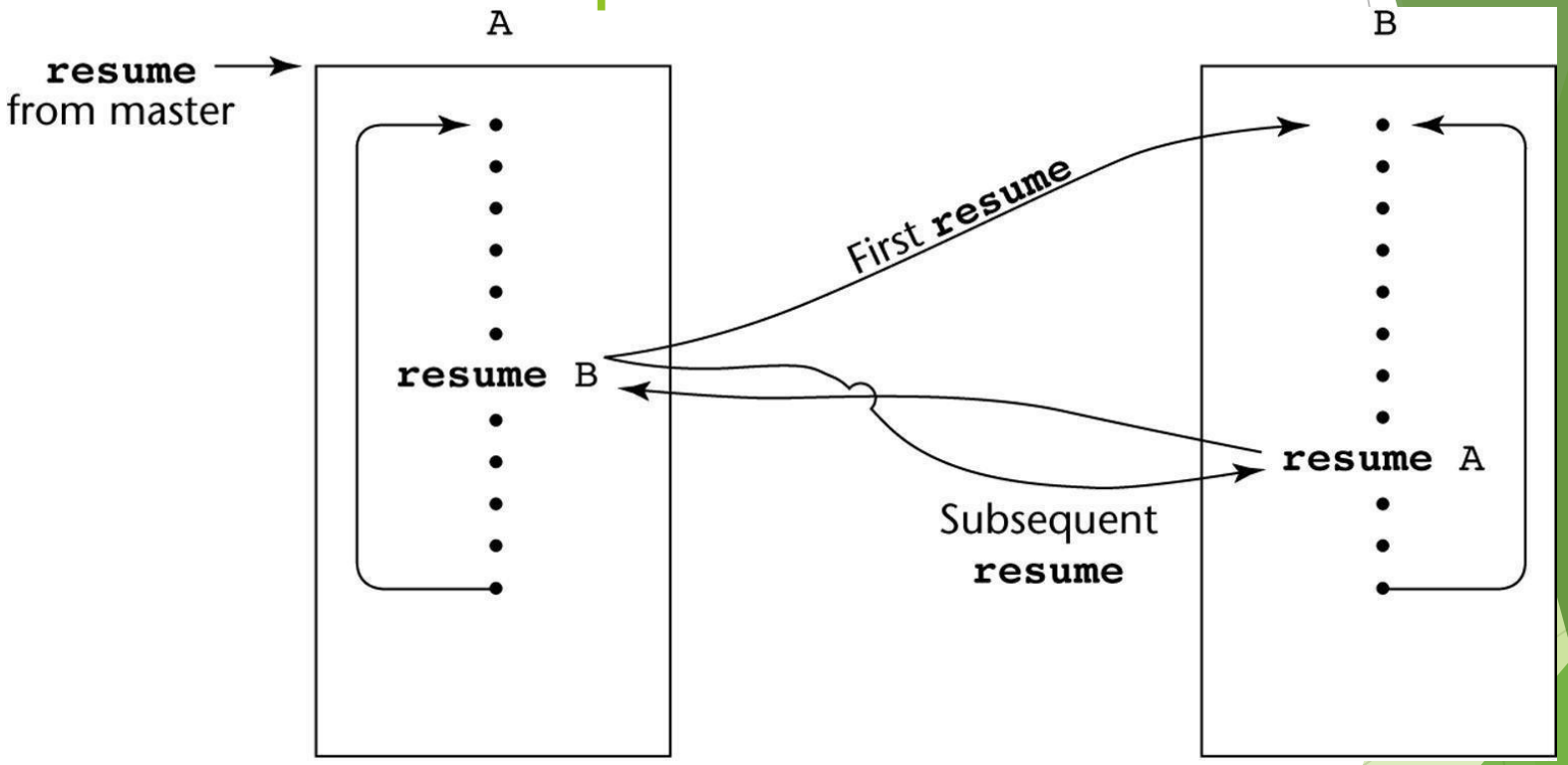
(b)

Coroutines Illustrated: Possible Execution Controls



(a)

Coroutines Illustrated: Possible Execution Controls with Loops



The General Semantics of Calls and Returns

The subprogram call and return operations of a language are together called its *subprogram linkage*

General semantics of subprogram calls

- Parameter passing methods
- Stack-dynamic allocation of local variables
- Save the execution status of calling program
- Transfer of control and arrange for the return
- If subprogram nesting is supported, access to nonlocal variables must be arranged

The General Semantics of Calls and Returns

General semantics of subprogram returns:

- In mode and inout mode parameters must have their values returned
- Deallocation of stack-dynamic locals
- Restore the execution status
- Return control to the caller

Implementing "Simple" Subprograms: Call Semantics

Call Semantics:

Save the execution status of the caller
Pass the parameters
Pass the return address to the callee
Transfer control to the callee

Implementing "Simple" Subprograms: Return Semantics

Return Semantics:

- If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
- If it is a function, move the functional value to a place the caller can get it
- Restore the execution status of the caller
- Transfer control back to the caller

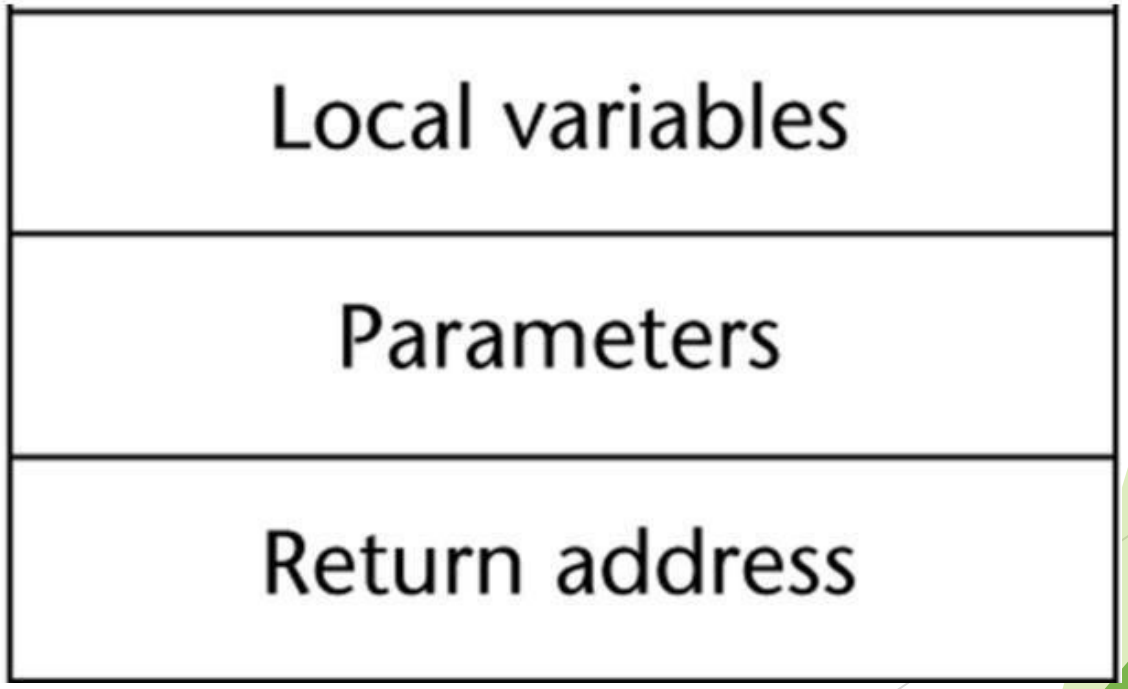
Required storage:

- Status information, parameters, return address, return value for functions

Implementing "Simple Subprograms: Parts

- Two separate parts: the actual code and the non- code part (local variables and data that can change)
- The format, or layout, of the non-code part of an executing subprogram is called an *activation record*
- An *activation record instance* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

An Activation Record for "Simple" Subprograms

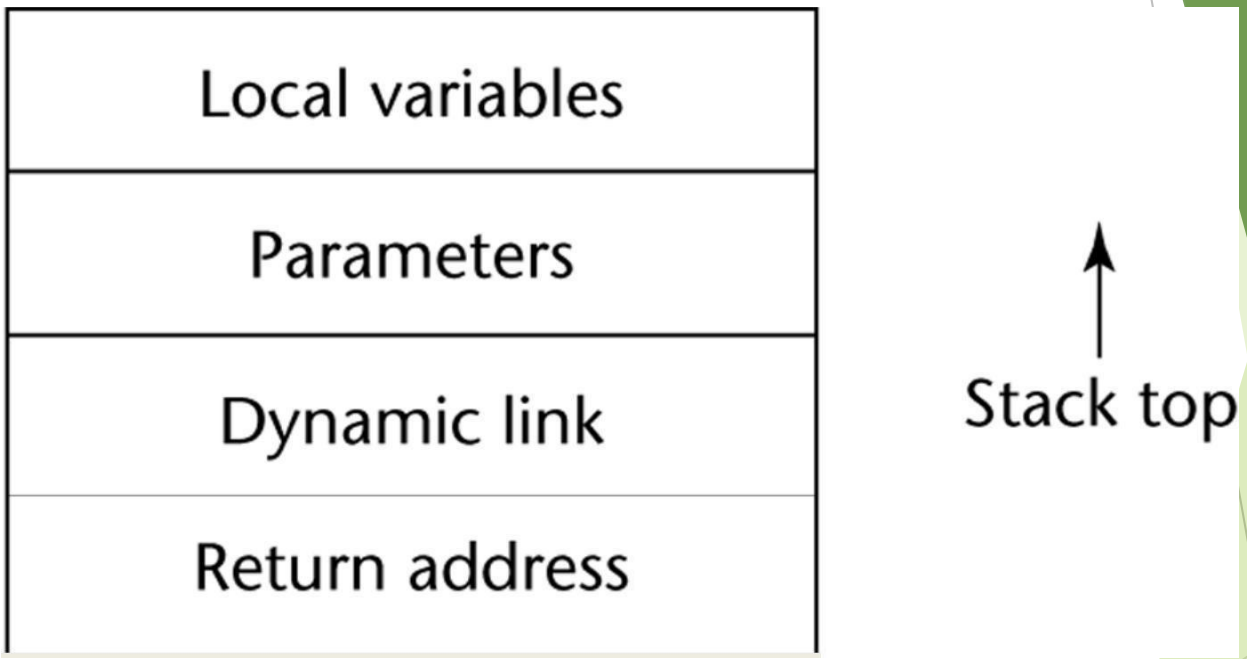


Implementing Subprograms with Stack-Dynamic Local Variables

More complex activation record

- The compiler must generate code to cause implicit allocation and deallocation of local variables
- Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

Typical Activation Record for a Language with Stack-Dynamic Local Variables



Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

The activation record format is static, but its size may be dynamic

The *dynamic link* points to the top of an instance of the activation record of the caller

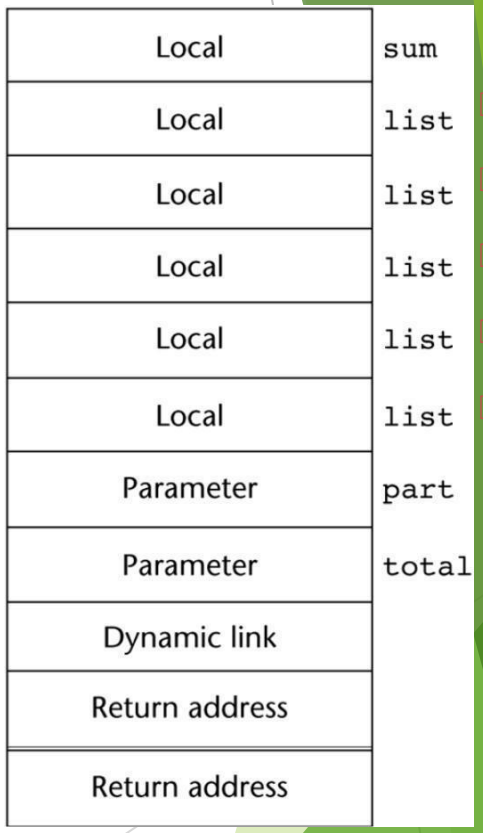
An activation record instance is dynamically created when a subprogram is called

Activation record instances reside on the run-time stack

The *Environment Pointer* (EP) must be maintained by the run-time system. It always points at the base of the activation record instance of the currently executing program unit

An Example: C Function

```
void sub(float total, int part)
{
   int list[5];
   float sum;
   …
}
```

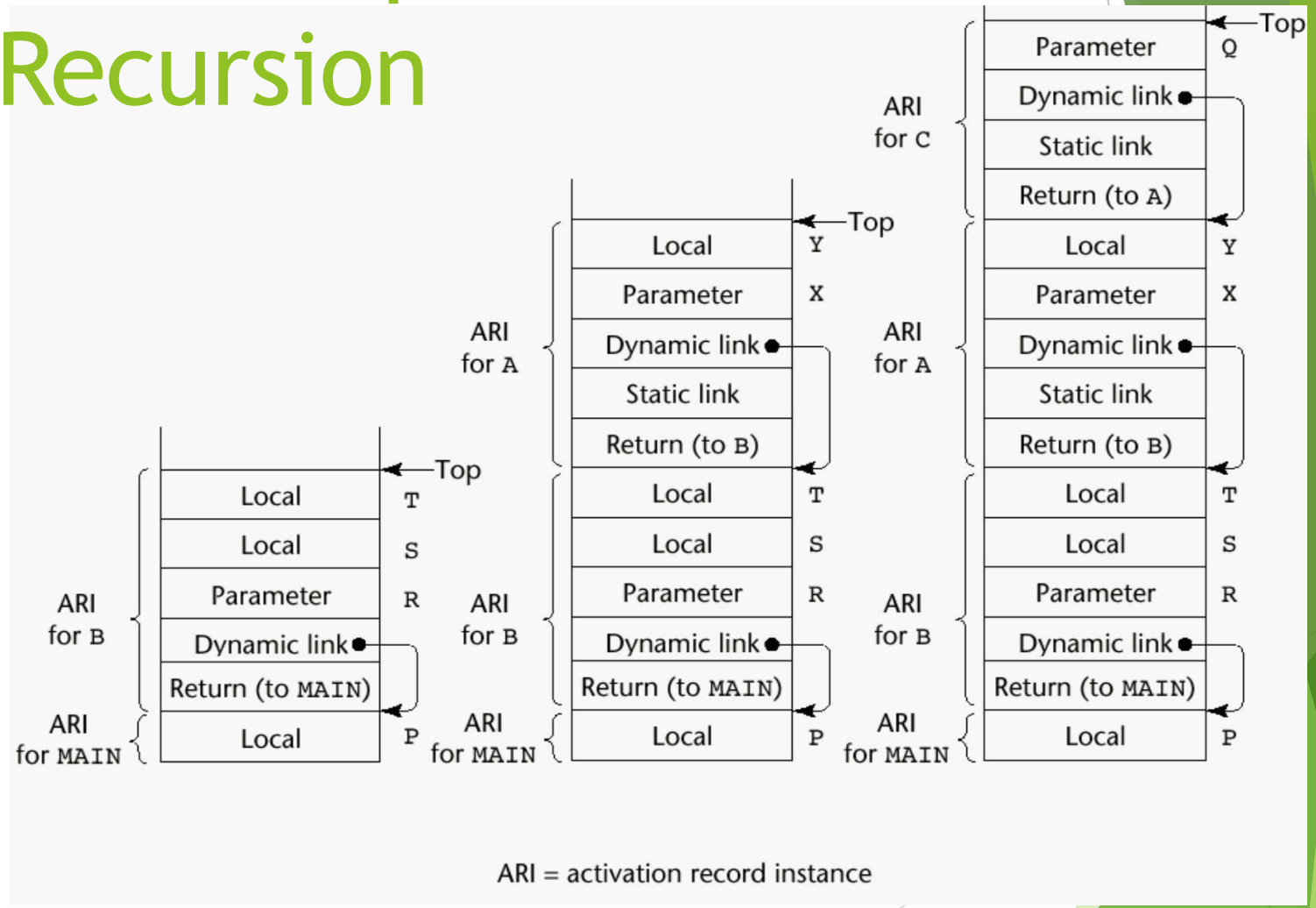


An Example Without Recursion

```
void A(int x) {
    int y;
    ...
    C(y);
    ...
}
void B(float r) {
    int s, t;
    ...
    A(s);
    ...
}
void C(int q) {
    ...
}
void main() {
    float p;
    ...
    B(p);
    ...
}
```

main calls B
B calls A
A calls C

An Example Without Recursion



Dynamic Chain and Local Offset

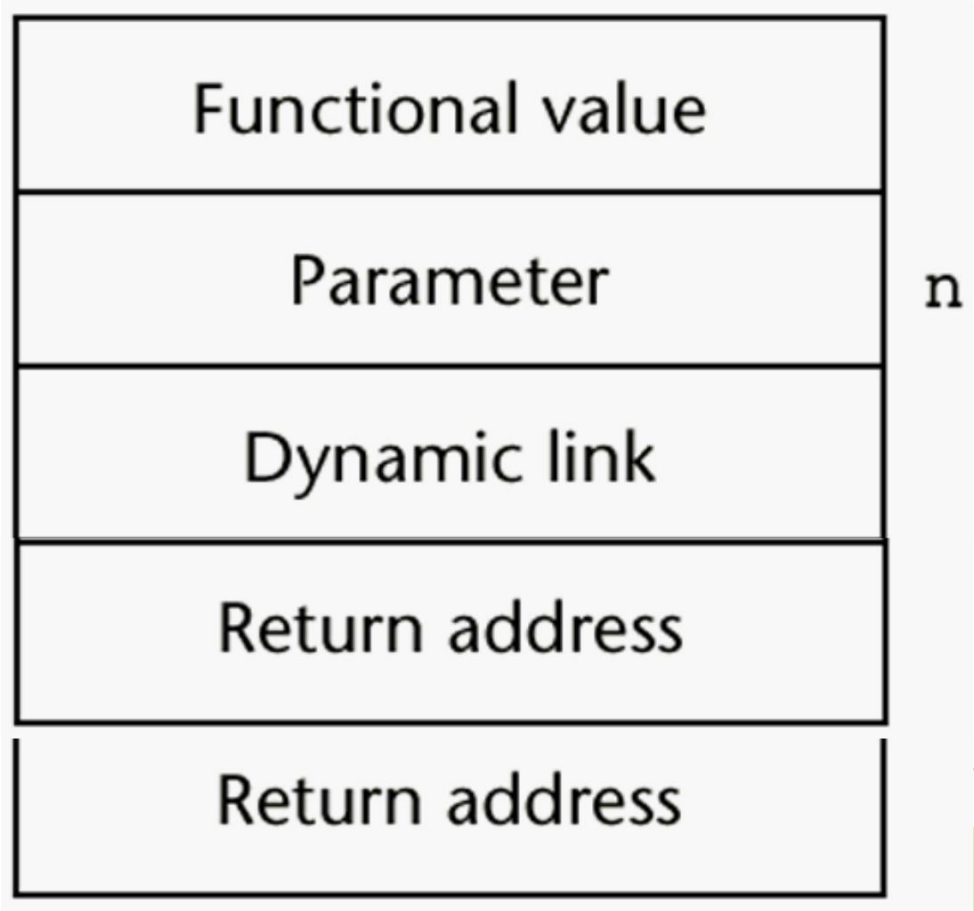
- The collection of dynamic links in the stack at a given time is called the *dynamic chain*, or *call chain*
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the *local_offset*
- The local_offset of a local variable can be determined by the compiler at compile time

An Example With Recursion

The activation record used in the previous example supports recursion, e.g.

```
int factorial (int n) {
   <-------------------------------------------------------------1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
    <-----------------------------2
}
void main() {
  int value;
  value = factorial(3);
    <-------------------------------3
}
```

Activation Record for `factorial`



Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, Python, JavaScript, Ruby, and Lua) use stack-dynamic local variables and allow subprograms to be nested
- All variables that can be non-locally accessed reside in some activation record instance in the stack
- The process of locating a non-local reference:
  - Find the correct activation record instance
  - Determine the correct offset within that activation record instance

Locating a Non-local Reference

Finding the offset is easy

Finding the correct activation record instance

- Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made

Static Scoping

A *static chain* is a chain of static links that connects certain activation record instances

The *static link* in an activation record instance for subprogram A points to one of the activation record instances of A's static parent

The static chain from an activation record instance connects it to all of its static ancestors

Static_depth is an integer associated with a static scope whose value is the depth of nesting of that scope

Static Scoping

(continued)

The *chain_offset* or *nesting_depth* of a nonlocal reference is the difference between the static_depth of the reference and that of the scope when it is declared

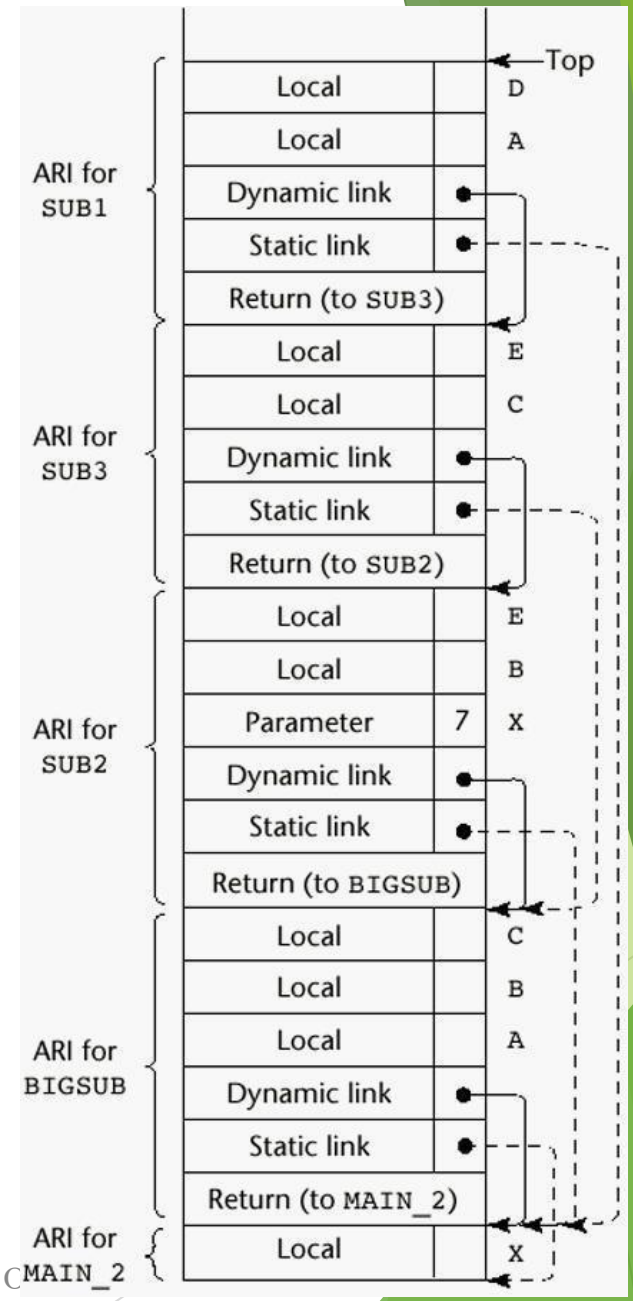
A reference to a variable can be represented by the pair:
(chain_offset, local_offset),
where local_offset is the offset in the activation record of the variable being referenced

Example Ada Program (continued)

Call sequence for `Main_2`

`Main_2` calls `Bigsub`
`Bigsub` calls `Sub2`
`Sub2` calls `Sub3`
`Sub3` calls `Sub1`

Stack Contents at Position 1



Static Chain Maintenance

At the call,

The activation record instance must be built

The dynamic link is just the old stack top pointer

The static link must point to the most recent ari of the static parent

Two methods:

Search the dynamic chain

Treat subprogram calls and definitions like variable references and definitions

Evaluation of Static Chains

- Problems:
  - A nonlocal areference is slow if the nesting depth is large
  - Time-critical code is difficult:
    - Costs of nonlocal references are difficult to determine

Displays

- An alternative to static chains that solves the problems with that approach
- Static links are stored in a single array called a display
- The contents of the display at any given time is a list of addresses of the accessible activation record instances

Blocks

- Blocks are user-specified local scopes for variables
- An example in C
- `{int temp;`
  - `temp = list [upper];`
  - `list [upper] = list [lower];`
  - `list [lower] = temp`
- `}`
  - The lifetime of `temp` in the above example begins when control enters the block
  - An advantage of using a local variable like

Implementing Blocks

Two Methods:

Treat blocks as parameter-less subprograms that are always called from the same location

– Every block has an activation record; an instance is created every time the block is executed

Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Dynamic Scoping

Deep Access: non-local references are found by searching the activation record instances on the dynamic chain

- Length of the chain cannot be statically determined

Every activation record instance must have variable names

Shallow Access: put locals in a central place

- One stack for each variable name
- Central table with an entry for each variable name