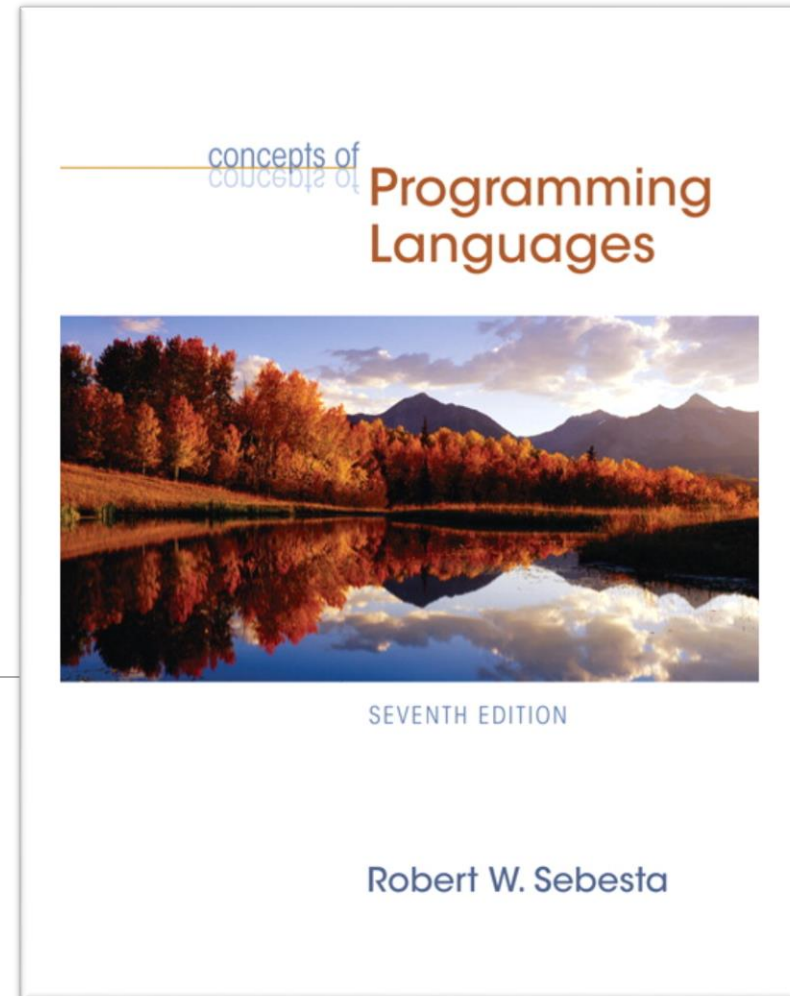


# Lecture 6 (Chapter 10)

COP3402 FALL 2015 – 9/16/2015  
FROM ADDISON-WESLEY, 2006



# An Example With Recursion

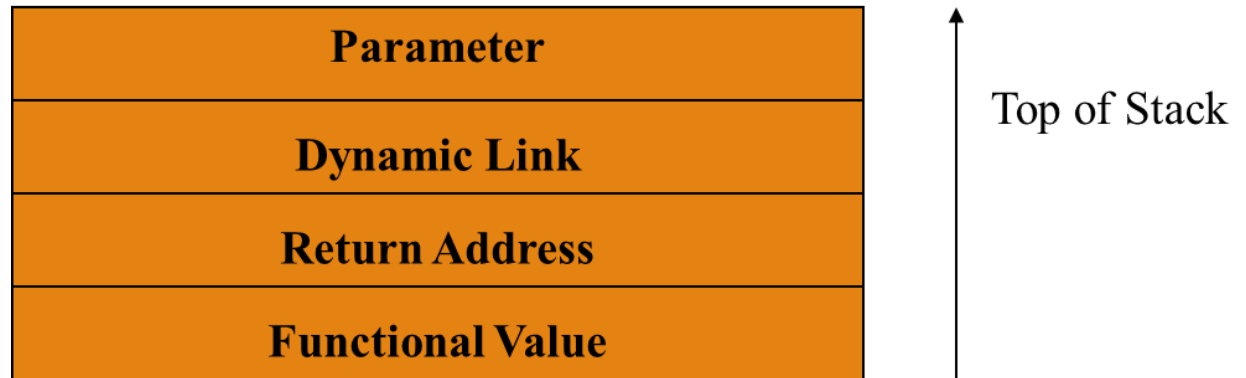
---

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  
}
```

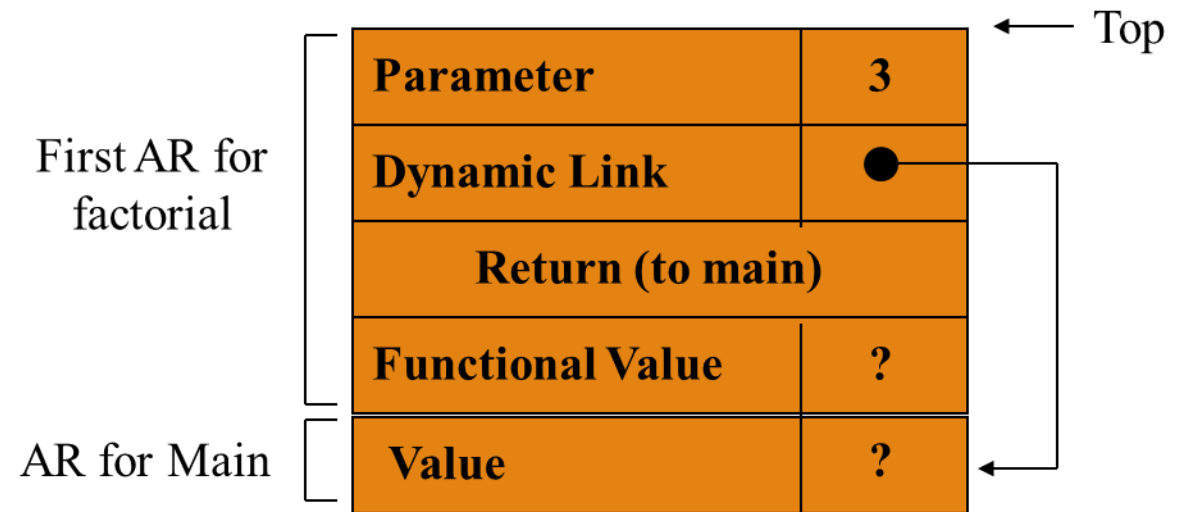
# Example with recursion (cont.)

Activation record format



# Example with recursion (cont.)

factorial() called from Main with a parameter of 3

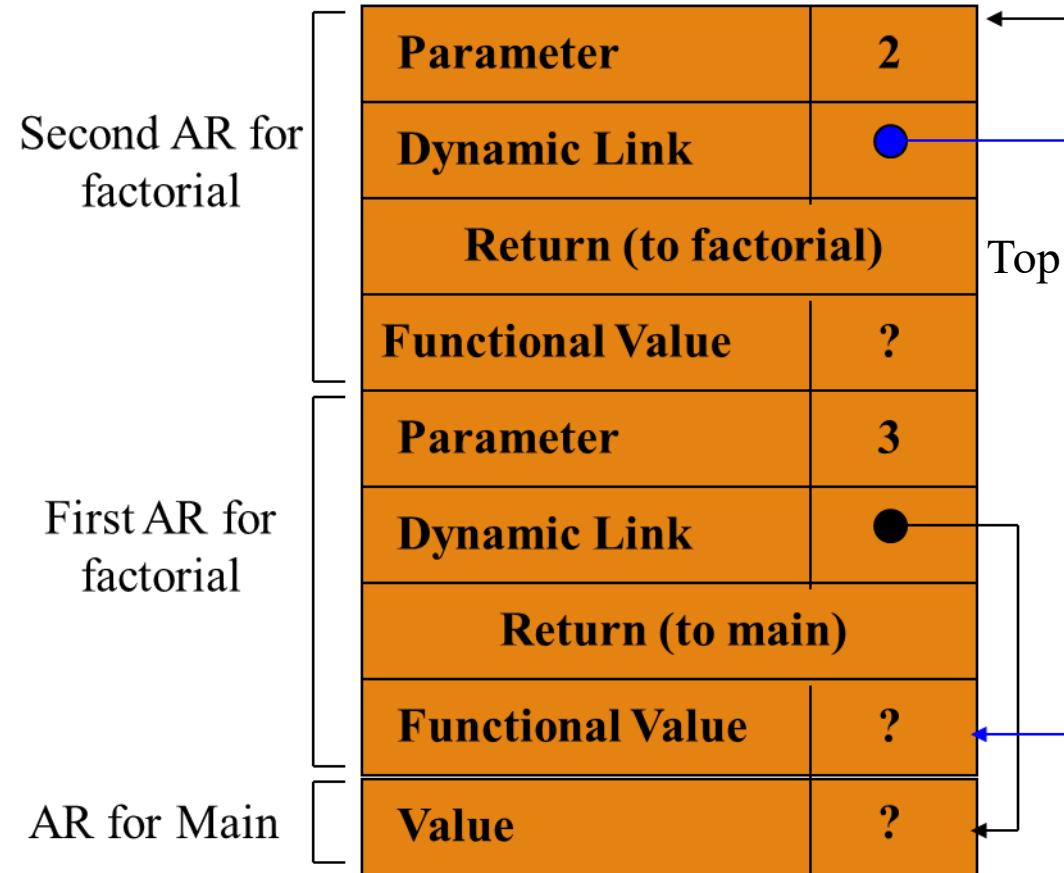


# Example with recursion (cont.)

factorial(2) called from factorial(3)

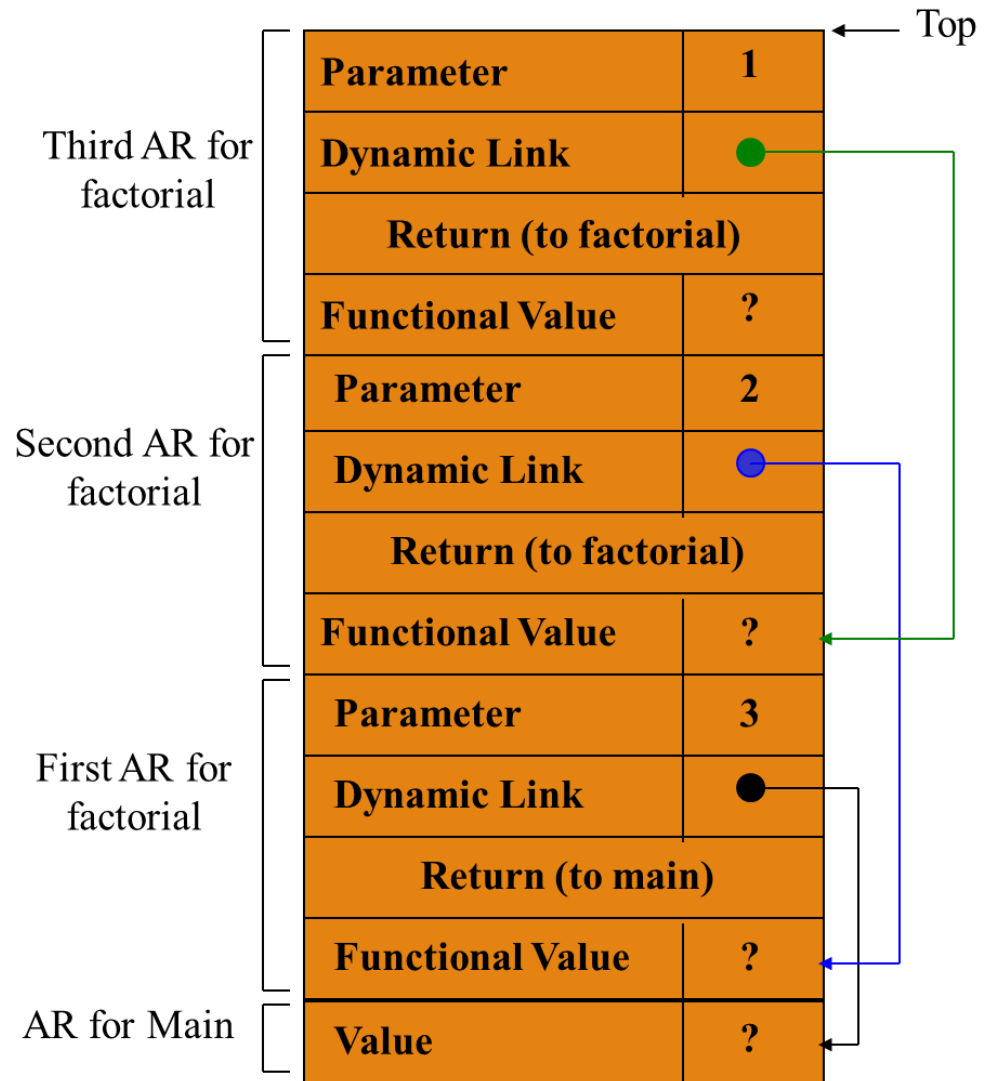
Notice how the dynamic link points back to the bottom of the activation record of the calling party?

This series of links (or *dynamic chain*) allows you to trace back the history of execution.



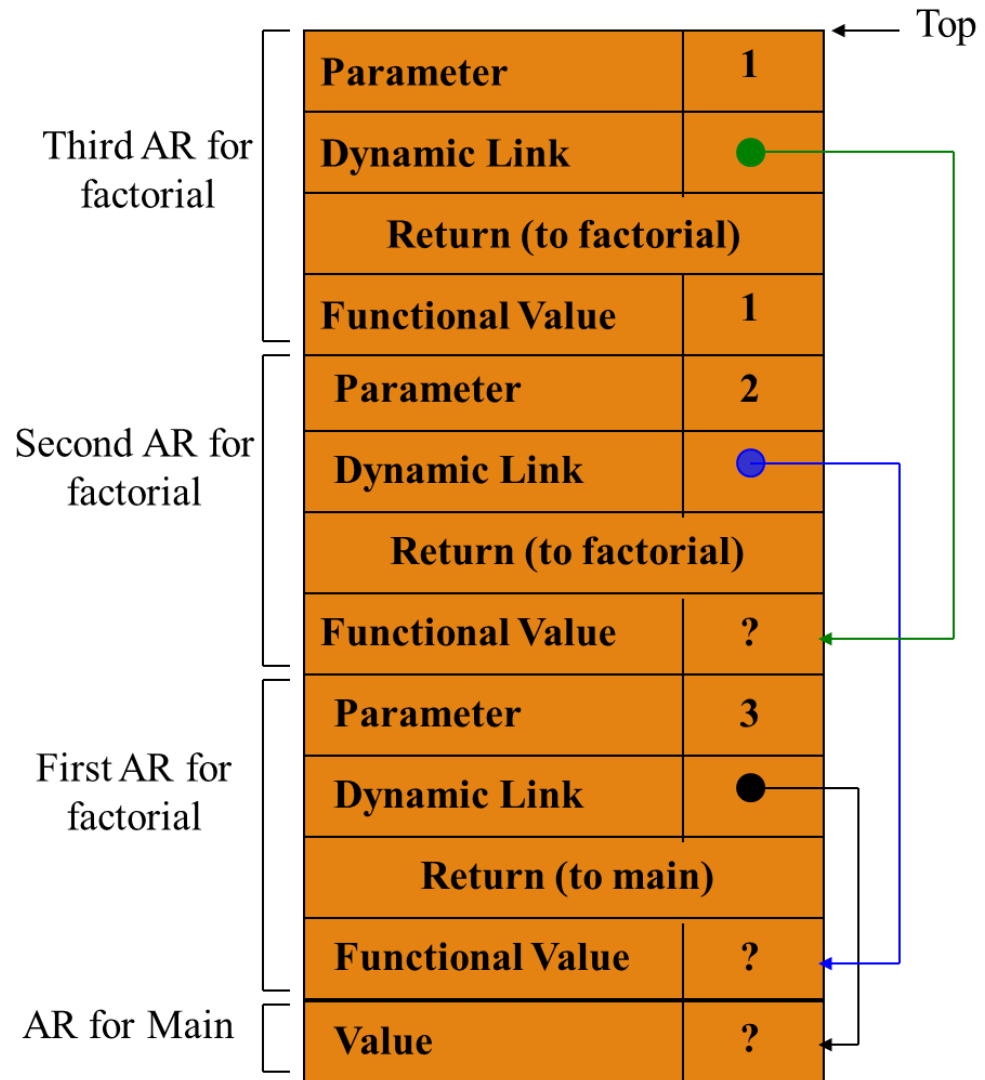
# Example with recursion (cont.)

factorial(1) called from factorial(2)



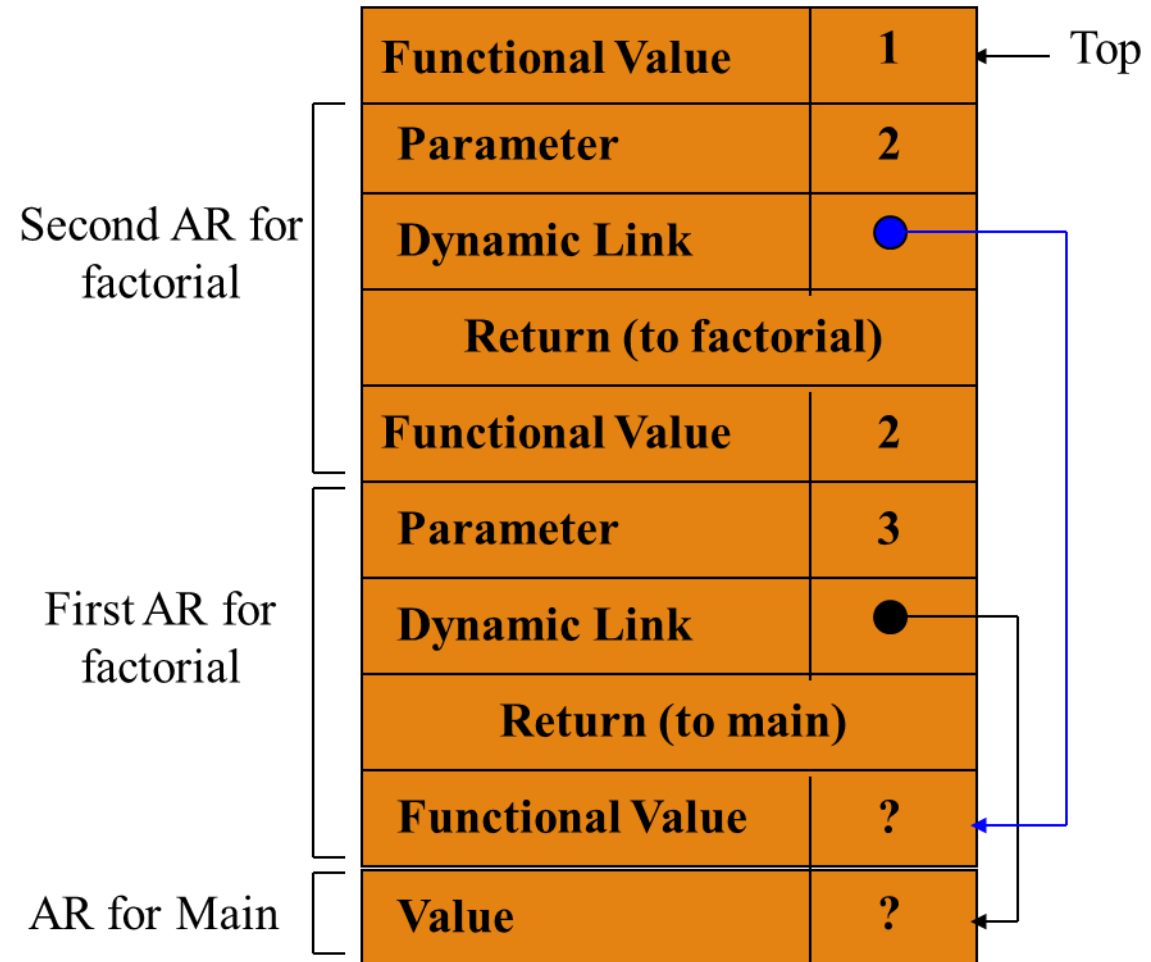
# Example with recursion (cont.)

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



# Example with recursion (cont.)

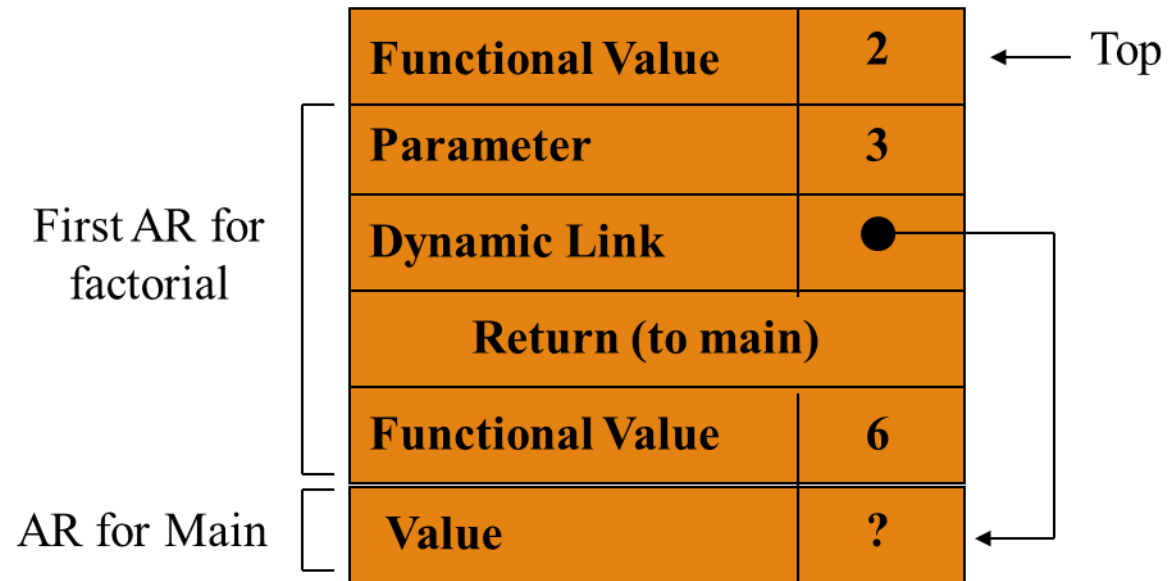
factorial(2) finished, and returns a value of 2 via functional value





## 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.

AR for Main	<b>Functional Value</b>	<b>6</b>	← Top
	<b>Value</b>	<b>6</b>	

# Example with recursion (cont.)

Main now sets 'value = 6'



# 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:
  - 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

# 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 }
```

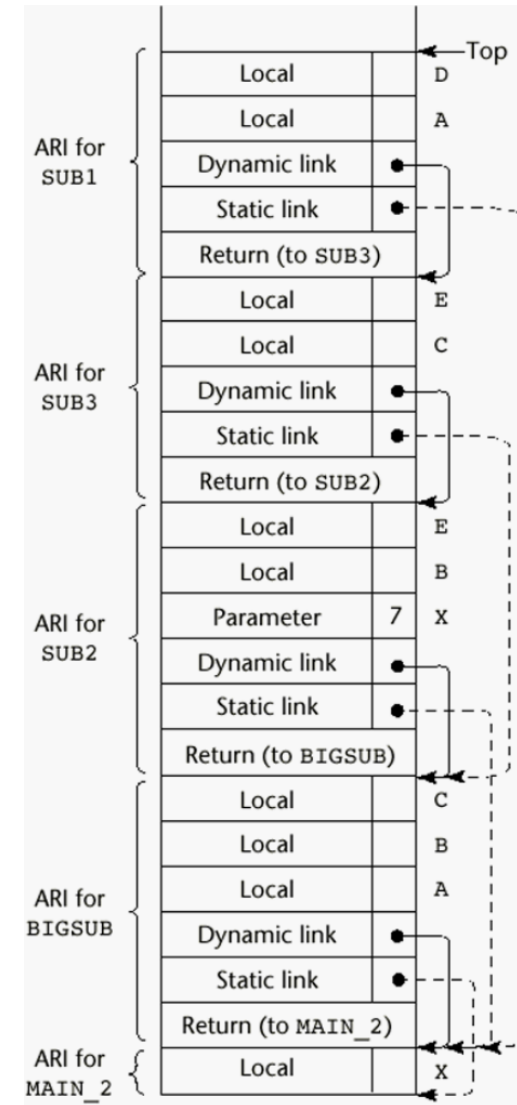
# Example Pascal Program (continued)

---

- Call sequence for MAIN\_2
  - MAIN\_2 calls BIGSUB
  - BIGSUB calls SUB2
  - SUB2 calls SUB3
  - SUB3 calls SUB1



# Stack Contents at Position 1



# 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

# 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 temp is that it cannot interfere with any other variable with the same name

# Implementing Blocks

---

## Two Methods:

1. 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
2. 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
- *Shallow Access*: put locals in a central place
  - One stack for each variable name
  - Central table with an entry for each variable name

# Using Shallow Access to Implement Dynamic Scoping

---

	A			B
	A	C		A
MAIN_6	MAIN_6	B	C	A
u	v	x	z	w

(The names in the stack cells indicate the program units of the variable declaration.)

# Summary

---

- Subprogram linkage semantics requires many actions 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



Next Time:  
Compilers

---