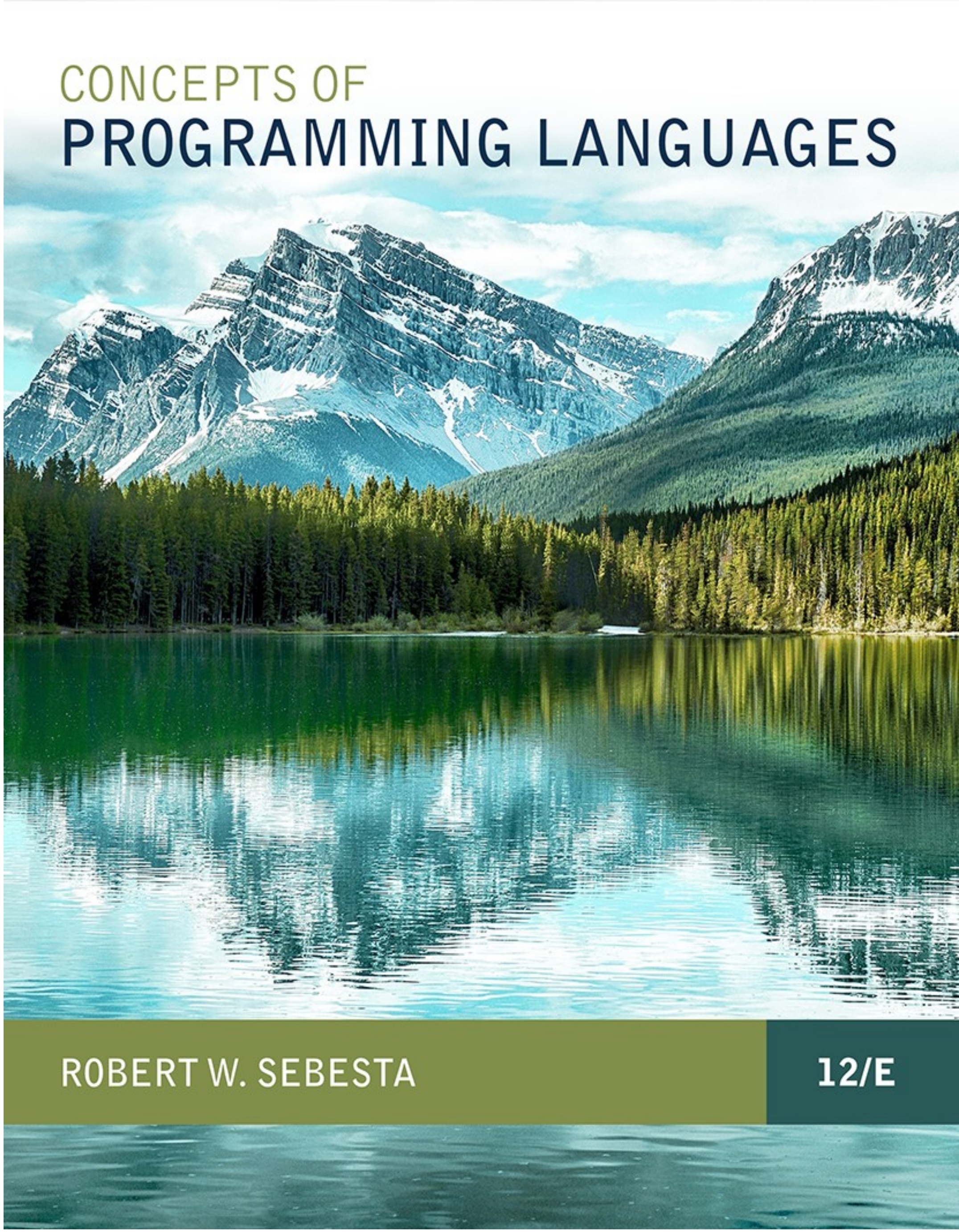
Chapter 10

Implementing Subprograms



ISBN 0- 0-321-49362-1

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

Copyright © 2018 Pearson. All rights reserved.

The General Semantics of Calls and Returns: Calls

- General semantics of calls to a subprogram
  - Parameter passing
  - 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

Copyright © 2018 Pearson. All rights reserved.

The General Semantics of Calls and Returns: Returns

- General semantics of subprogram returns:
  - Out mode and inout mode parameters must have their values returned
  - Restore the execution status
  - Return control to the caller

Copyright © 2018 Pearson. All rights reserved.

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

Copyright © 2018 Pearson. All rights reserved.

Implementing “Simple” Subprograms

- Call Semantics:
  - Save the execution status of the caller
  - Pass the parameters
  - Save the return address
  - Transfer control to the callee

Copyright © 2018 Pearson. All rights reserved.

Implementing "Simple" Subprograms

- 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

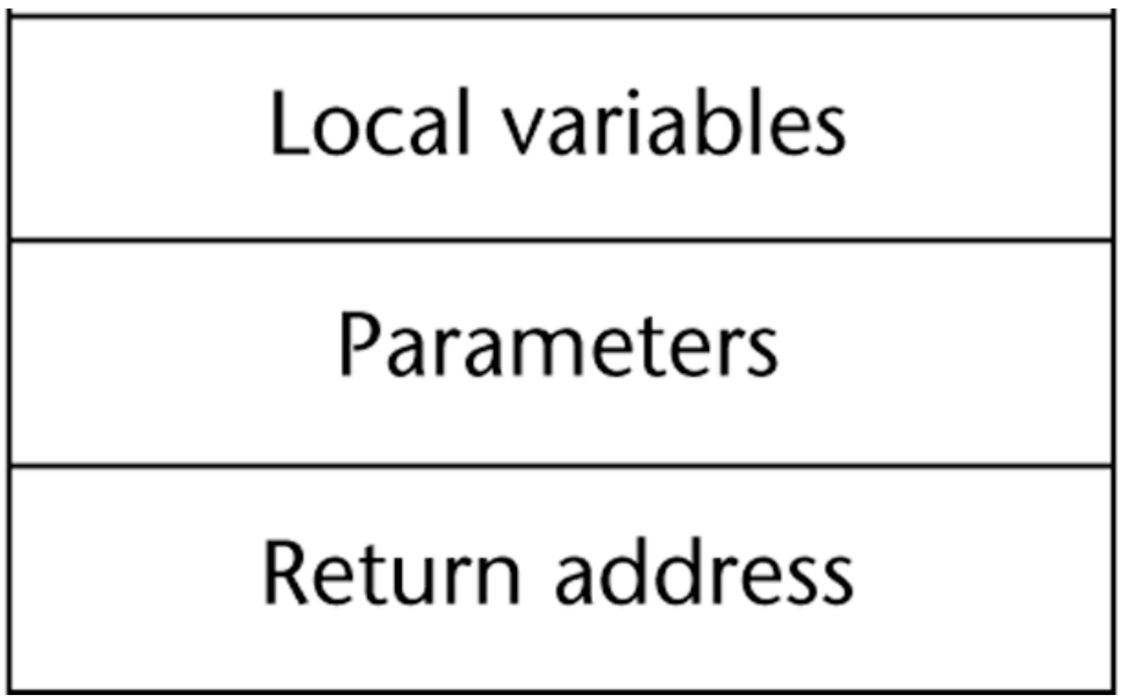
Copyright © 2018 Pearson. All rights reserved.

Implementing "Simple" Subprograms : Parts

- Two separate parts:
  - actual code. (i.e., instructions)
  - non-code part (local variables and input parameters that can be changed)
- *Activation record:*
  - the format, or layout, of the non-code part of an executing subprogram
- *Activation record instance:*
  - a concrete example of an activation record -- the collection of data for a particular subprogram activation

Copyright © 2018 Pearson. All rights reserved.

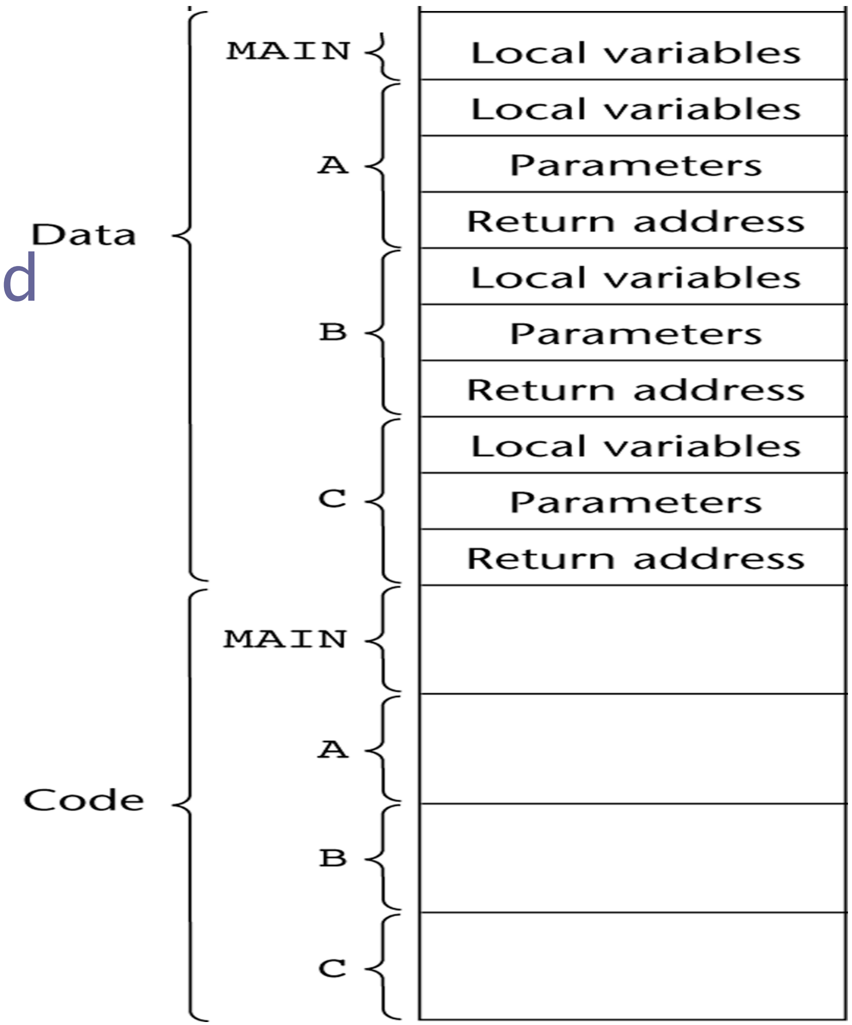
An Activation Record for “Simple” Subprograms



Copyright © 2018 Pearson. All rights reserved.

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

- Code and Activation Records of a Program with "Simple" Subprograms—A, B, and C
- A calls B; returned address in B's ARI tells where to resume A's next instruction after B returns.



Copyright © 2018 Pearson. All rights reserved.

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

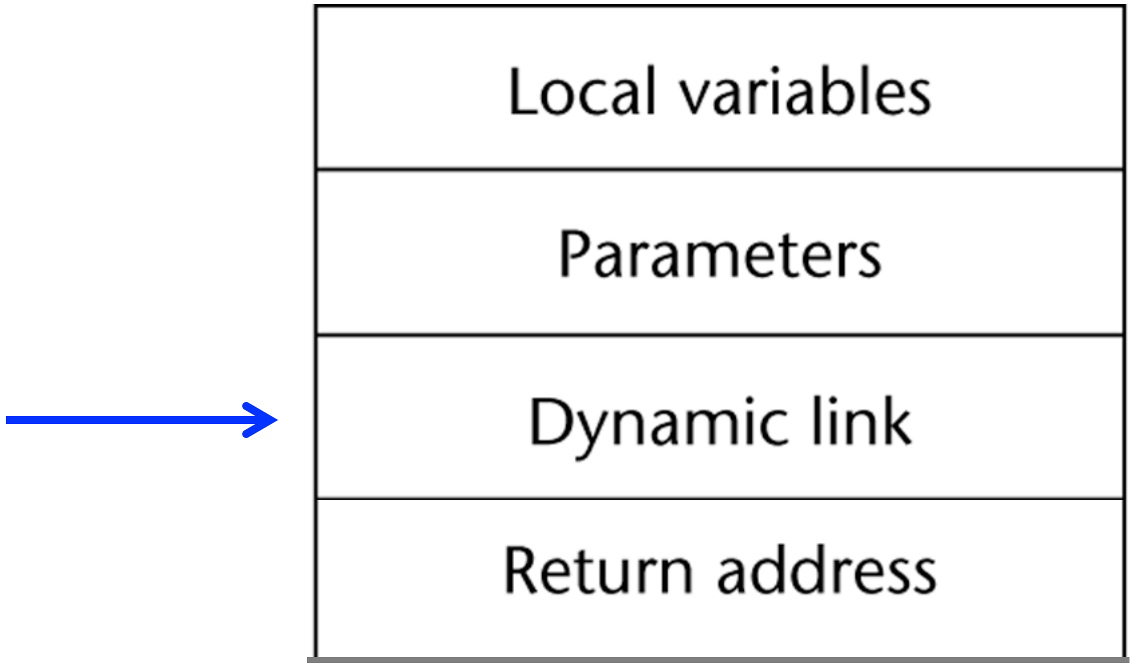
Copyright © 2018 Pearson. All rights reserved.

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)

Copyright © 2018 Pearson. All rights reserved.

Typical Activation Record with Stack-Dynamic Local Variables



Copyright © 2018 Pearson. All rights reserved.

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

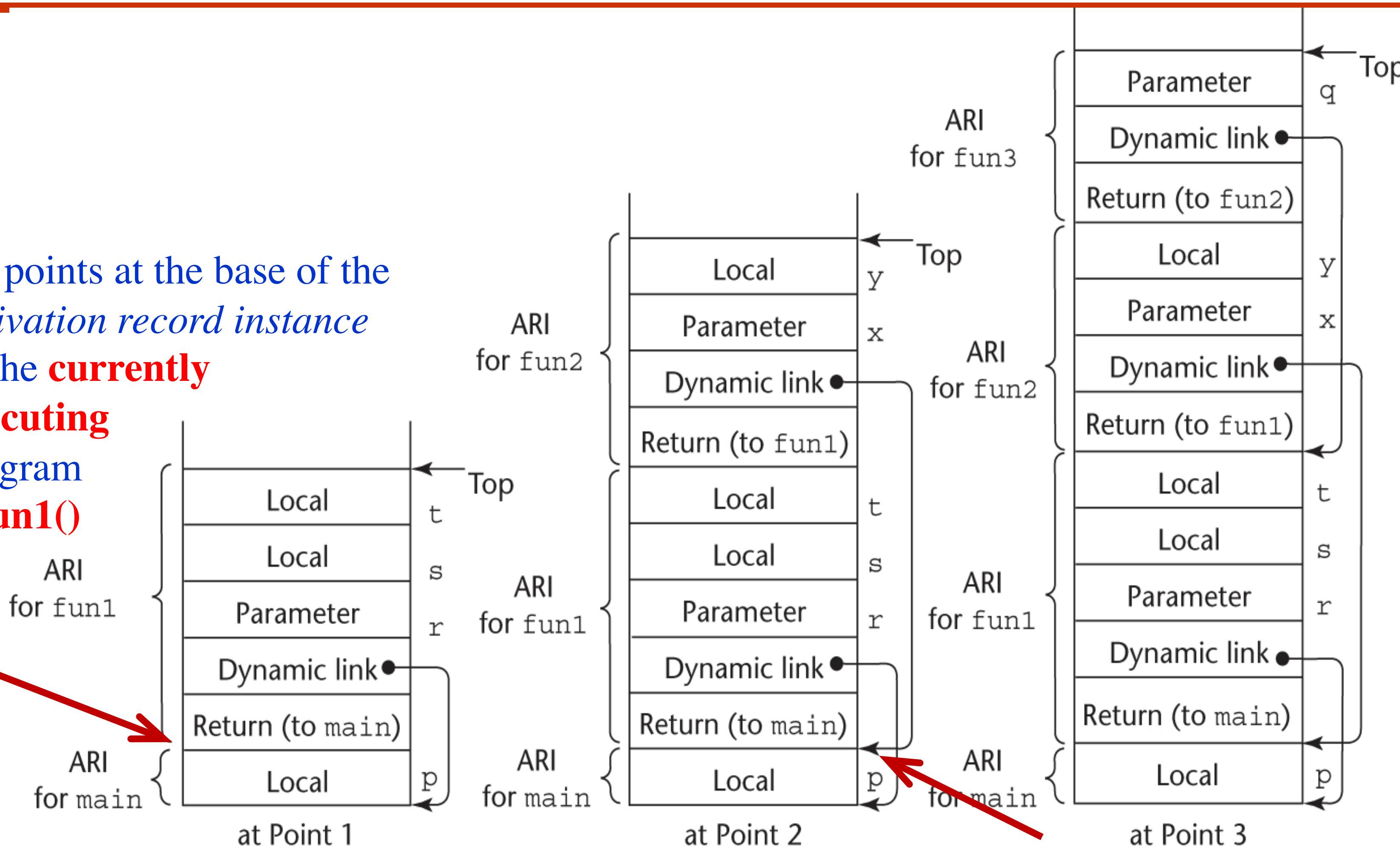
- The activation record format is static, but its size may be dynamic
- The *dynamic link*
  - points to the **base** of an instance of the activation record of the caller
- An activation record instance
  - is dynamically created when a subprogram is called
  - reside on the run-time memory stack
- 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

Copyright © 2018 Pearson. All rights reserved.

An Example Without Recursion

EP points at the base of the *activation record instance* of the **currently executing** program – **fun1()**

EP



ARI = activation record instance

Dynamic link in ARI for **fun2** points to the **base** of ARI for **fun1--**caller

Copyright © 2018 Pearson. All rights reserved.

Revised Semantic Call/Return Actions

- Caller Actions:
  - Create an activation record instance
  - Save the execution status of the current program unit
  - Compute and pass the parameters
  - Save the return address
  - Transfer control to the callee

Copyright © 2018 Pearson. All rights reserved.

Revised Semantic Call/Return Actions

- Callee Actions:
  - Save the old EP in the stack as the dynamic link, create the new value
  - Allocate local variables
  - Prepare returned values:
    - If pass-by-value-result or out-mode parameters, move current values of parameters to corresponding actual parameters
    - If subprogram is a function, move its value to a place accessible to the caller
  - Restore the EP by setting EP to the old dynamic link
  - Restore the execution status of the caller
  - Transfer control back to the caller

Copyright © 2018 Pearson. All rights reserved.

An Example Without Recursion

```
void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}
void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}
void fun3(int q) {
    ...
}
void main() {
    float p;
    ...
    fun1(p);
    ...
}
```

Call sequences:

`main()-> fun1(p)-> fun2(s)-> fun3(y)`

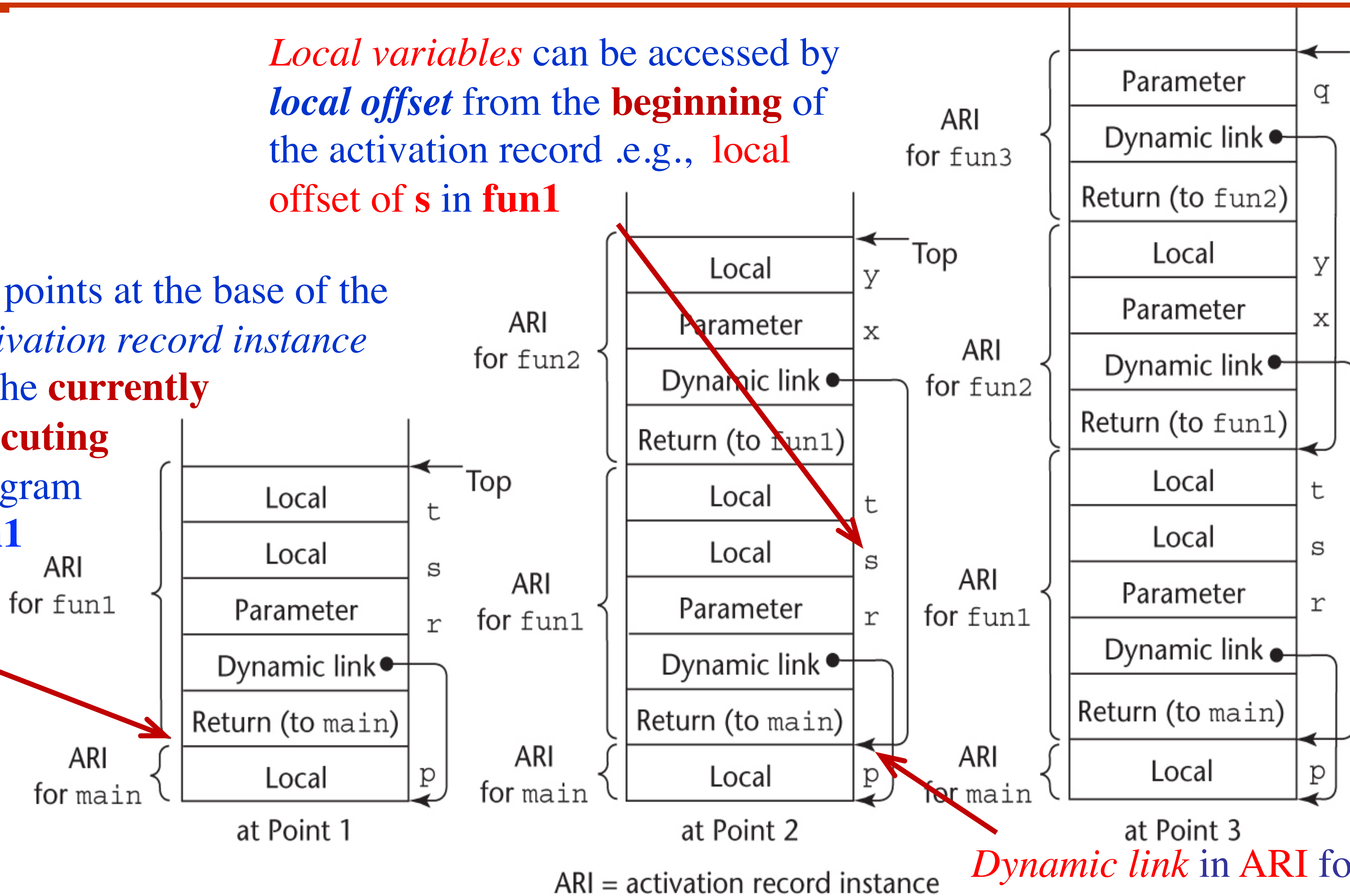
Copyright © 2018 Pearson. All rights reserved.

An Example Without Recursion

Local variables can be accessed by ***local offset*** from the **beginning** of the activation record .e.g., local offset of **s** in **fun1**

EP points at the base of the *activation record instance* of the **currently executing** program **fun1**

EP



Dynamic link in ARI for **fun2** points to the base of ARI for **fun1**--caller

Copyright © 2018 Pearson. All rights reserved.

Dynamic Chain and Local Offset

- *Dynamic chain*
  - the collection of dynamic links in the stack at a given time
- Local_Offset
  - can be determined by the compiler at compile time
  - use to access **local variables**, from the beginning of ARI

Copyright © 2018 Pearson. All rights reserved.

An Example With Recursion

- The activation record used in the previous example supports recursion

```
int factorial (int n) {

  if (n <= 1) return 1;
  else return (n * factorial(n - 1));

}
void main() {
  int value;
  value = factorial(3);

}
```

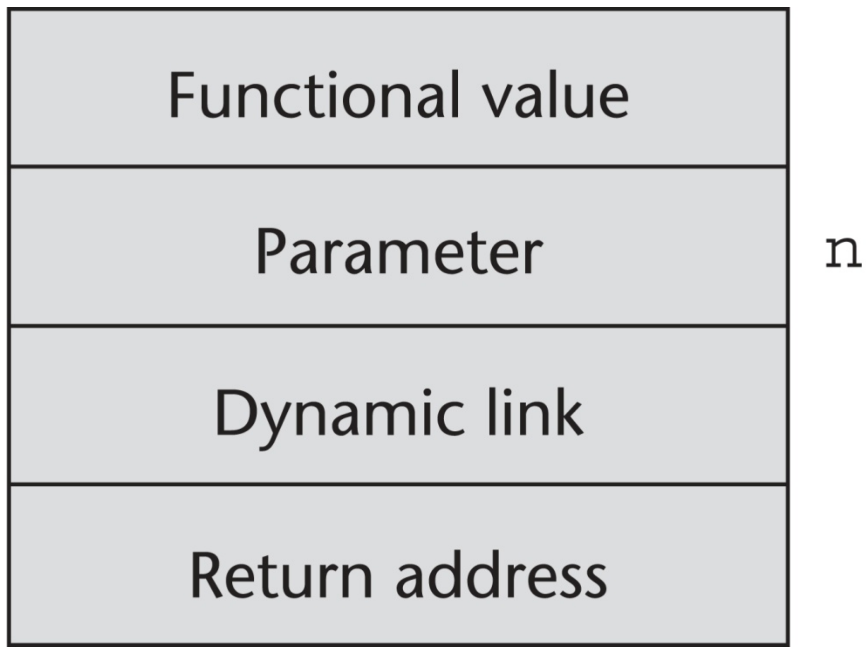
Factorial (3) → 3 * factorial (2)

→ 3 * (2 * factorial (1))

→ 3 * (2 * (1))

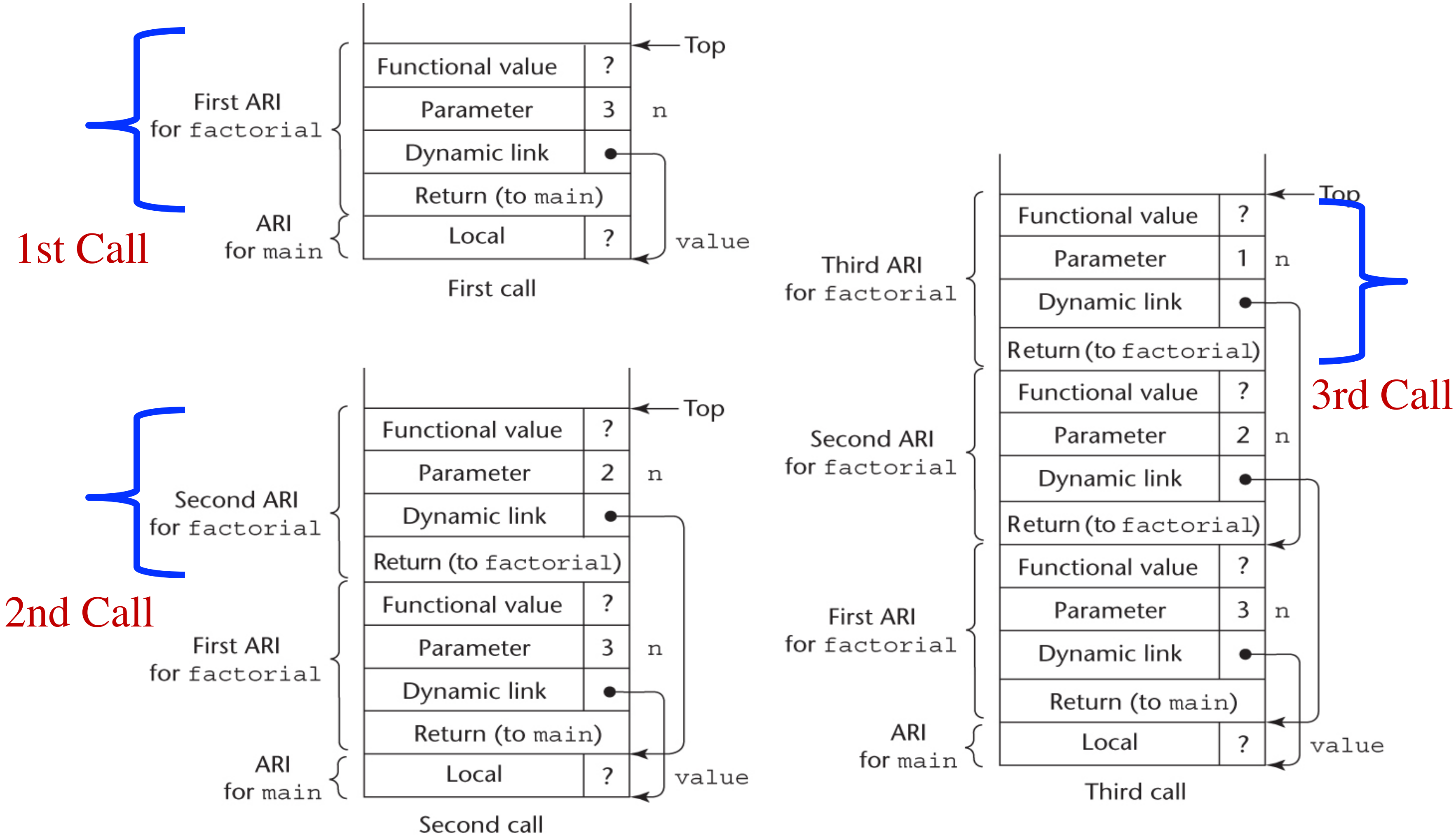
Copyright © 2018 Pearson. All rights reserved.

Activation Record for `factorial`



Copyright © 2018 Pearson. All rights reserved.

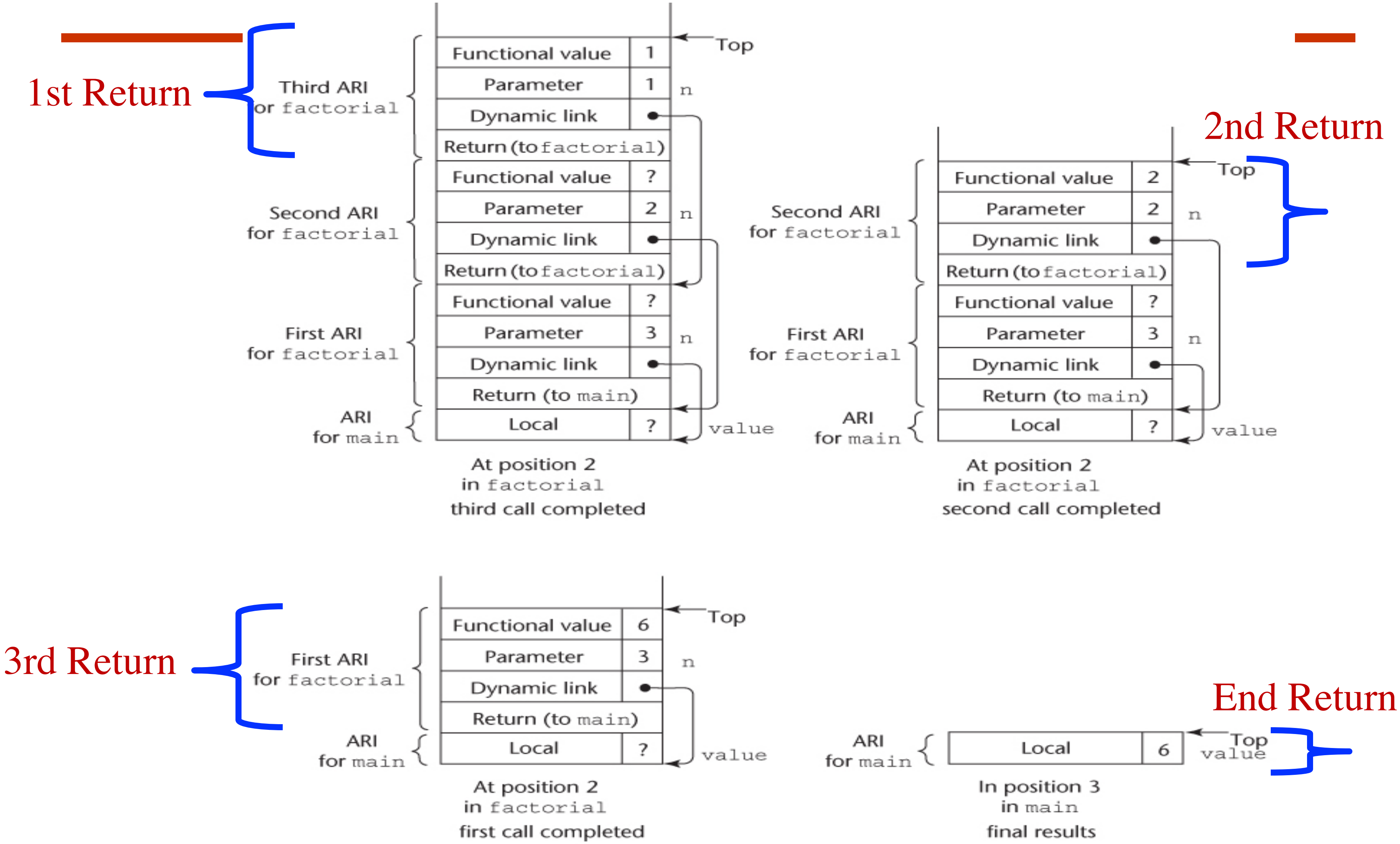
Stacks for calls to `factorial`



ARI = activation record instance

Copyright © 2018 Pearson. All rights reserved.

Stacks for returns from `factorial`



ARI = activation record instance

Copyright © 2018 Pearson. All rights reserved.

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

Copyright © 2018 Pearson. All rights reserved.

Nested Subprograms

- Some non-C-based static-scoped languages
  - use stack-dynamic local variables
  - allow subprograms to be nested
  - e.g., Fortran 95+, Ada, Python, JavaScript, Ruby, and Lua
- All **variables** reside in some ARI in the stack
- The process of locating a **non-local** reference:
  1. Find the correct ARI
  2. Determine the correct offset within that ARI

Copyright © 2018 Pearson. All rights reserved.

Static Scoping

- *Static link* in subprogram A's ARI
  - points to the bottom of ARI of A's static parent.
    - E.g., on slide #30: bigsub() is the static parent of sub1() and sub2()
  - use to access to non-local variables
  - *static chain:* a chain of static links connects certain ARIs
- *Static depth*
  - a static scope whose value--an integer, is the depth of nesting of that scope
  - static depth = 0 if a program is **not** nested inside any other unit
  - static depth = 1 if a subprogram is defined in a non-nested program

Copyright © 2018 Pearson. All rights reserved.

Static Scoping

- 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.
  - E.g., static depth of f1, f2, f3 are 0, 1, 2, respectively.
  - If f3 refers a variable defined in f1
    chain offset = 2 (= 2– 0)

```
def f1 ():
    def  f2() :
        def f3() :

        #end of f3
    #enf of f2
#end of f1
```

- A reference to a variable can be represented by the pair: (chain_offset, local_offset)
  - use chain_offset to find the right ARI
  - use local_offset to find the local variable in the ARI identified in step #1.

Copyright © 2018 Pearson. All rights reserved.

Example JavaScript Program

```
function main () {
    var x;
    function bigsub() {
        var a, b, c;
        function sub1() {
            var a, d;   a = b + c;
        } // end of sub1()
        function sub2(x) {
            var b, e;
            function sub3() {
                var c, e;
                sub1(); // call sub1()
                e = a;  // access a defined in bigsub()
            }  // end of sub3()
            sub3();  // call sub3()
        }  // end sub2()
        sub2(7);  // call sub2()
    }  // end of bigsub()
    bigsub();  // calling bigsub()
}  // end of main()
```

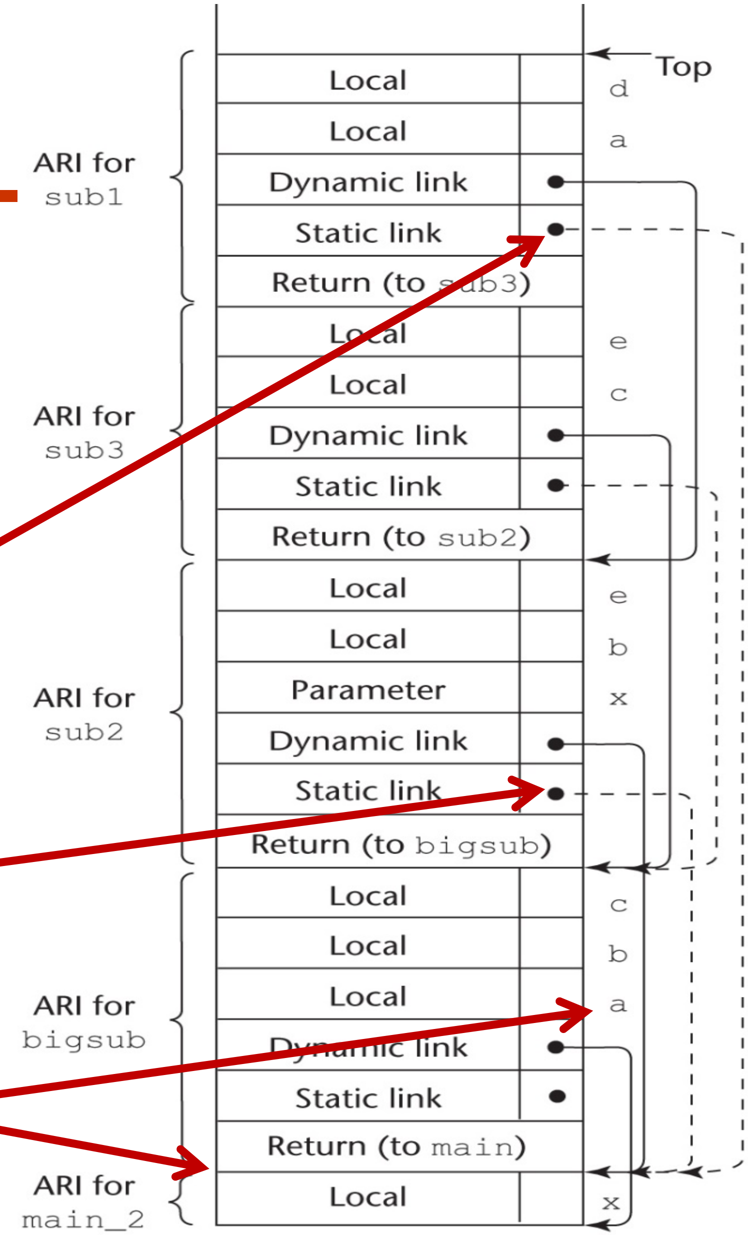
- Call sequence for `main`
  - `main` calls `bigsub`
  - `bigsub` calls `sub2`
  - `sub2` calls `sub3`
  - `sub3` calls `sub1`

Copyright © 2018 Pearson. All rights reserved.

Example JavaScript Program

- Call sequence for `MAIN`

  `MAIN` **calls** `BIGSUB`
  `BIGSUB` **calls** `SUB2`
  `SUB2` **calls** `SUB3`
  `SUB3` **calls** `SUB1`



Static link of **sub1**() and **sub2**() points to the ARI of its static **parent** program **bigsub**()

sub3() needs to access variable **a** defined in **bigsub**(): chain_offset = 2

ARI = activation record instance

Copyright © 2018 Pearson. All rights reserved.

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

Copyright © 2018 Pearson. All rights reserved.

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

Copyright © 2018 Pearson. All rights reserved.

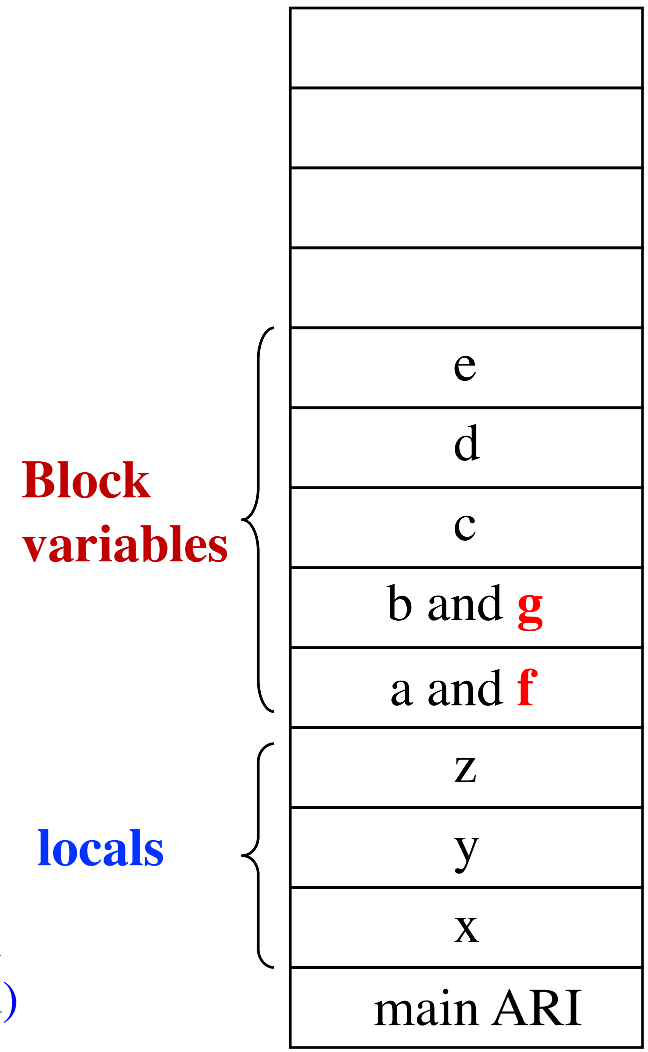
Implementing Blocks

- 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

Copyright © 2018 Pearson. All rights reserved.

Implementing Blocks

```
void main  {
    int x, y, z;
    while ( ..) {
         int a, b, c;
         ….
         while (…) {
              int d, e;
         }
    }
    while ( ..) {
         int f, g;  ….
    }
}
```



Note: *a* and *b* are popped off the stack when their block is exited (before **g** and **f** allocated)

Copyright © 2007 Addison-Wesley. All rights reserved.

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

Copyright © 2018 Pearson. All rights reserved.

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 local variables in a central place
  - One stack for each variable name
  - Central table with an entry for each variable name

Copyright © 2018 Pearson. All rights reserved.

Implement Dynamic Scoping: Deep Access

- Call sequence for `MAIN`

  `main` calls `sub1`

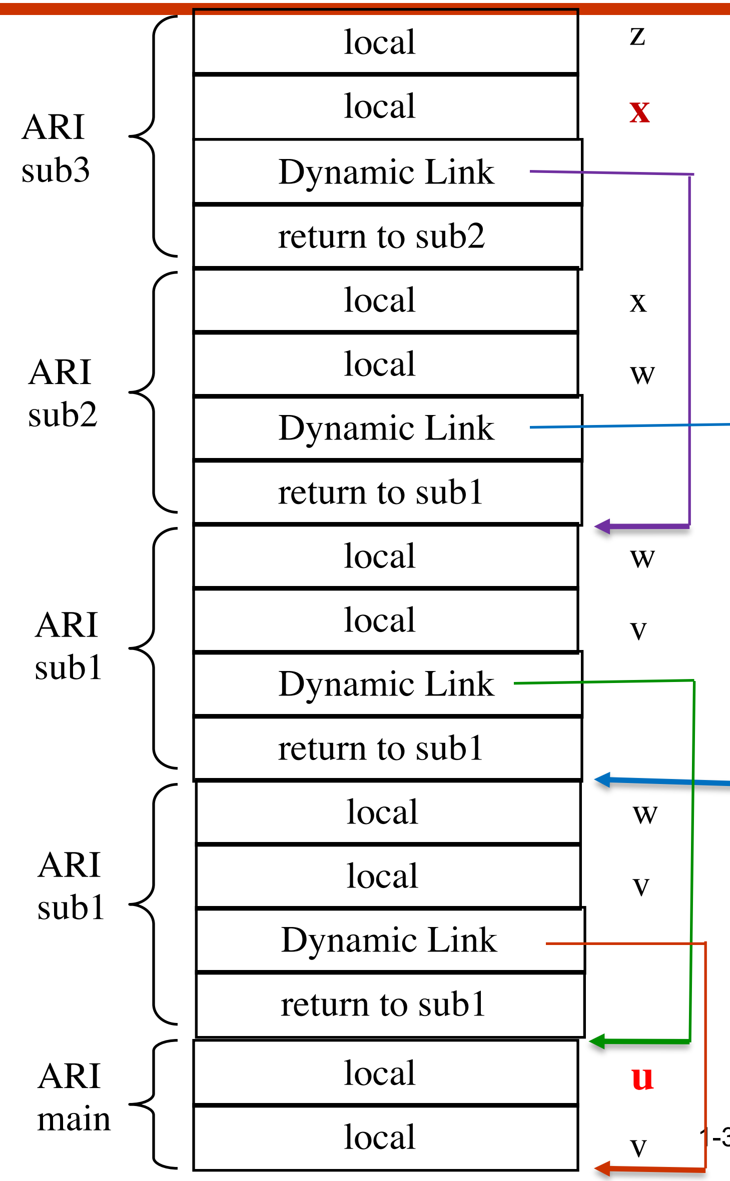
  `sub1` calls `sub1`

  `sub1` calls `sub2`

  `sub2` calls `sub3`

- To find reference **u** in sub3()— search all ARIs to get **u—nonlocal variable,** thru **dynamic links**

```
void sub3() {
   int x, z;
   x = u + v;
   …
}
void sub2() {
   int w, x;
   …
}
void sub1() {
   int v, w;
   …
}
void main() {
   int v, u;
   …
}
```



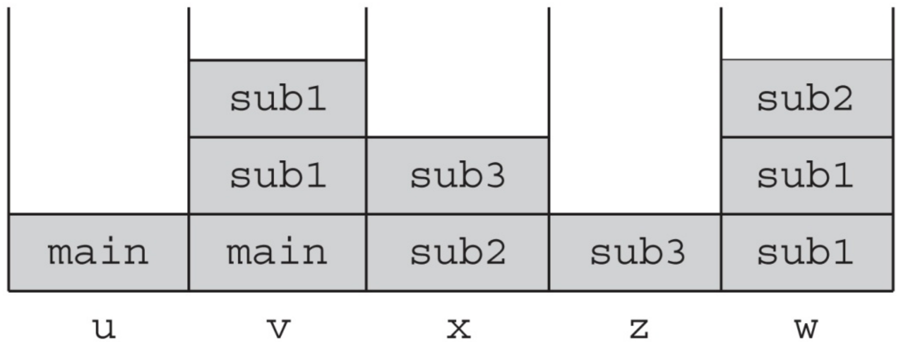
Copyright © 2007 Addison-Wesley. All rights reserved.

Implement Dynamic Scoping: Shallow Access

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```

1. **Subprogram name** is added to the **top** of the **stack** of **variables** when **called**.
2. when a **subprogram terminates**, the lifetime of its local variables ends, the sub-program name **is popped from the stack**.
3. **Top of subprogram** is **used** for a given variable.

variables



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

- Call sequence for `main`
  - `main` calls `sub1`
  - `sub1` calls `sub1`
  - `sub1` calls `sub2`
  - `sub2` calls `sub3`

Copyright © 2018 Pearson. All rights reserved.