

CONCEPTS OF
**Programming
Languages**



AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Chapter 10

Implementing
Subprograms

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*
- General semantics of calls to a subprogram
 - 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:
 - Save the execution status of the caller
 - Pass the parameters
 - Pass the return address to the called
 - Transfer control to the called

Implementing “Simple” Subprograms (continued)

- 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, temporaries

Implementing “Simple” Subprograms (continued)

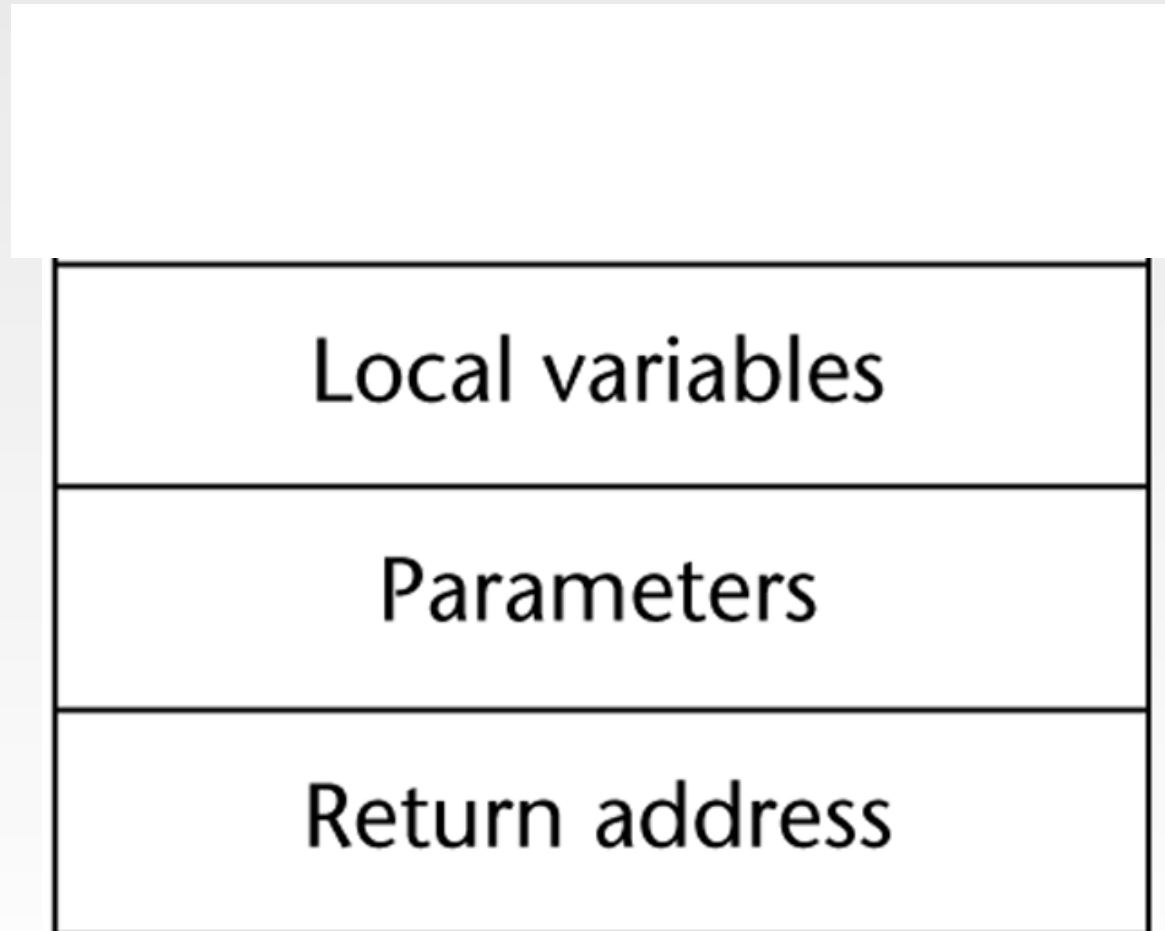
- Two separate parts: the actual code and the non-code part (local variables and data that can change)
- The format, or layout, of the non-code part of an executing subprogram is called an *activation record*
- An *activation record instance* is a concrete example of an activation record (the collection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



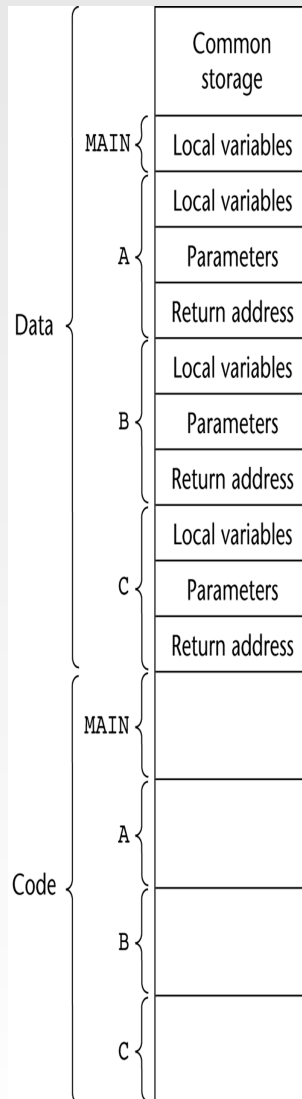
AUBURN
UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

An Activation Record for “Simple” Subprograms



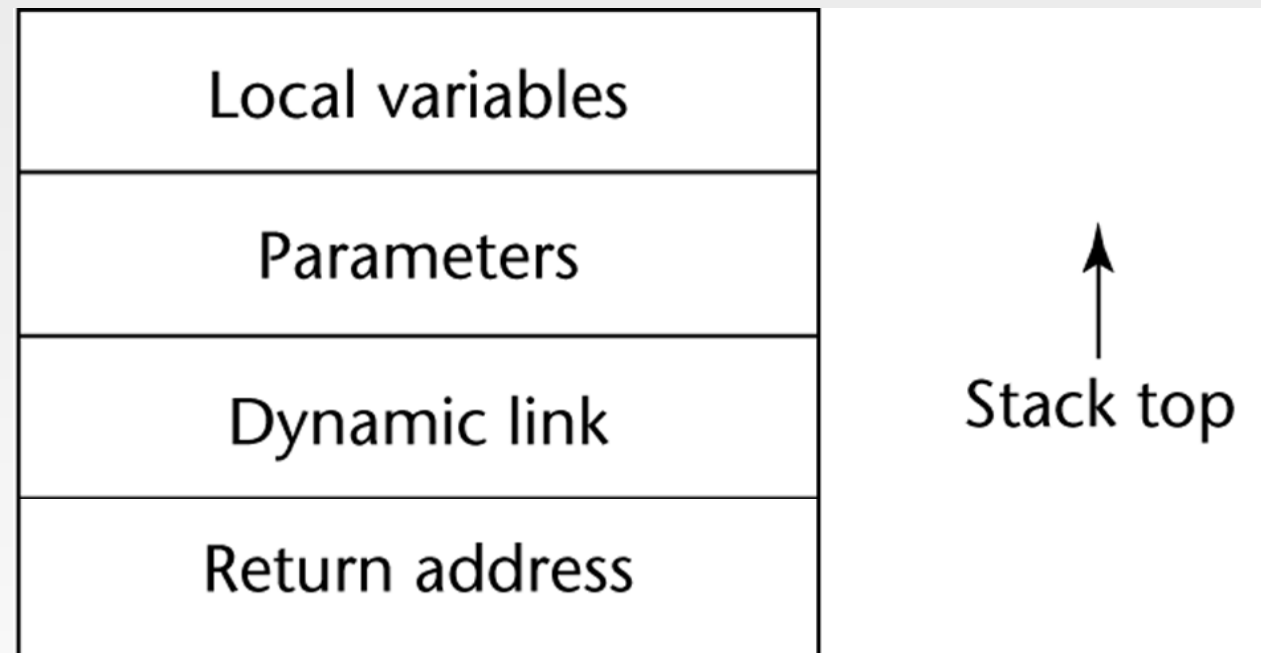
Code and Activation Records of a Program with “Simple” Subprograms



Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
 - The compiler must generate code to cause implicit allocation and 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;
    ...
}
```

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

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
 - Pass the return address to the called
 - Transfer control to the called
- **Prologue actions of the called:**
 - Save the old EP in the stack as the dynamic link and create the new value
 - Allocate local variables

Revised Semantic Call/Return Actions (continued)

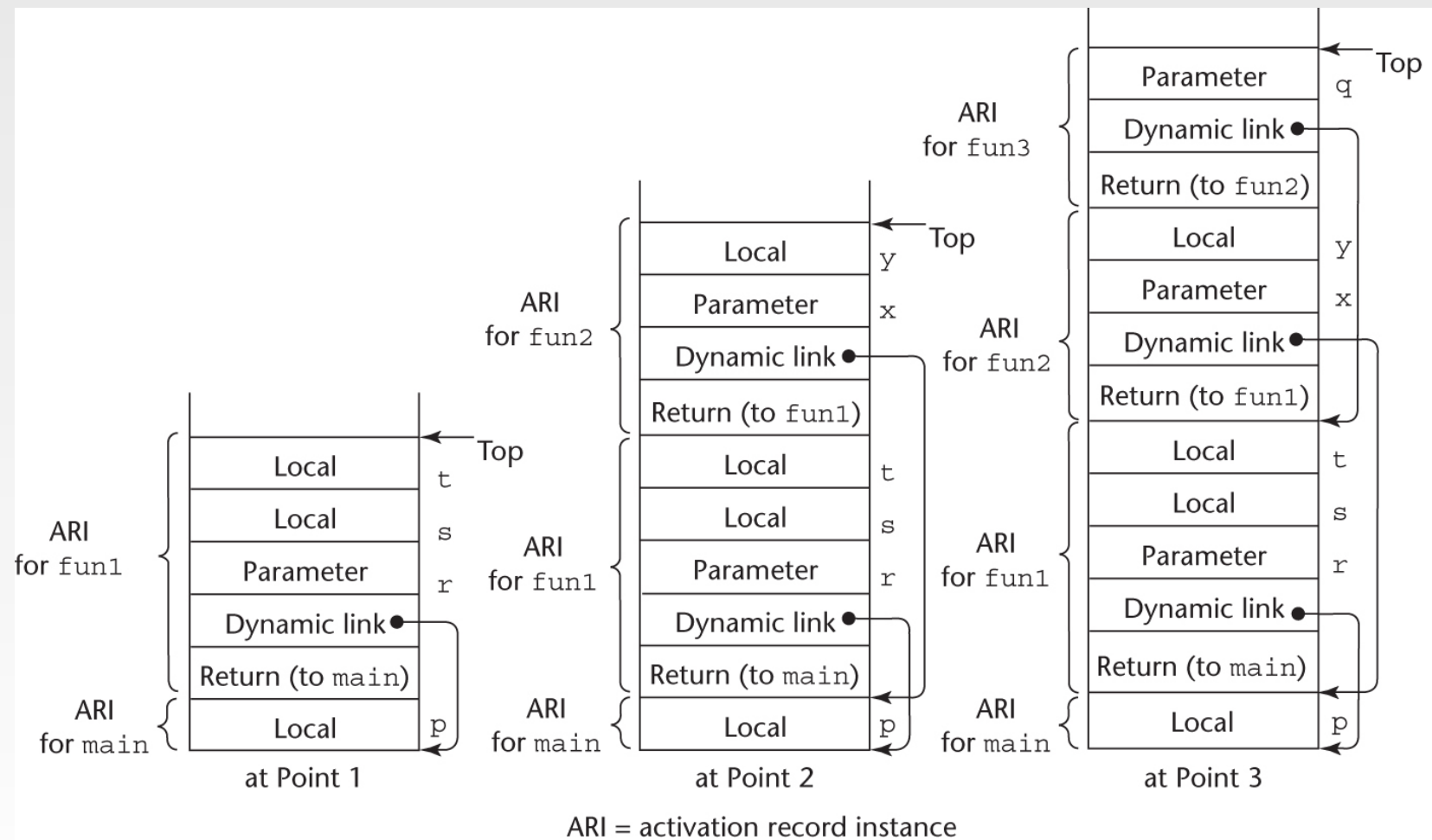
- Epilogue actions of the called:
 - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
 - If the subprogram is a function, its value is moved to a place accessible to the caller
 - Restore the stack pointer by setting it to the value of the current EP-1 and set the EP to the old dynamic link
 - Restore the execution status of the caller
 - Transfer control back to the caller

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);  
    ...  
}
```

main calls fun1
fun1 calls fun2
fun2 calls fun3

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

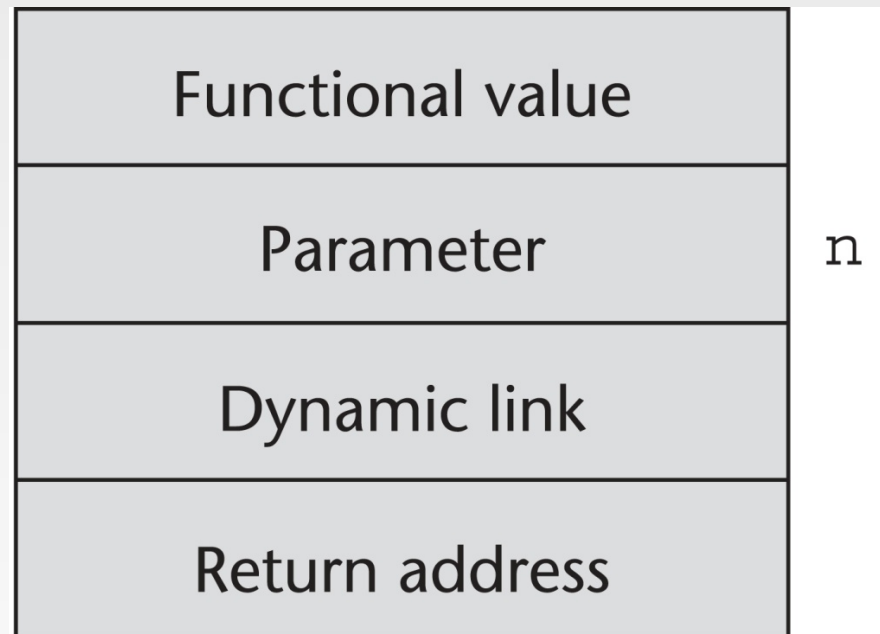


An Example With Recursion

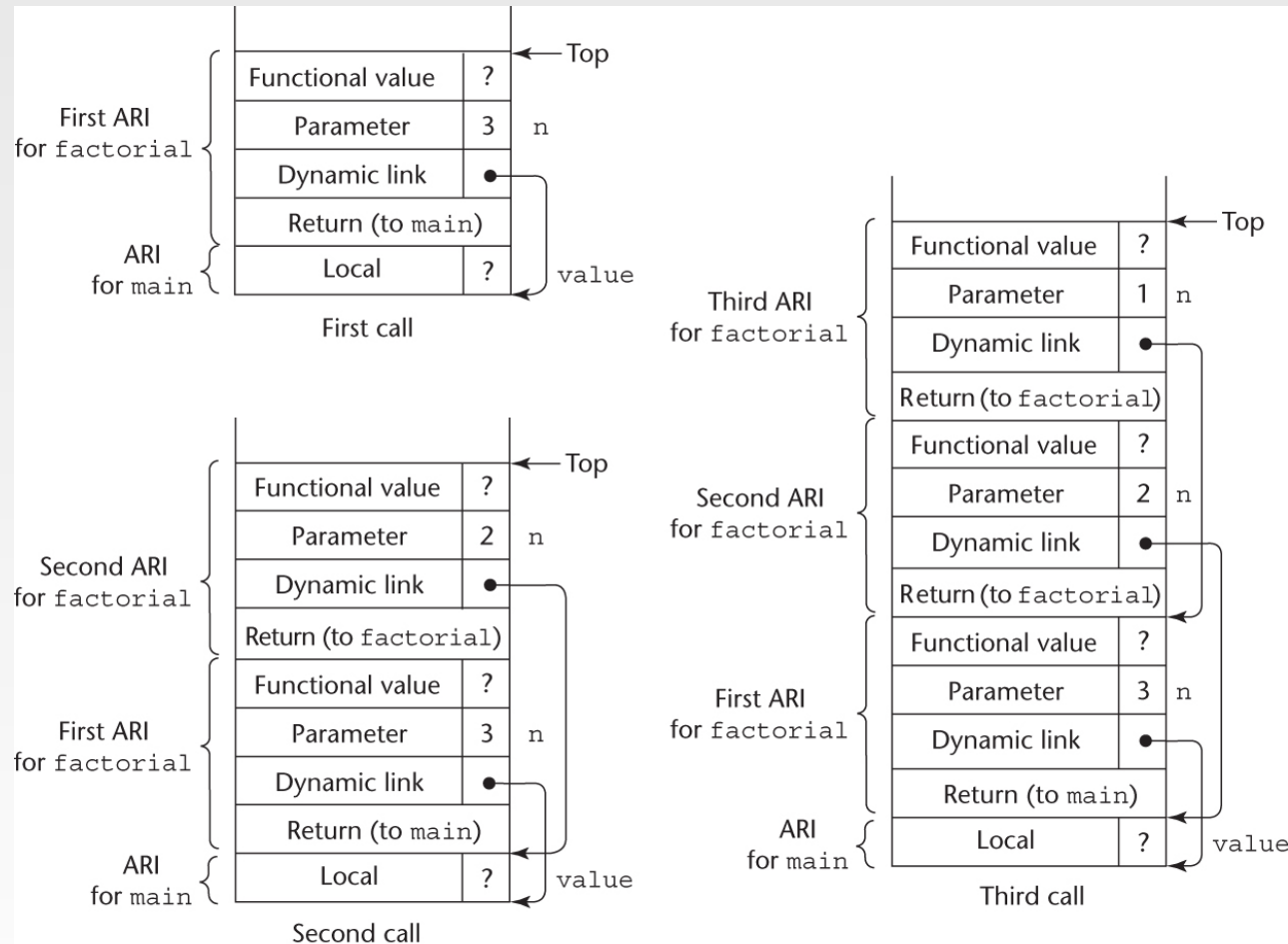
- The activation record used in the previous example supports recursion

```
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`

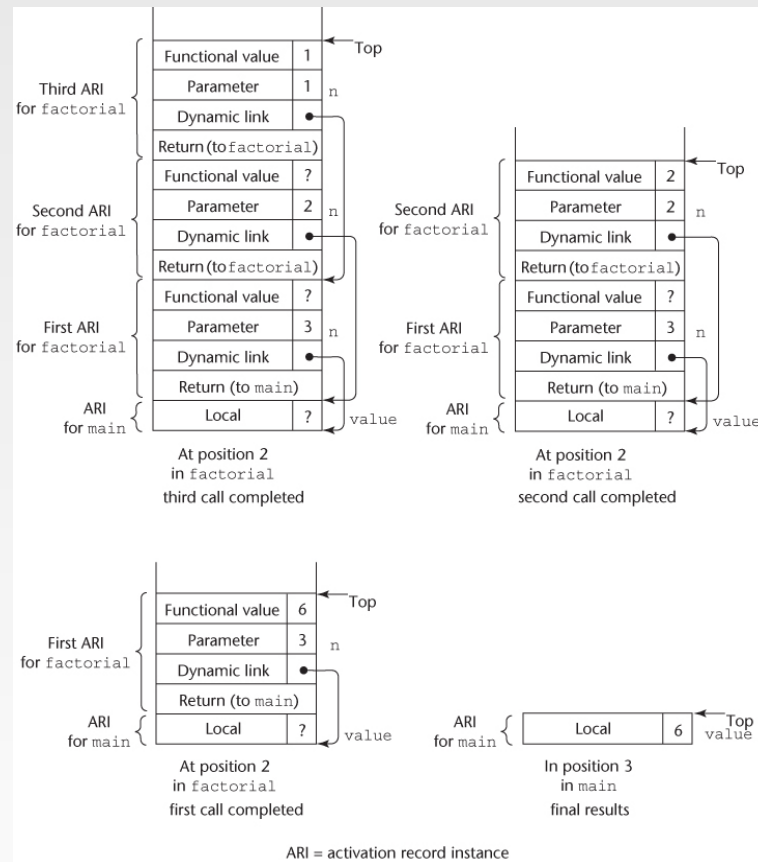


Stacks for calls to `factorial`



ARI = activation record instance

Stacks for returns from 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:
 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_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

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
        A := B + C; <-----1
      end; -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
          Sub1;
          E := B + A; <-----2
        end; -- of Sub3
      begin -- of Sub2
        Sub3;
        A := D + E; <-----3
      end; -- of Sub2 }
    begin -- of Bigsub
      Sub2(7);
    end; -- of Bigsub
  begin
    Bigsub;
  end; of Main_2 }
```



Example Ada Program (continued)

- Call sequence for `Main_2`

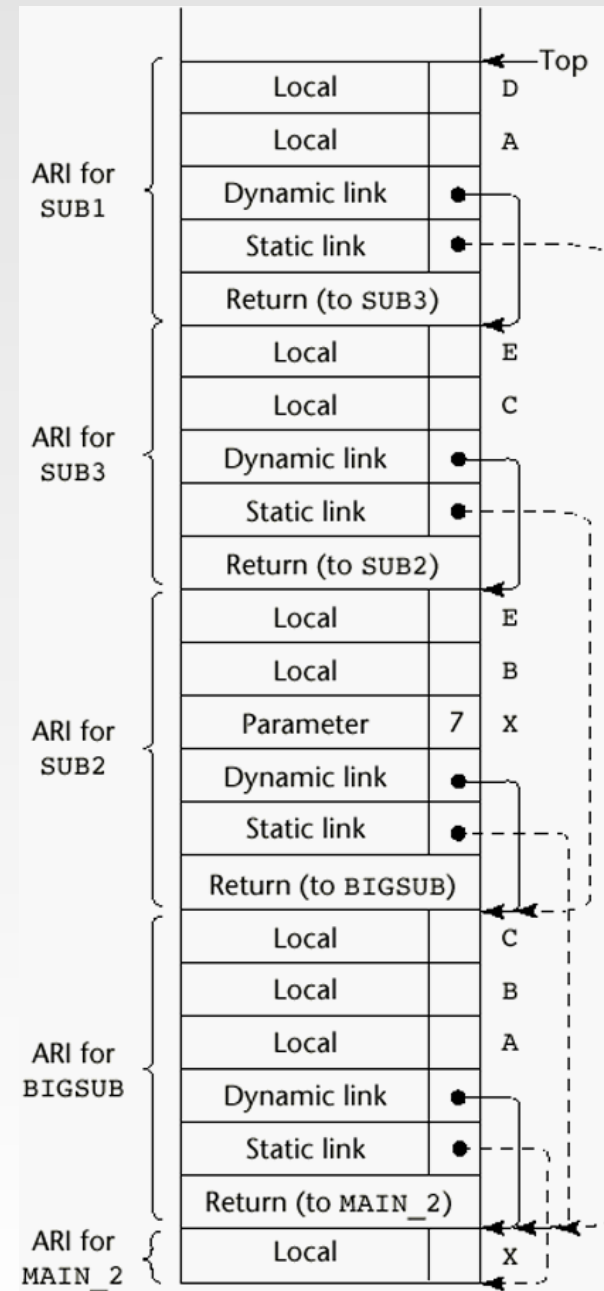
`Main_2` **calls** `Bigsub`

`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:
 1. Search the dynamic chain
 2. Treat subprogram calls and definitions like variable references and definitions

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

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

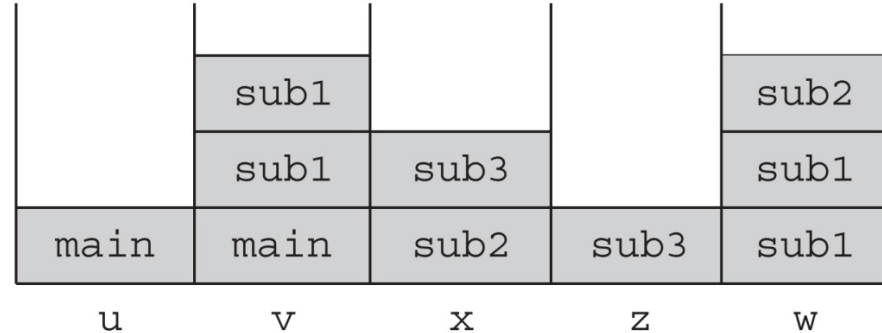
Implementing Dynamic Scoping

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



Using Shallow Access to Implement Dynamic Scoping

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



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

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 table¹⁻³⁷ method



AUBURN
UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING