Systems Software: Implementing Subprograms

Andrew Harn

University of Central Florida

Subprogram Linkage

- **Subprograms** are functional units of a program, commonly referred to as a function, procedure, or method
- **Subprogram linkage** refers to the call and return operations of the subprograms
  - Subprograms are “linked” by calls and returns of each other
- Designing linkage requires consideration on numerous actions associated with it
  - Parameter passing methods
  - Static local variables
  - Execution status of calling program
  - Transfer of control
  - Subprogram nesting

Subprogams: 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

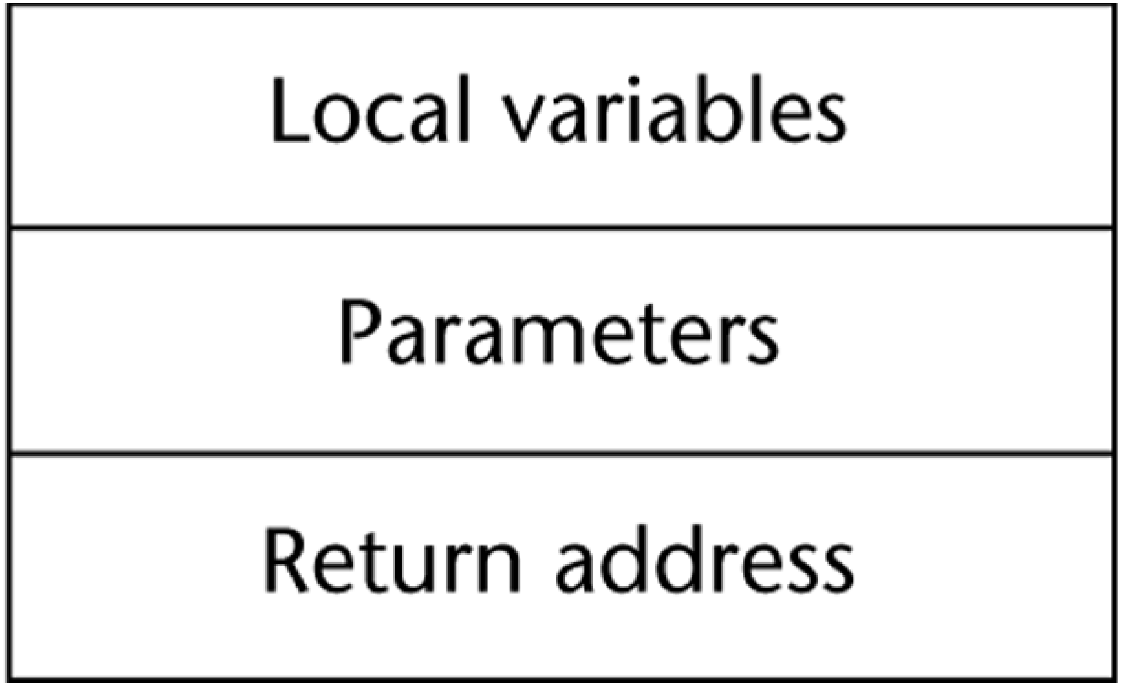
Subprograms: Return Semantics

- If the result is a value, then move the value to return over to the given address
- Restore the execution status of the caller
- Transfer control back to the caller

Implementing Subprograms

- Two parts of the subprogram:
  - The code
  - The non-code (local variables and data that can change)
- The non-code part of an executing subprogram is called an **activation record**
- An activation record can have many instances (a particular subprogram is called multiple times)

Basics of an Activation Record

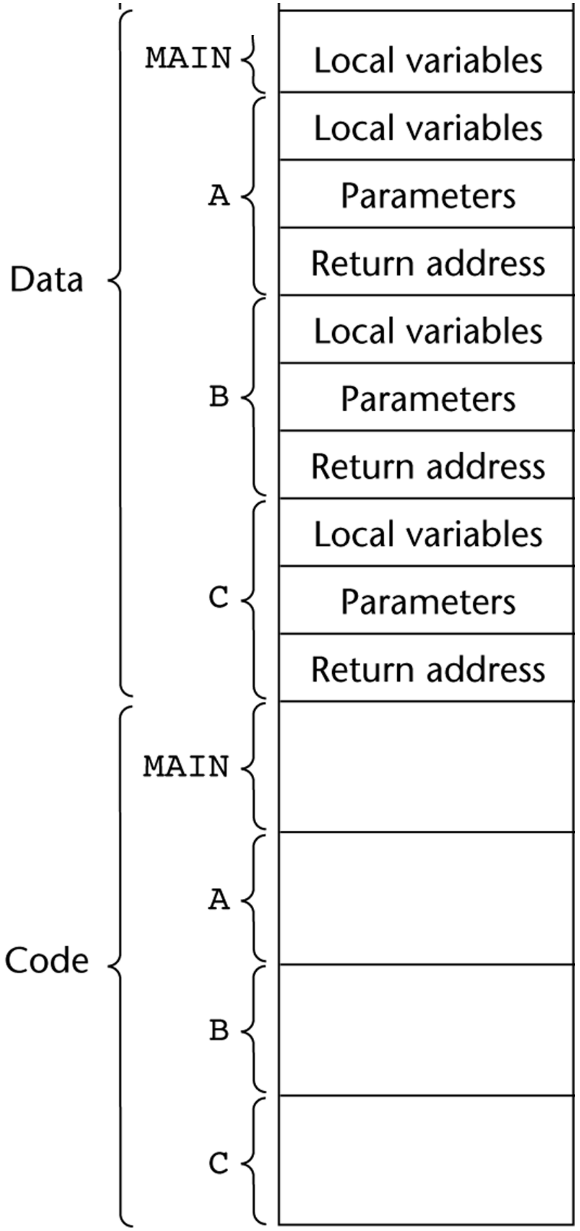


Local variables

Parameters

Return address

Activation Record Example



Data

MAIN: Local variables

A: Local variables, Parameters, Return address

B: Local variables, Parameters, Return address

C: Local variables, Parameters, Return address

Code

MAIN

A

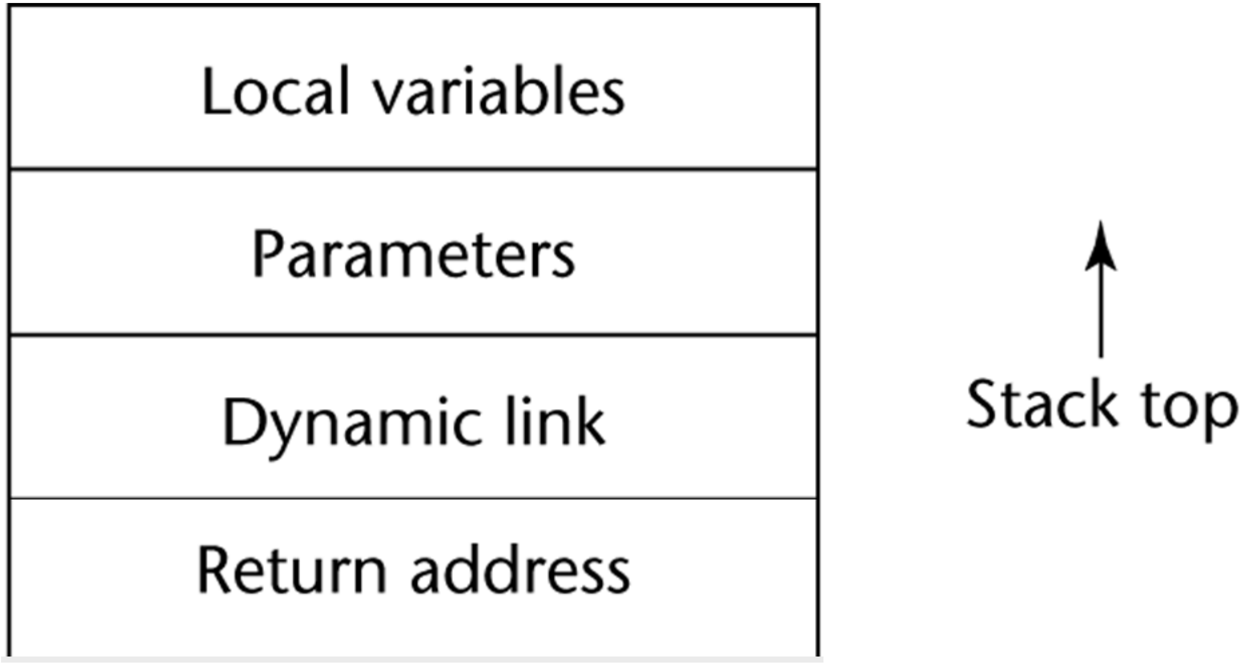
B

C

Stack-Dynamic Local Variables

- Number of instances for an activation record can be a quantity that’s not one
- Causes activation record design to be more complex
  - 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



Local variables

Parameters

Dynamic link

Return address

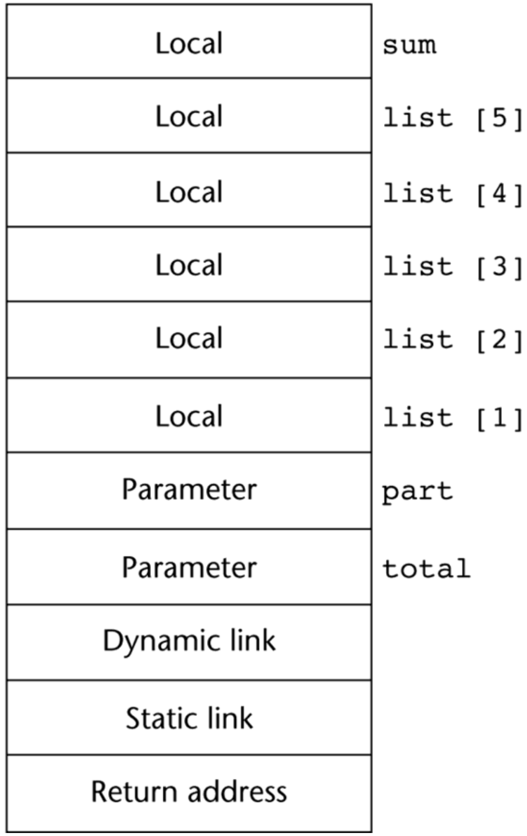
Stack top

Implementing the Activation Record

- Static format yet dynamic size for the stack
- The dynamic link points to the top of the instance of the activation record of the caller
- Instances of activation records are dynamically created due to unknown size of the local variables
- Stack is made on the fly (run-time stack)

An Example in C

```
void sub(float total, int part)
{
  int list[4];
  float sum;
  …
}
```



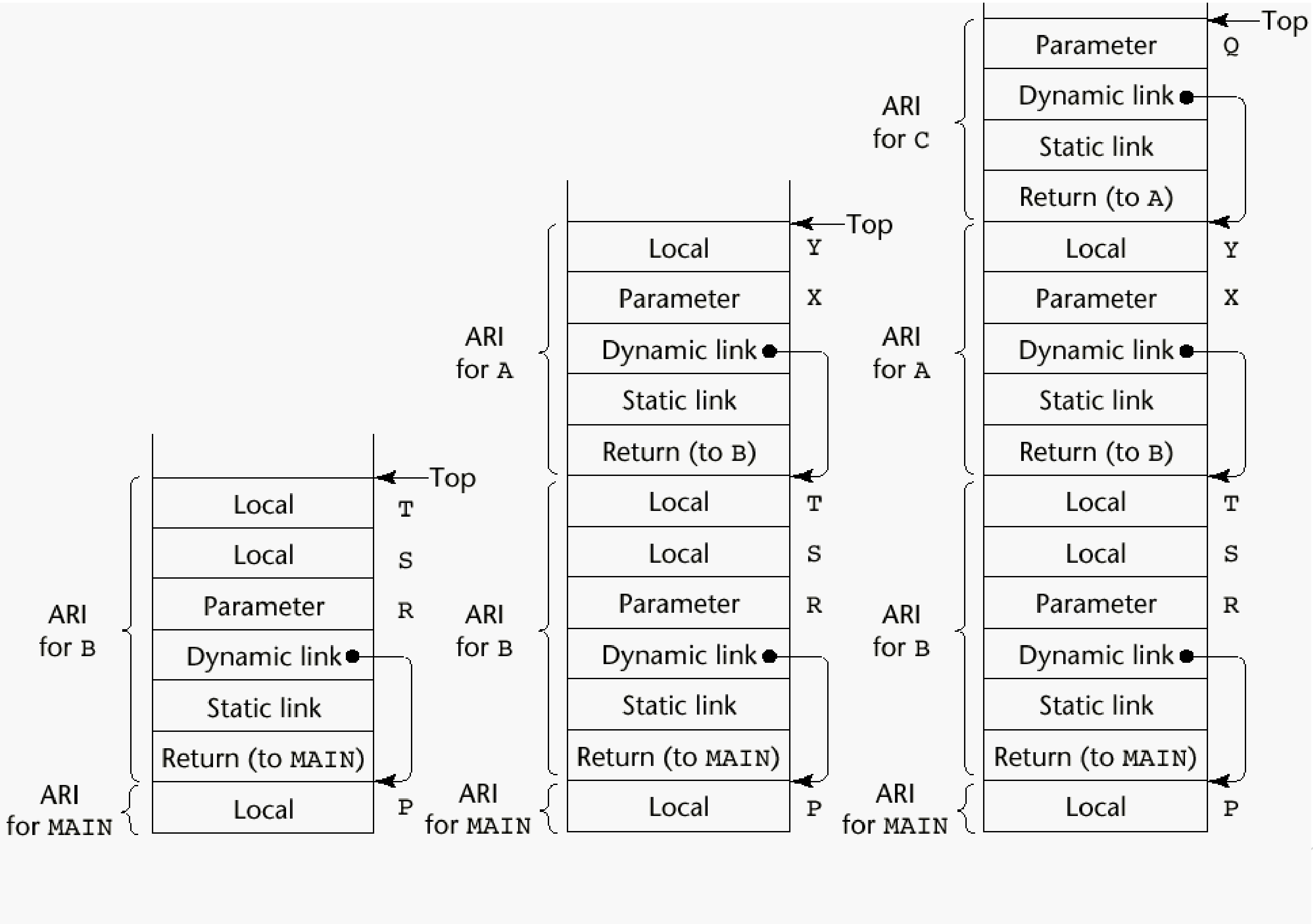
Local	sum
Local	list [5]
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Parameter	part
Parameter	total
Dynamic link	
Static link	
Return address	

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 (Continued)



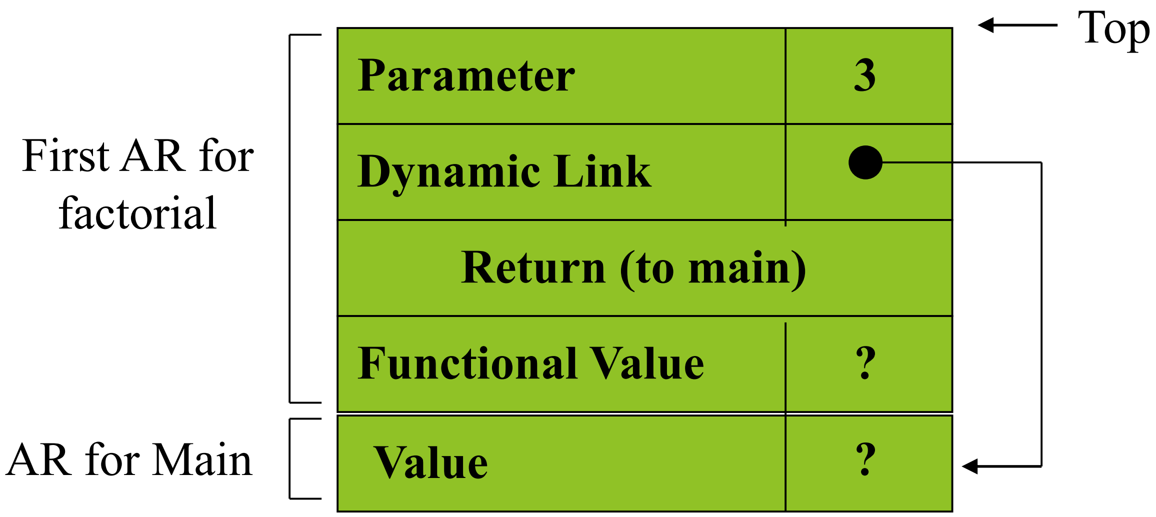
Dynamic Chain and Local Offset

- The collection of dynamic links in the stack at a given time is called the **dynamic chain**, or **call stack**
  - Used to differentiate activation records and understand which subprograms called what
- Within an activation record, local variables are identified by how far they are from the base of the activation record. This offset is called the **local offset**
  - This offset can be determined during compilation

An Example of Recursion

```
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 (Continued): Activation Record of Main() and Factorial(3)



Top

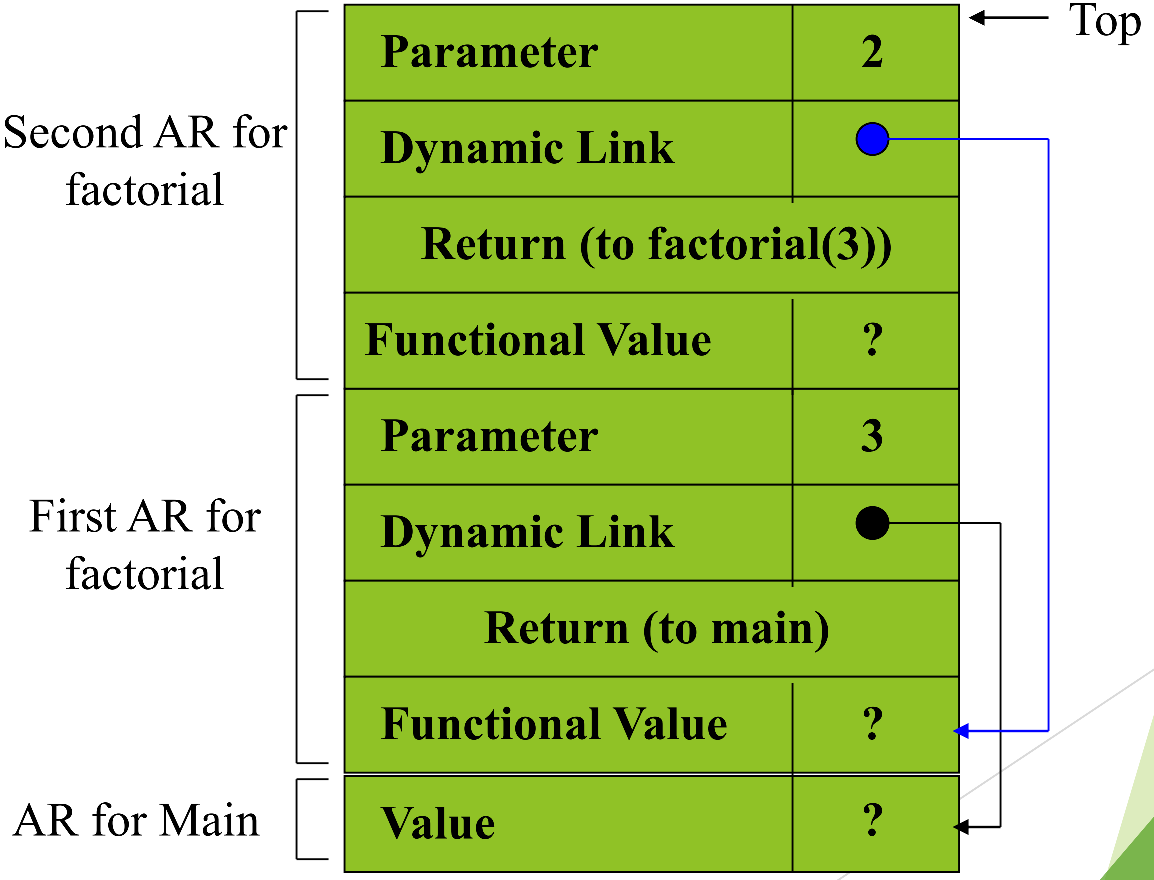
First AR for factorial

Parameter	3
Dynamic Link	●
Return (to main)	
Functional Value	?
Value	?

AR for Main

Example with Recursion (Continued): With call to Factorial(2)

- The dynamic link of factorial(2) points at the base of the subprogram that called it, which is factorial(3)
- This dynamic link and the link in factorial(3) is the current dynamic chain, which shows the trace back of the program execution



Top

Second AR for factorial

Parameter	2
Dynamic Link	●
Return (to factorial(3))	
Functional Value	?

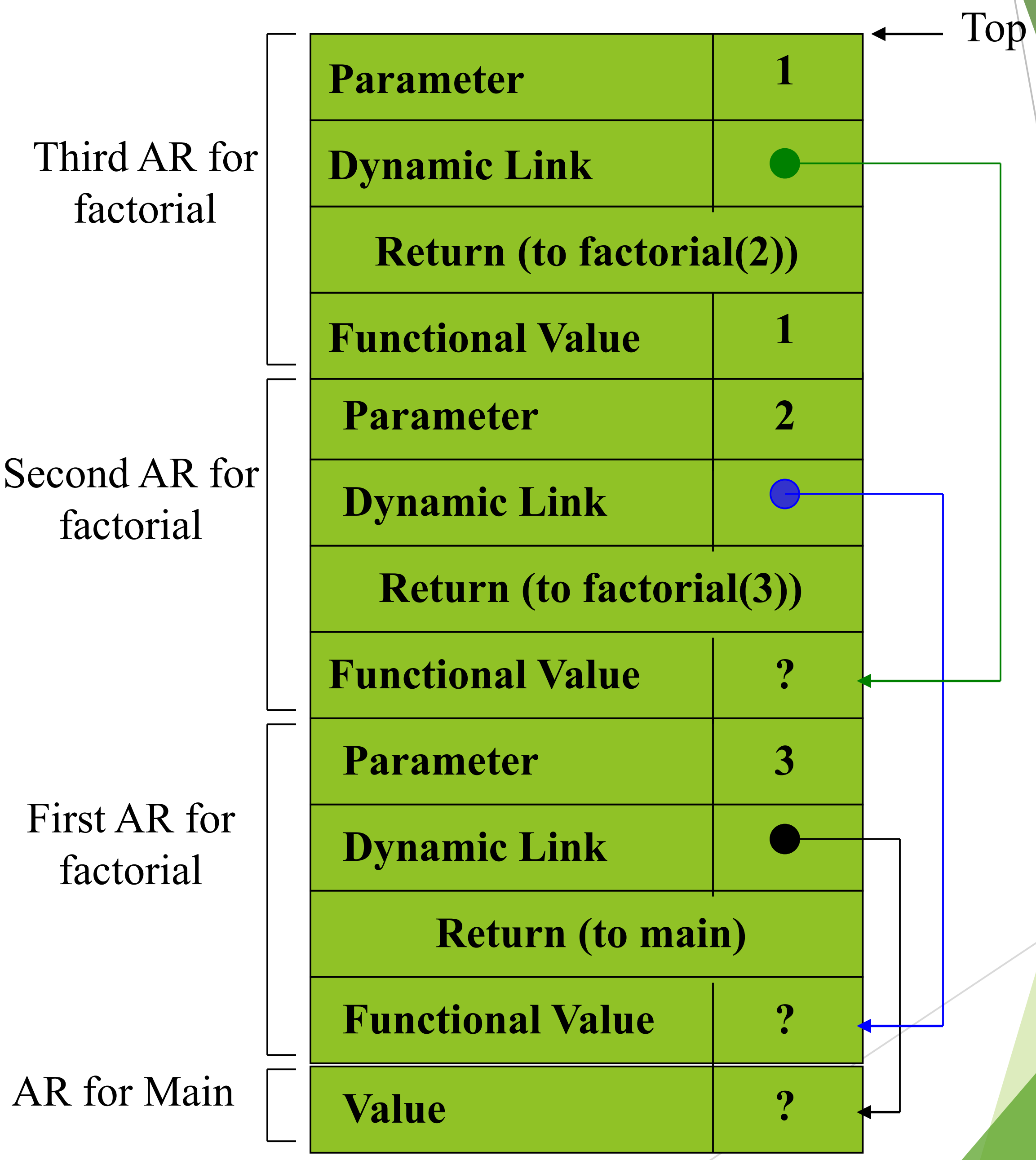
First AR for factorial

Parameter	3
Dynamic Link	●
Return (to main)	
Functional Value	?

AR for Main

Value	?

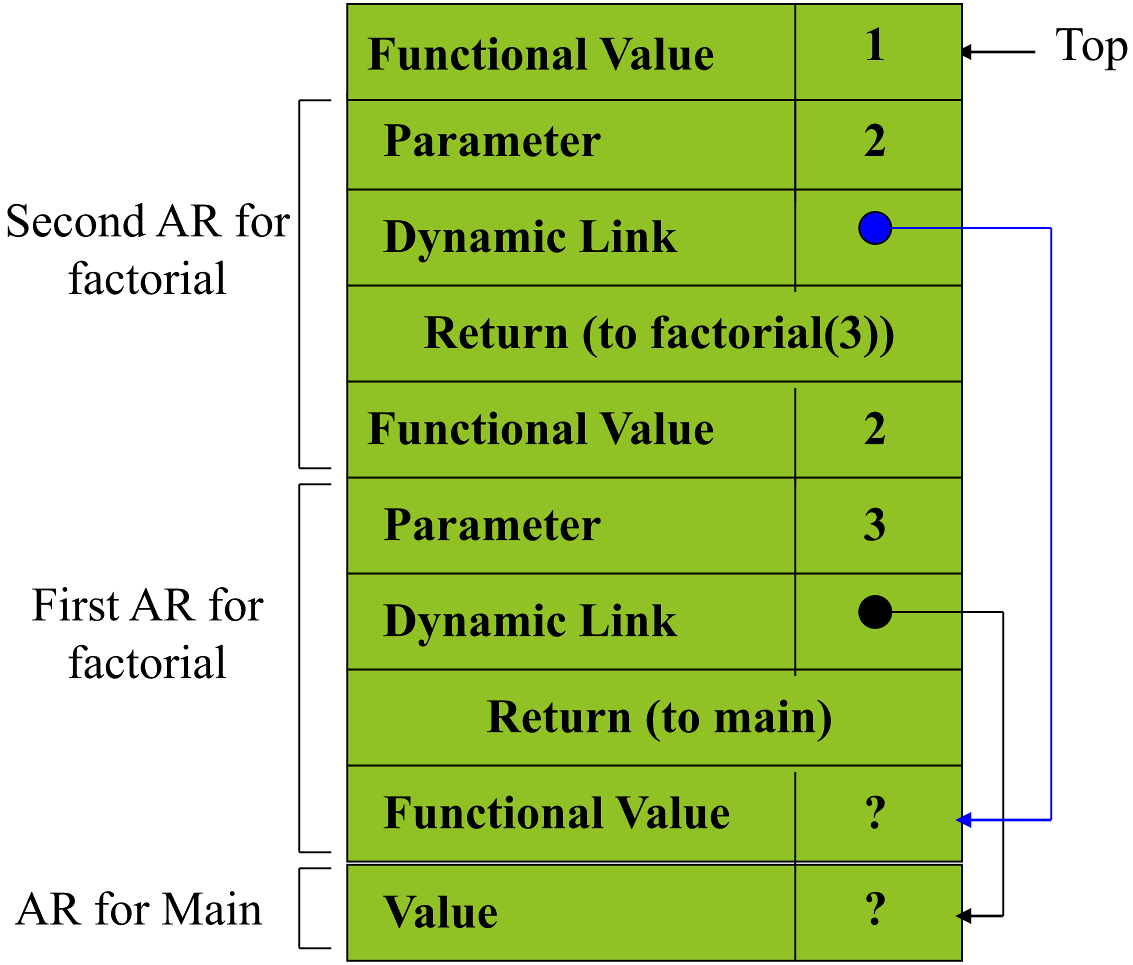
Example (Continued)



Top

Third AR for factorial	Parameter	1
	Dynamic Link	●
	Return (to factorial(2))	
	Functional Value	1
Second AR for factorial	Parameter	2
	Dynamic Link	●
	Return (to factorial(3))	
	Functional Value	?
First AR for factorial	Parameter	3
	Dynamic Link	●
	Return (to main)	
	Functional Value	?
AR for Main	Value	?

Example (Continued): Returning values



Top

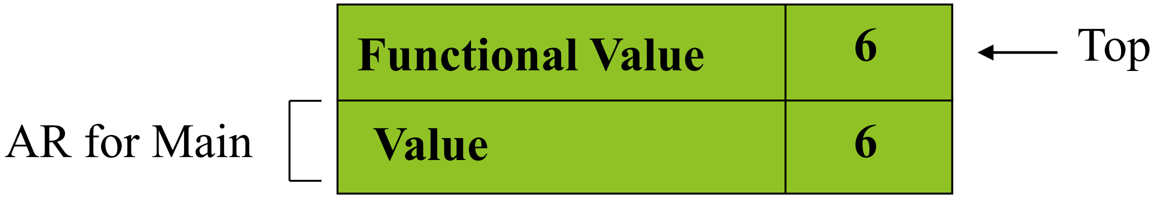
Second AR for factorial

First AR for factorial

AR for Main

Functional Value	1
Parameter	2
Dynamic Link	●
Return (to factorial(3))	
Functional Value	2
Parameter	3
Dynamic Link	●
Return (to main)	
Functional Value	?
Value	?

Example (Continued): Returning values



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

← Top

AR for Main

Nested Subprograms

- Some static-scoped languages (such as Fortran 95, Ada, and JavasScript) use stack-dynamic local variables as well as nested subprograms
- All variables that can be non-locally accessed reside in some activation record instance in the stack
- To locate a non-local reference:
  - Find the correct activation record instance
  - Determine the correct offset within that activation record instance

Locating a Non-local Reference: Static Scoping

- Finding the offset is easy
- Finding the correct activation record instance
  - Static semantic rules guarantee that the instance exists
- A **static link** in an activation record instance for a subprogram A points to an activation record instance of A's static parent
  - The set of static links in the stack is the **static chain**

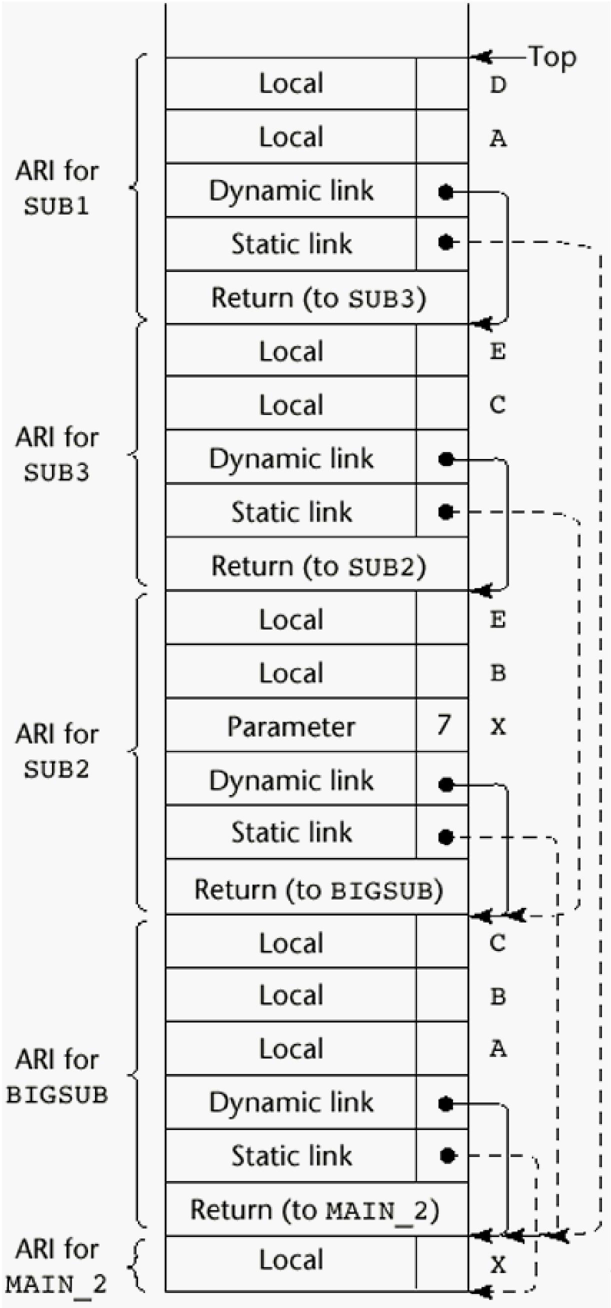
Example Pascal Program

```
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 }
```

Call sequence for `MAIN_2`

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

Stack Contents at Position 1



Top

ARI for SUB1

Local D

Local A

Dynamic link

Static link

Return (to SUB3)

ARI for SUB3

Local E

Local C

Dynamic link

Static link

Return (to SUB2)

ARI for SUB2

Local E

Local B

Parameter 7 X

Dynamic link

Static link

Return (to BIGSUB)

ARI for BIGSUB

Local C

Local B

Local A

Dynamic link

Static link

Return (to MAIN_2)

ARI for MAIN_2

Local X

Displays & Blocks

- **Displays** are an alternative to static chains
  - Static links are stored in a single array called a display
  - The contents of a display at any given time is a list of address of the accessible activation record instances
- **Blocks** are user-defined local scopes for variables
  - In C, just surround a custom scope with brackets
  - All variables defined in that scope are only defined for that block
  - Can be implemented by treating blocks as anonymous parameter-less subprograms that are always called from the same location
    - Size can be statically determined at compile time