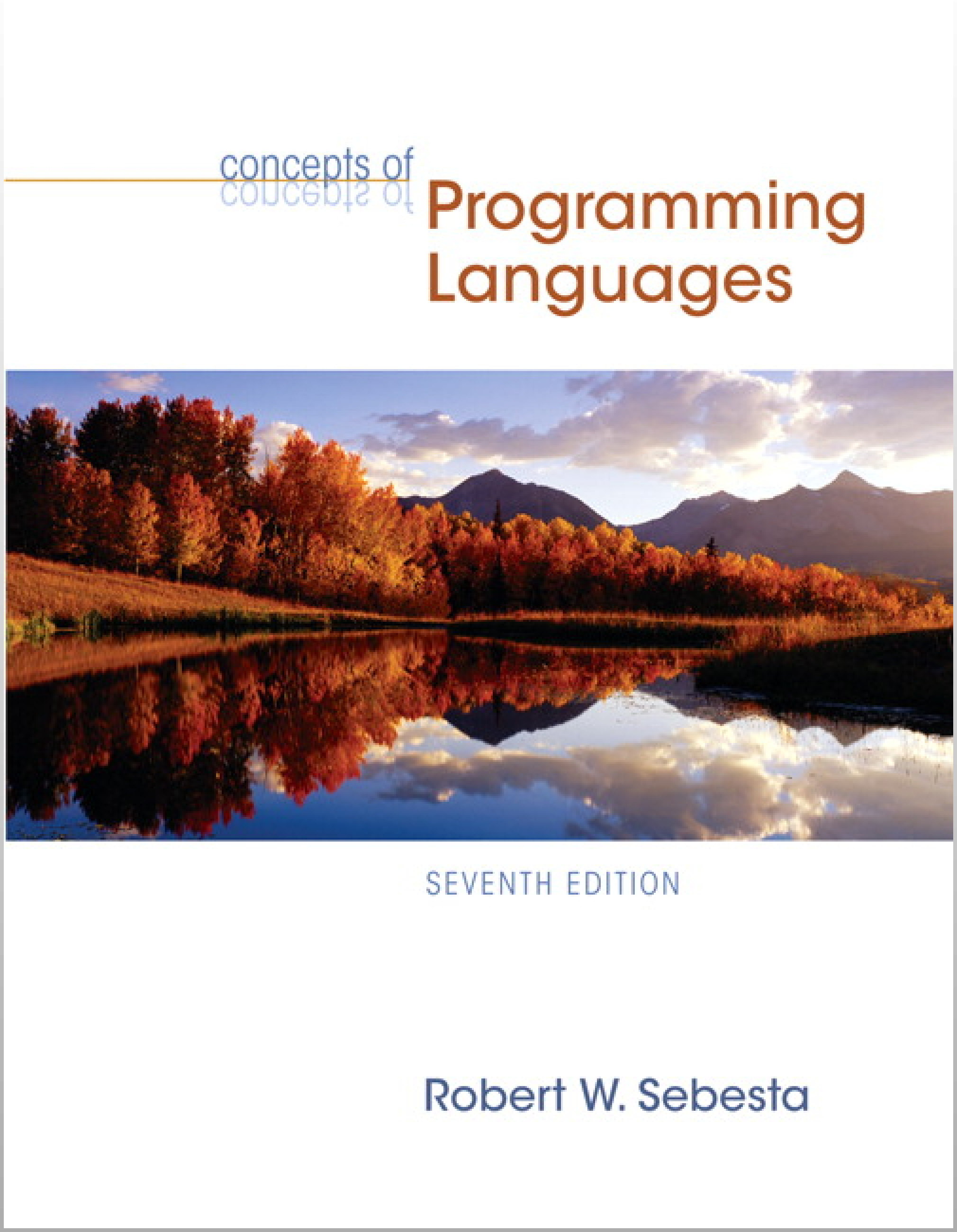
Chapter 10

Implementing Subprograms



ISBN 0-321-33025-0

Chapter 10 Topics

- The General Semantics of Calls and Returns
- Implementing “Simple” Subprograms
- Implementing Subprograms with Stack-Dynamic Local Variables
- Nested Subprograms
- Blocks
- Implementing Dynamic Scoping

The General Semantics of Calls and Returns

- The subprogram call and return operations of a language are together called its *subprogram linkage*
- A subprogram call has numerous actions associated with it
  - Parameter passing methods
  - Static local variables
  - Execution status of calling program
  - Transfer of control
  - Subprogram nesting

Implementing "Simple" Subprograms: Call Semantics

- Save the execution status of the caller
- Carry out the parameter-passing process
- Pass the return address to the callee
- Transfer control to the callee

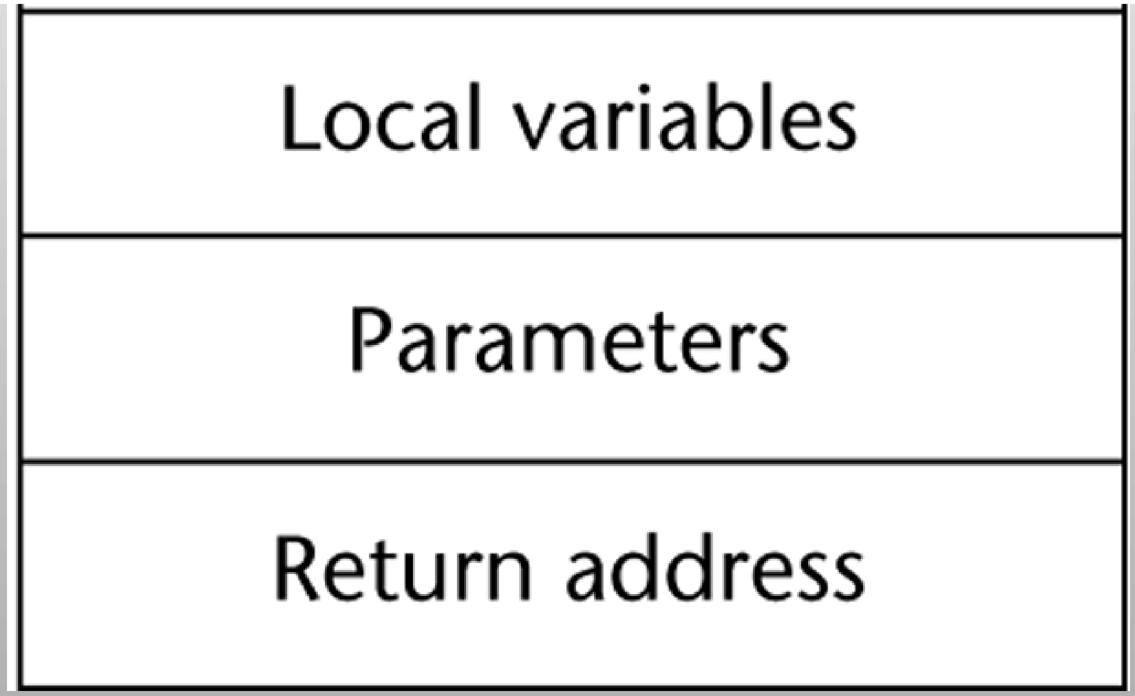
Implementing “Simple” Subprograms: Return Semantics

- If pass-by-value-result 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

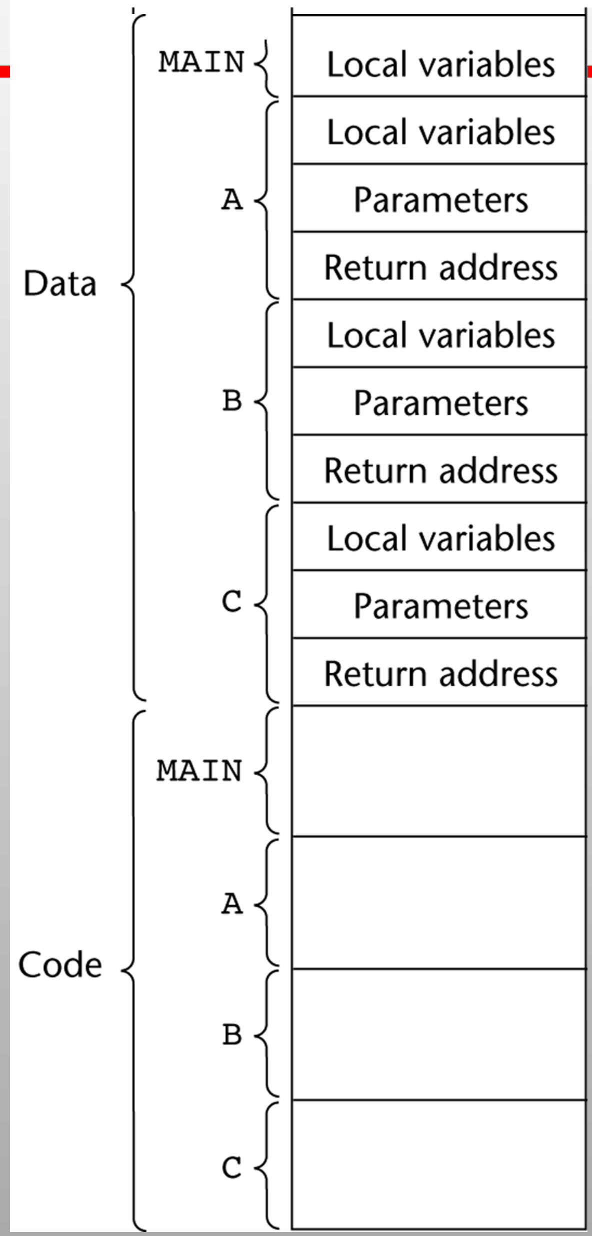
Implementing “Simple” Subprograms: Parts

- Two separate parts: the actual code and the noncode part (local variables and data that can change)
- The format, or layout, of the noncode 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



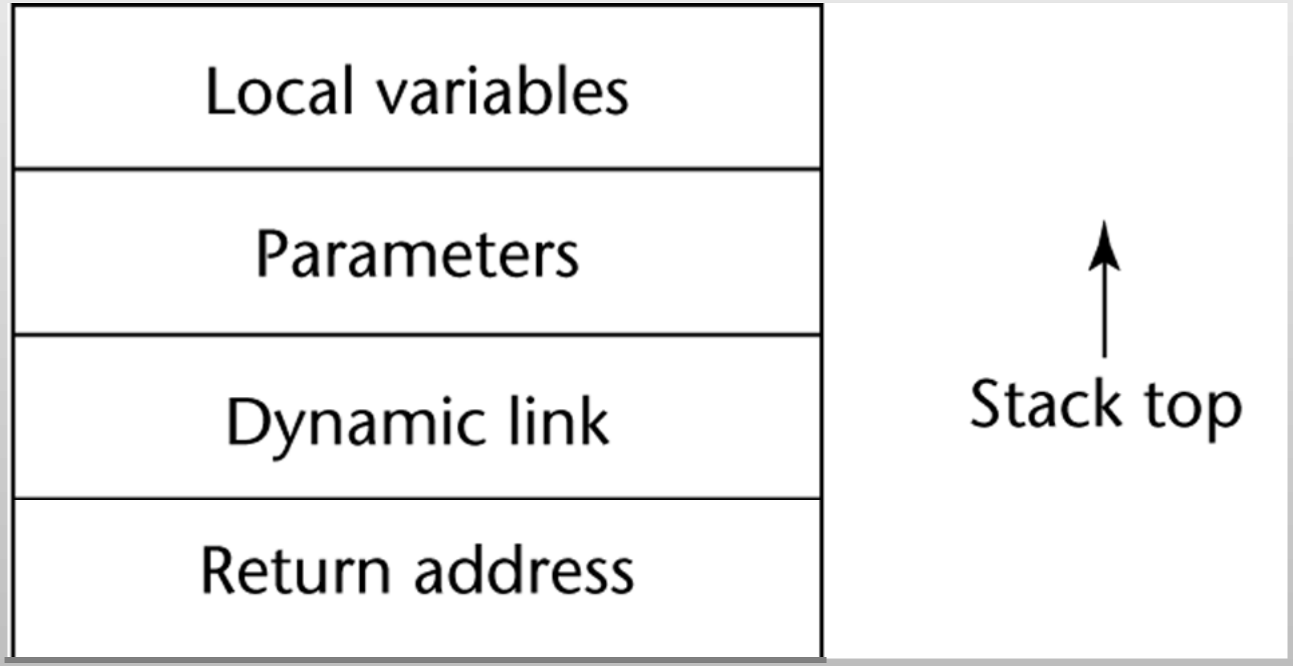
Code and Activation Records of a Program with "Simple" Subprograms



Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
  - The compiler must generate code to cause implicit allocation and de-allocation 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 bottom of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called
- Run-time stack

An Example: C Function

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

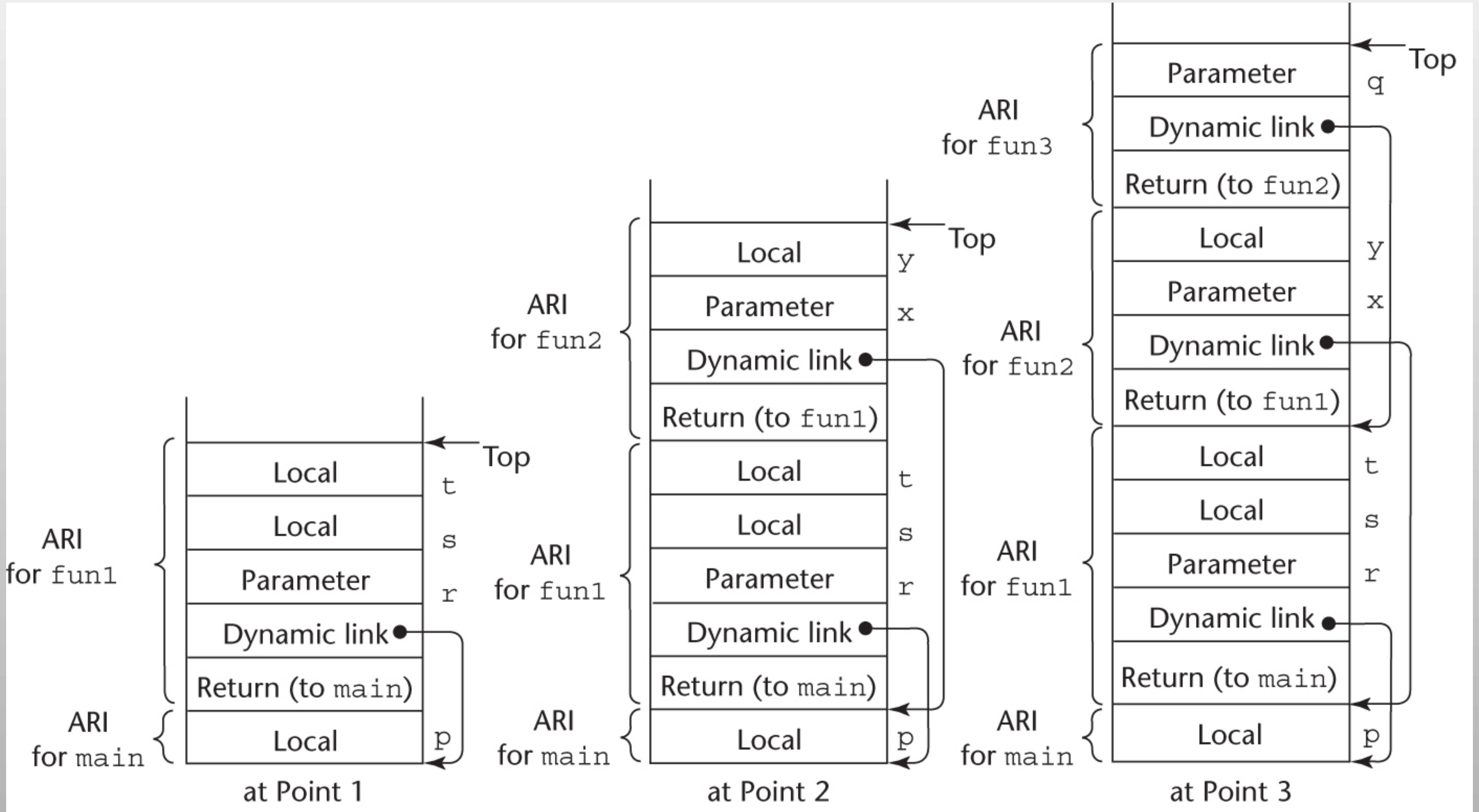
Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parameter	part
Parameter	total
Dynamic link	
Static link	
Return address	

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



ARI = activation record instance

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. 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
}
```

Example with recursion (cont.)

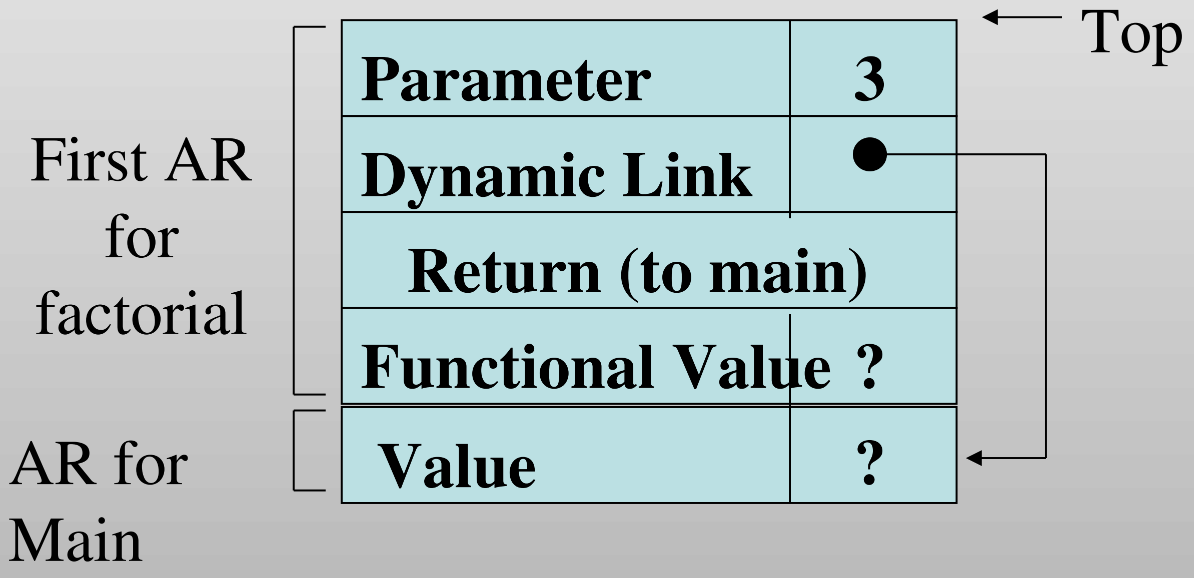
- Activation record format

Parameter
Dynamic Link
Return Address
Functional Value

Top of Stack

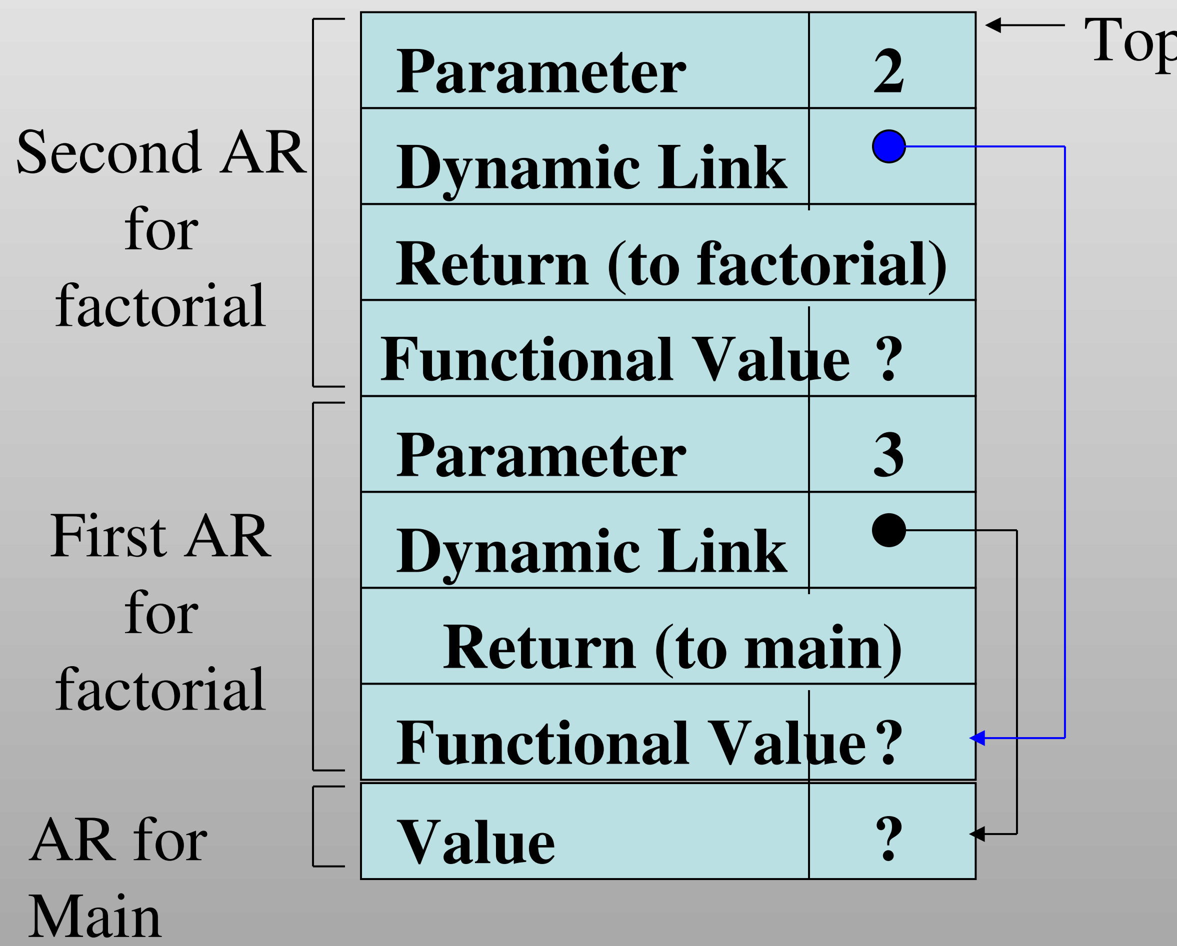
Example with recursion (cont.)

- **factorial() called from Main with a parameter of 3**



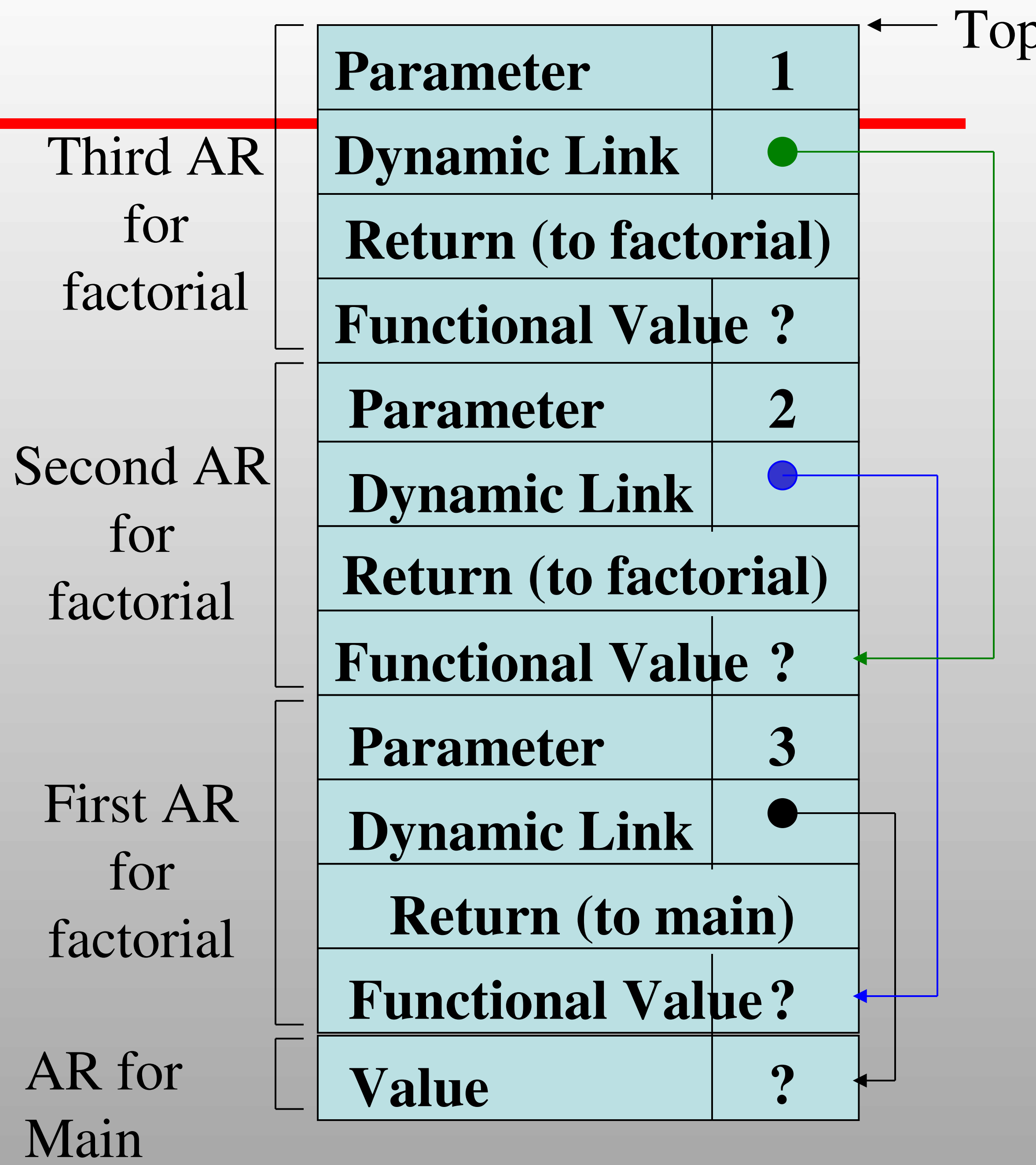
Example with recursion (cont.)

- **factorial(2) called from factorial(3)**
- **Notice how the dynamic link points back to the bottom of the activation record of the calling party? This series of links (or dynamic chain) allows you to trace back the history of execution.**



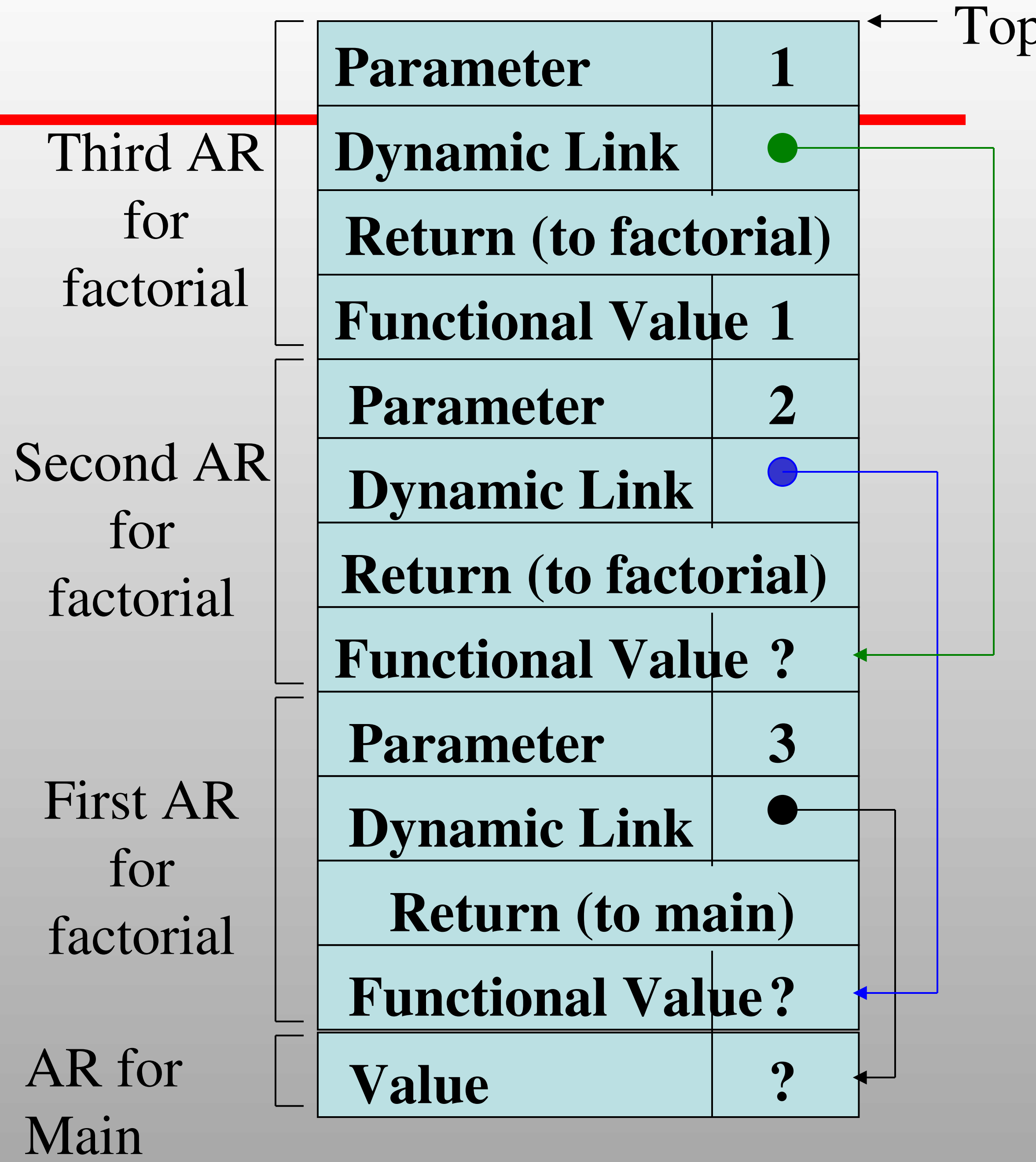
Example with recursion (cont.)

- **factorial(1) called from factorial(2)**



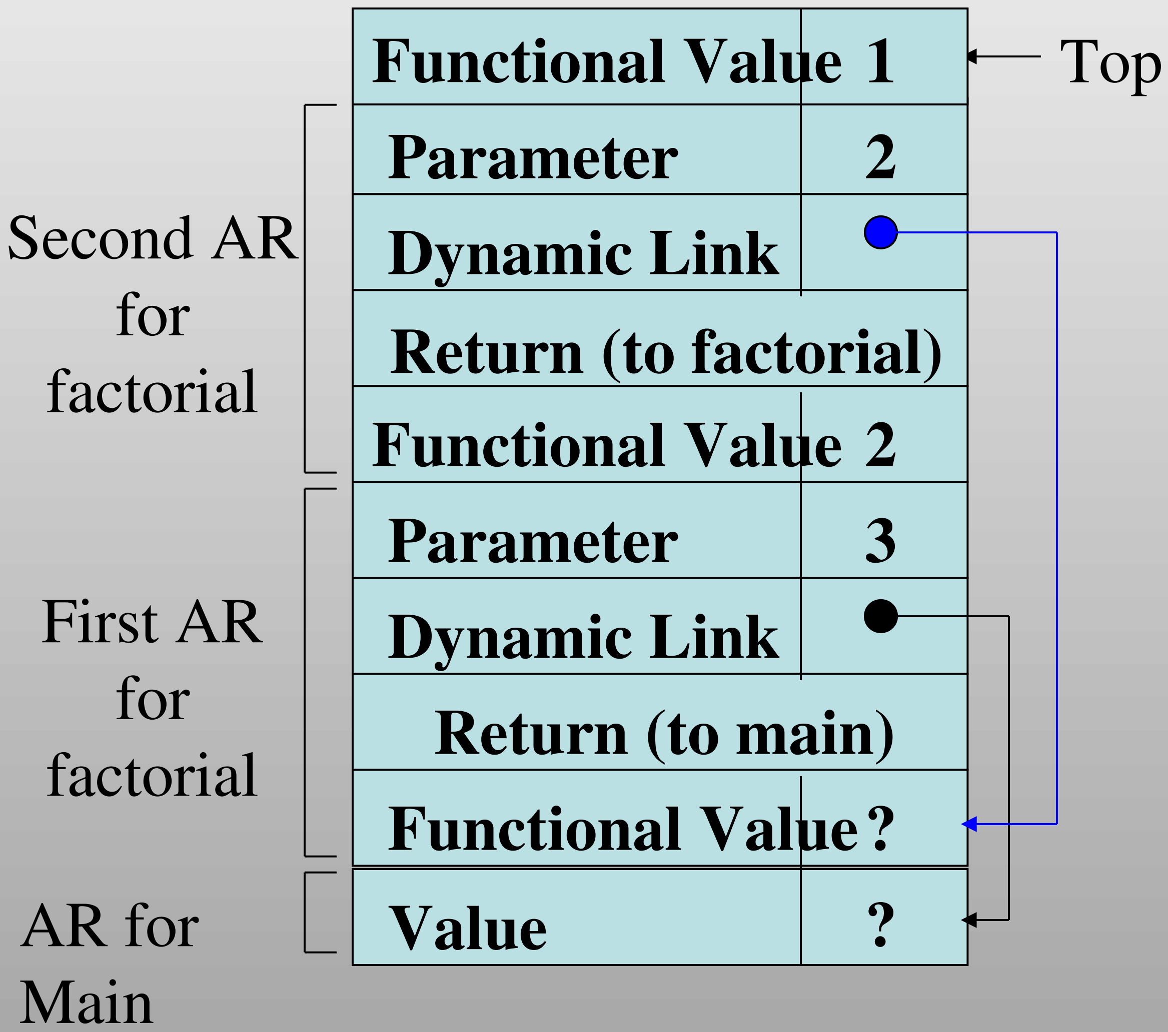
Example with recursion (cont.)

- **factorial(1) finished, and returns a value of 1 via functional value**



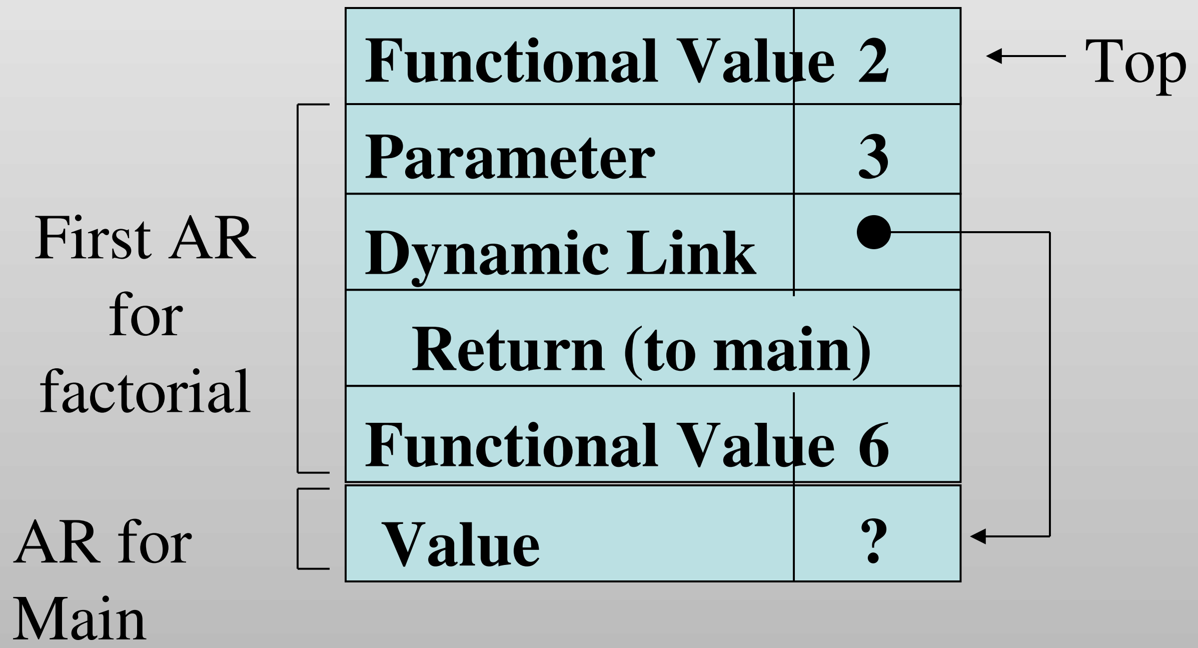
Example with recursion (cont.)

- **factorial(2) finished, and returns a value of 2 via functional value**



Example with recursion (cont.)

- **factorial(3) finishes and calculates a value of 6 to be returned to main via functional value**



Example with recursion (cont.)

- **Main now sets value of local variable 'value' to calculated value of factorial.**

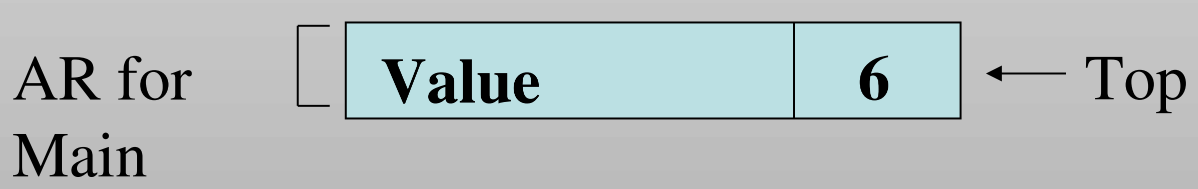
Functional Value	**6**
Value	**6**

← Top

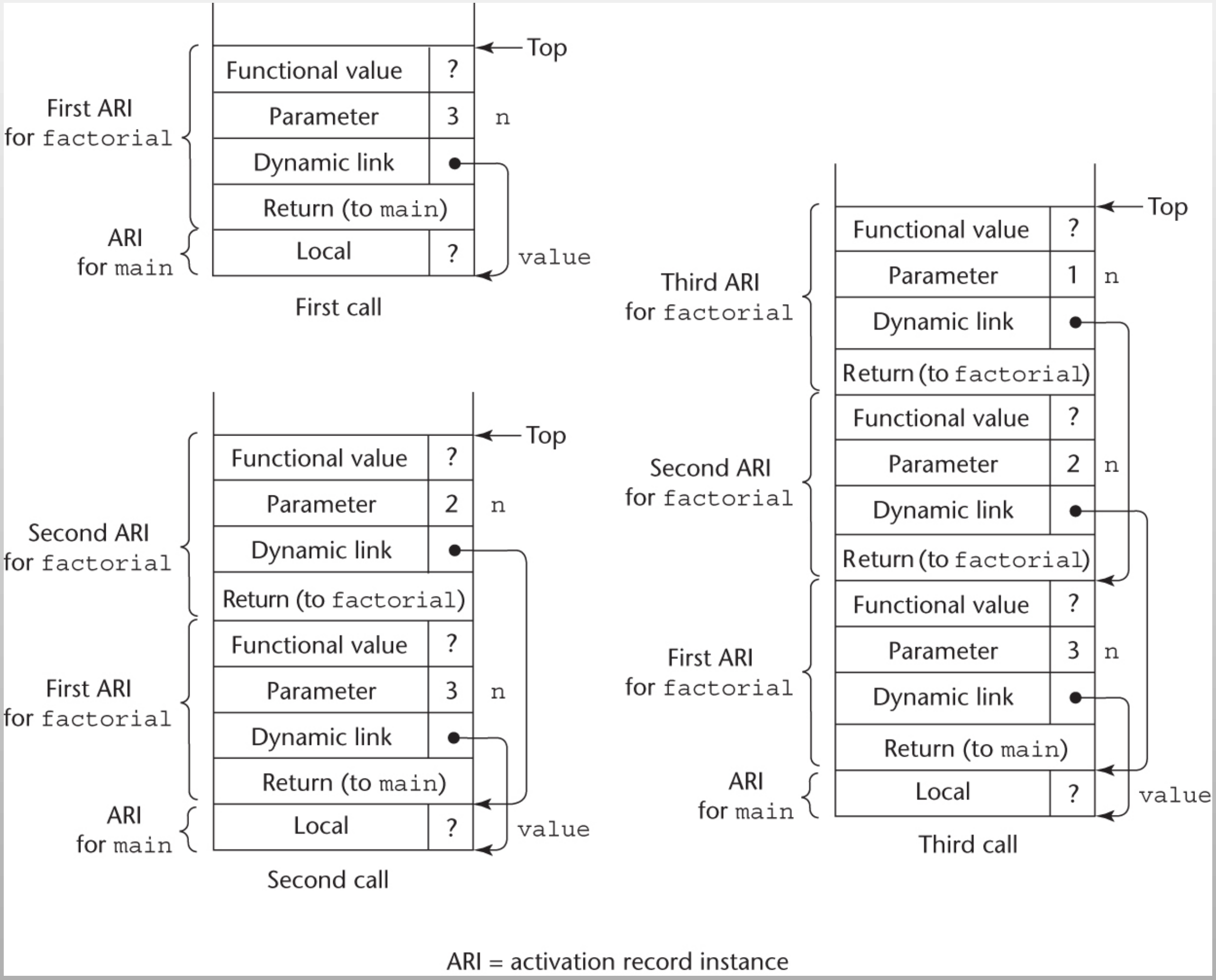
AR for Main

Example with recursion (cont.)

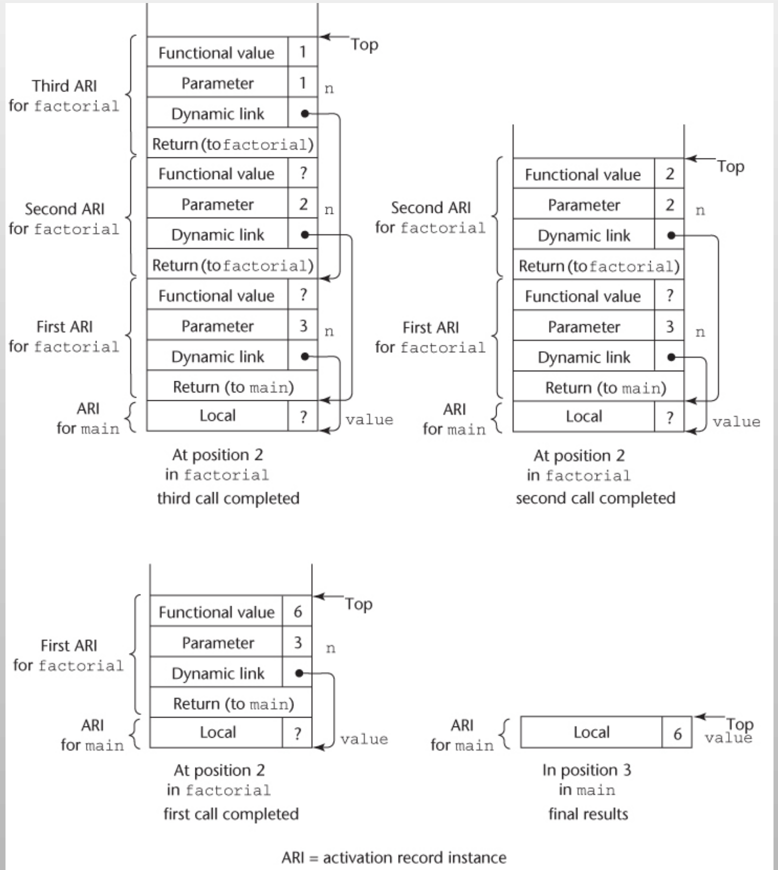
- **Main now sets 'value = 6'**



Stacks for calls to `factorial`



Stacks for returns from `factorial`



Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, JavaScript) 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:
  1. Find the correct activation record instance
  2. 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 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 Pascal Program

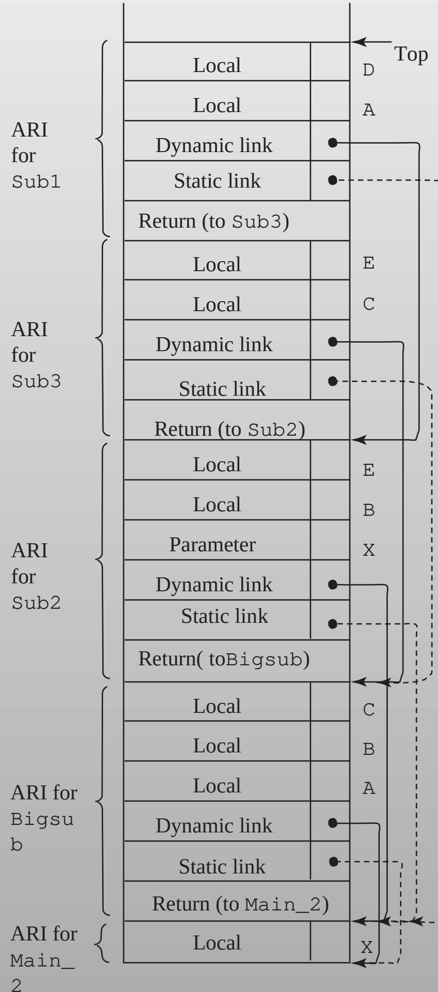
```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <----------------------- 1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <-------------------- 2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <----------------------- 3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end; { MAIN_2 }
```

Stack Contents at Position 1

Call sequence for `MAIN_2`

`MAIN_2` calls `BIGSUB`
`BIGSUB` calls `SUB2`
`SUB2` calls `SUB3`
`SUB3` calls `SUB1`

Stack state at “Sub1” in the program



Evaluation of Static Chains

- Problems:
  1. A nonlocal areference is slow if the nesting depth is large
  2. Time-critical code is difficult:
     a. Costs of nonlocal references are difficult to determine
     b. Code changes can change the nesting depth, and therefore the cost

Displays

- An alternative to static chains
- 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

Summary

- Subprogram linkage semantics requires many action by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex
- Subprograms with stack-dynamic local variables and nested subprograms have two components
  - actual code
  - activation record

Summary (continued)

- Activation record instances contain formal parameters and local variables among other things
- Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or thru some central variable table method