COSC252: Programming Languages:

Runtime Stack and Scope

Jeremy Bolton, PhD
Asst Teaching Professor

Topics

- The General Semantics of Calls and Returns
- Implementing “Simple” Subprograms (no runtime stack – no recursion)
- 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

GEORGETOWN UNIVERSITY

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

GEORGETOWN UNIVERSITY

Implementing "Simple" Subprograms

- Be simple:
  - NO stack-dynamic variables.

- Call Semantics:

1. Save the execution status of the caller
2. Pass the parameters
3. Pass the return address to the called
4. Transfer control to the called

Implementing "Simple" Subprograms (continued)

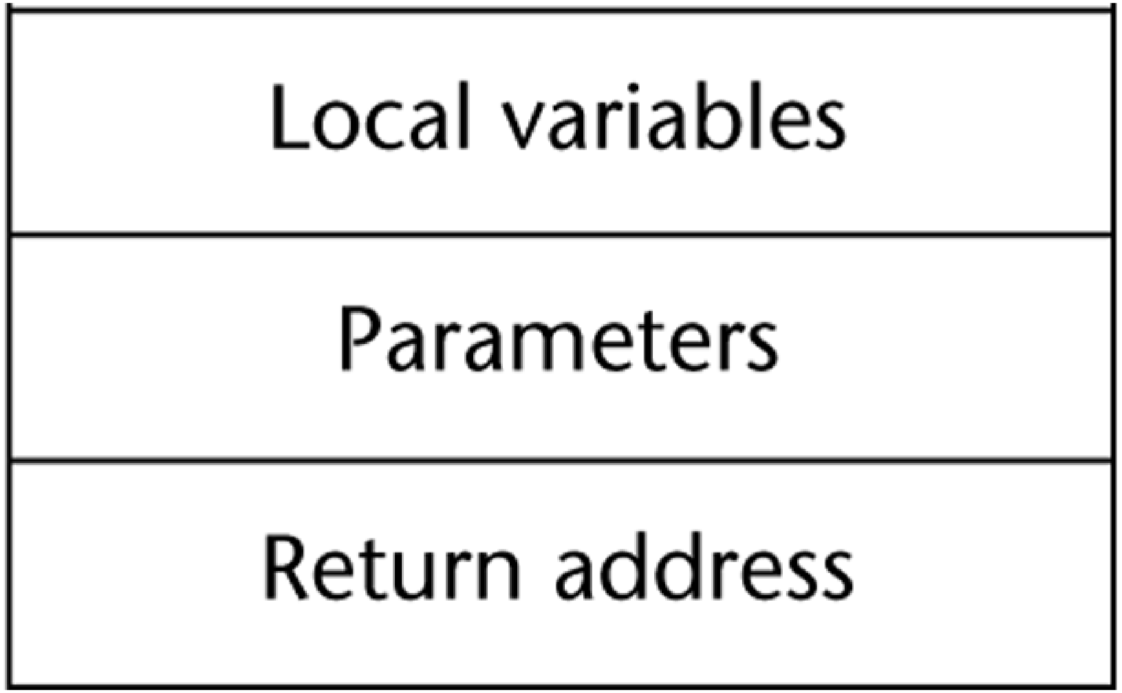
- Return Semantics:
  1. If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
  2. If it is a function, move the functional value to a place the caller can get it
  3. Restore the execution status of the caller
  4. Transfer control back to the caller

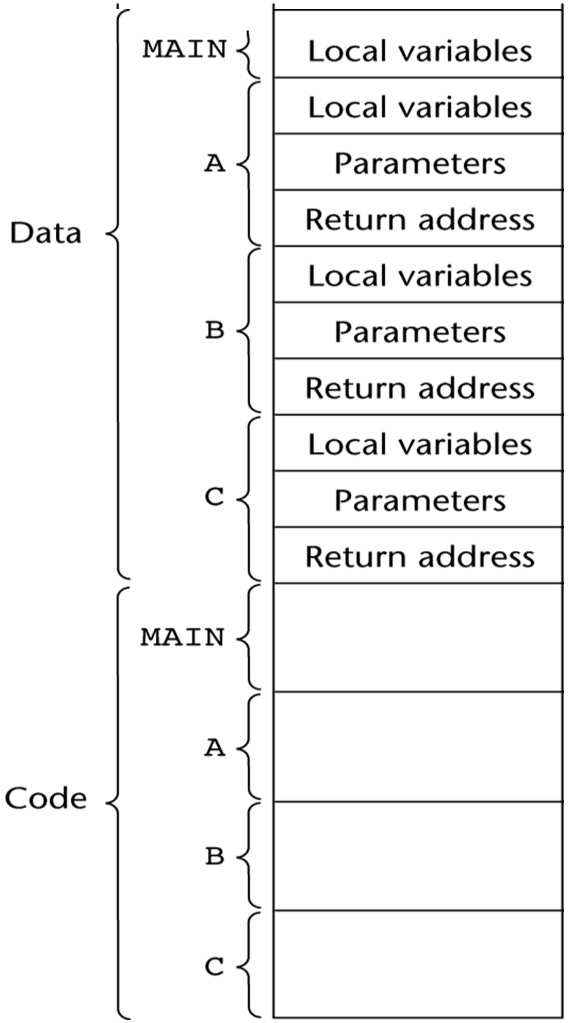
- Required storage:
  - Status information, parameters, return address (return control), return value for functions, temporaries

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)

An Activation Record for "Simple" Subprograms

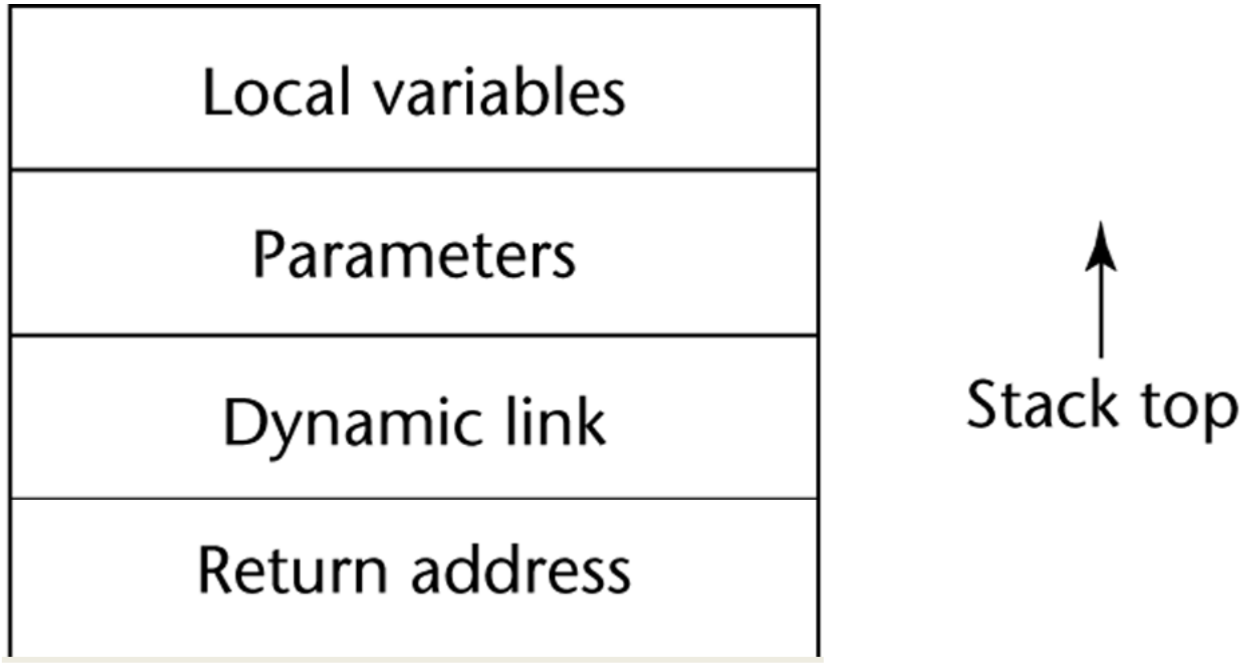




Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
  - The compiler must generate code to cause implicit allocation and deallocation of local variables
  - ***Recursion*** supported (adds the possibility of multiple simultaneous activations of a subprogram)

Typical Activation Record for a Language with Stack-Dynamic Local Variables

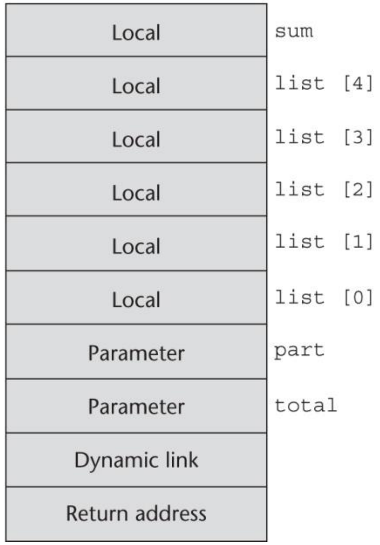


Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- 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;
  …
}
```



Revised Semantic Call/Return Actions

- Caller Actions:
  - A function is invoked: allocate and load new stack frame

1. Create an activation record instance
2. Save the execution status of the current program unit
3. Compute and pass the parameters
4. Pass the return address to the called
5. Transfer control to the called

- Prologue actions of the callee:
  - Load new runtime environment before execution begins

6. Save the old EP in the stack as the dynamic link and create the new value
7. Allocate local variables

Revised Semantic Call/Return Actions (continued)

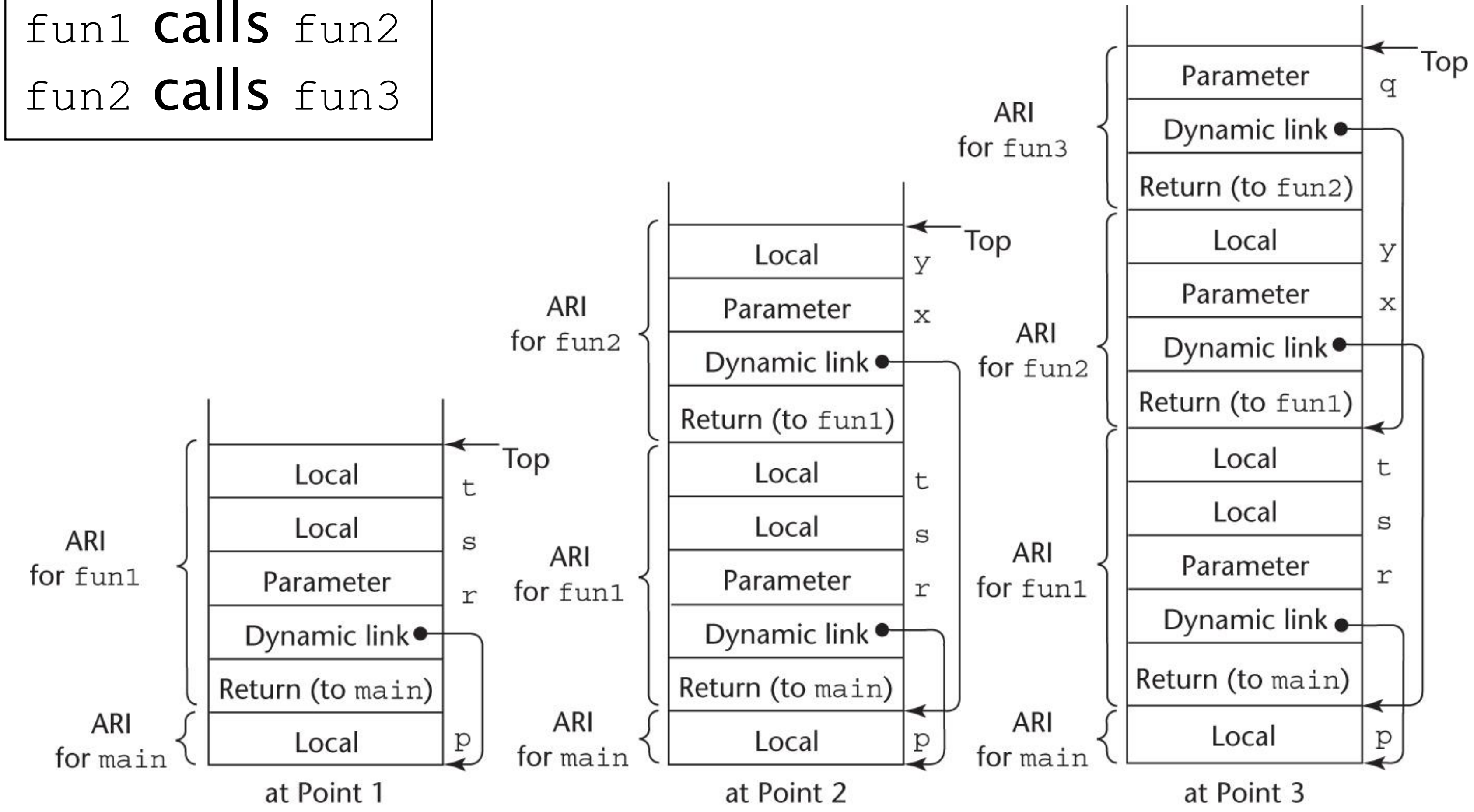
- Epilogue actions of the called :
  - Called function terminates: deallocate stack frame and return control

8. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
9. If the subprogram is a function, its value is moved to a place accessible to the caller
10. Restore the stack pointer by setting it to the value of the current EP-1 and set the EP to the old dynamic link
11. Restore the execution status of the caller
12. 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`



ARI = activation record instance

Dynamic Chain and Local Offset

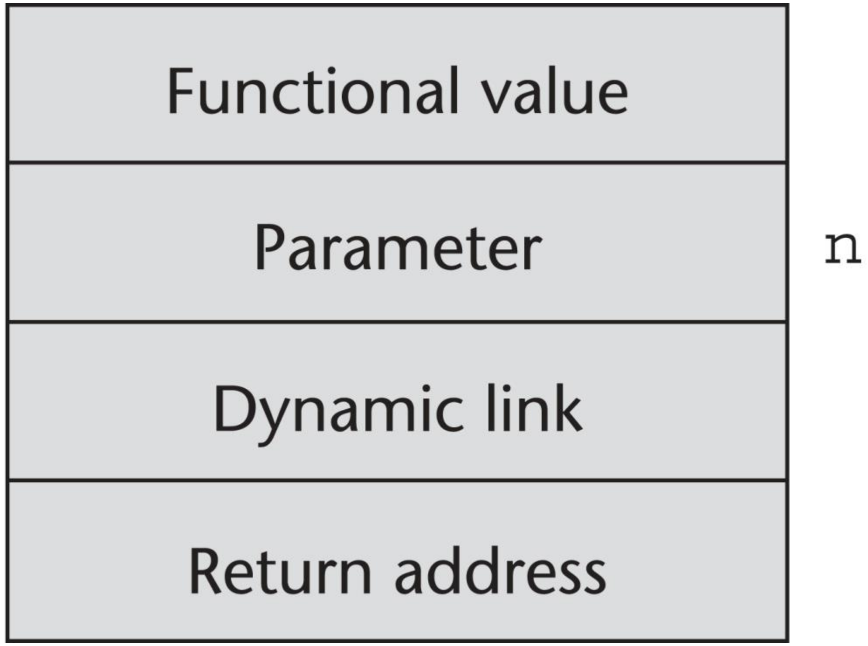
- The collection of dynamic links in the stack at a given time is called the *dynamic chain*, or *call chain*

- Local variables can be accessed by their offset from the beginning of the activation record, 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

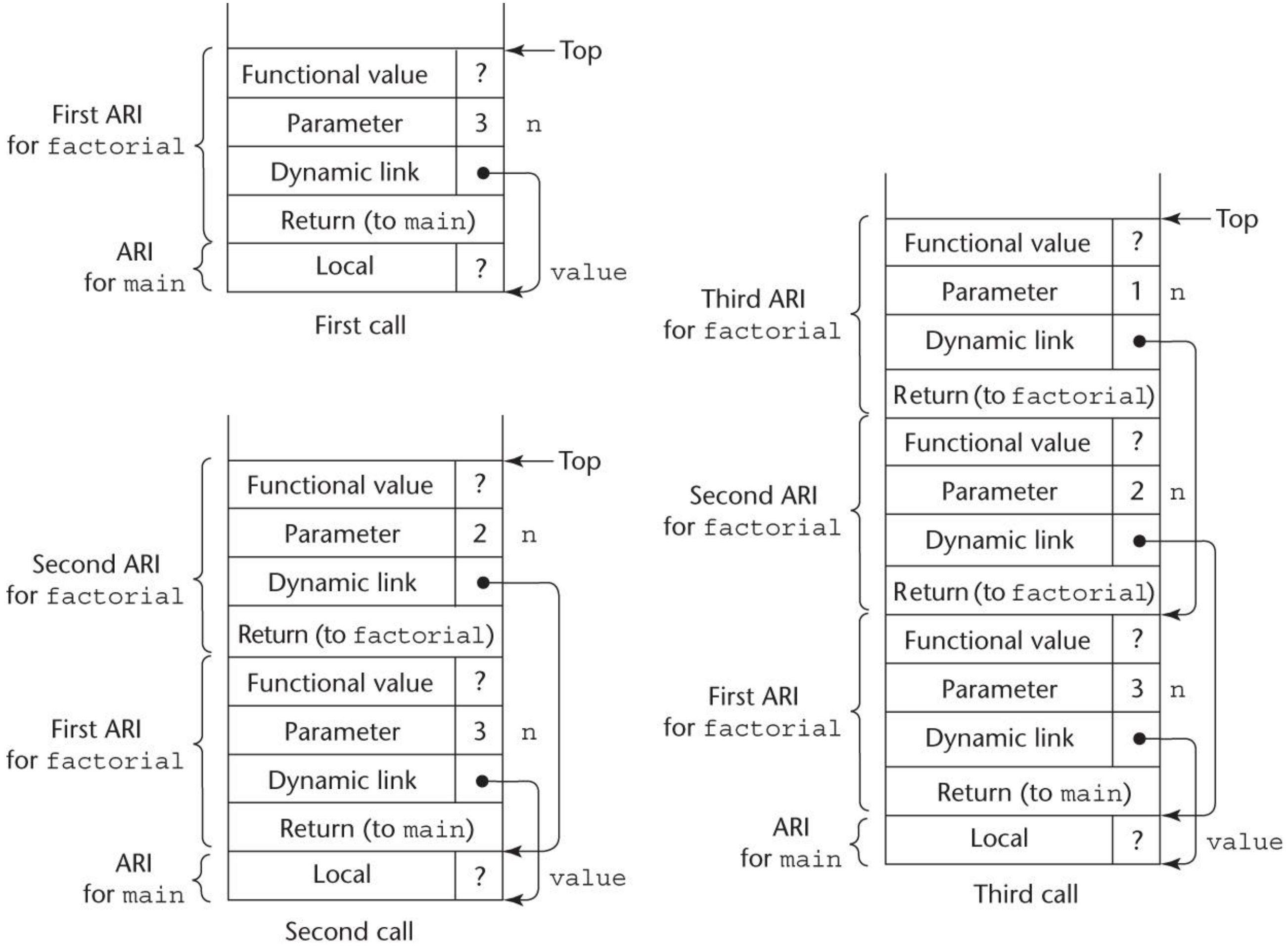
An Example With Recursion

```
int factorial (int n) {
   if (n <= 1) return 1;
   else return (n * factorial(n - 1));
}
    void main() {
      int value;
      value = factorial(3);
}
```



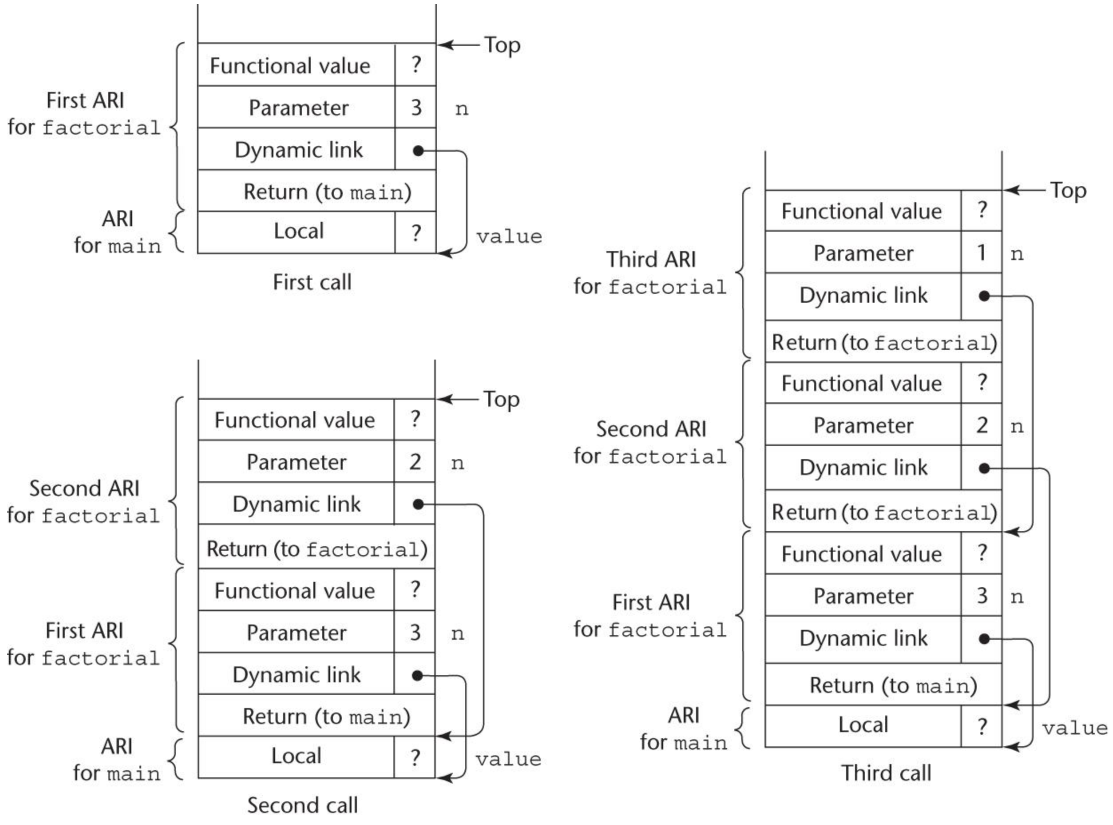
Stacks for calls to `factorial`

```
int factorial (int n) {
    if (n <= 1) return 1;
 else return (n * factorial(n - 1));
}
  void main() {
    int value;
    value = factorial(3);
}
```

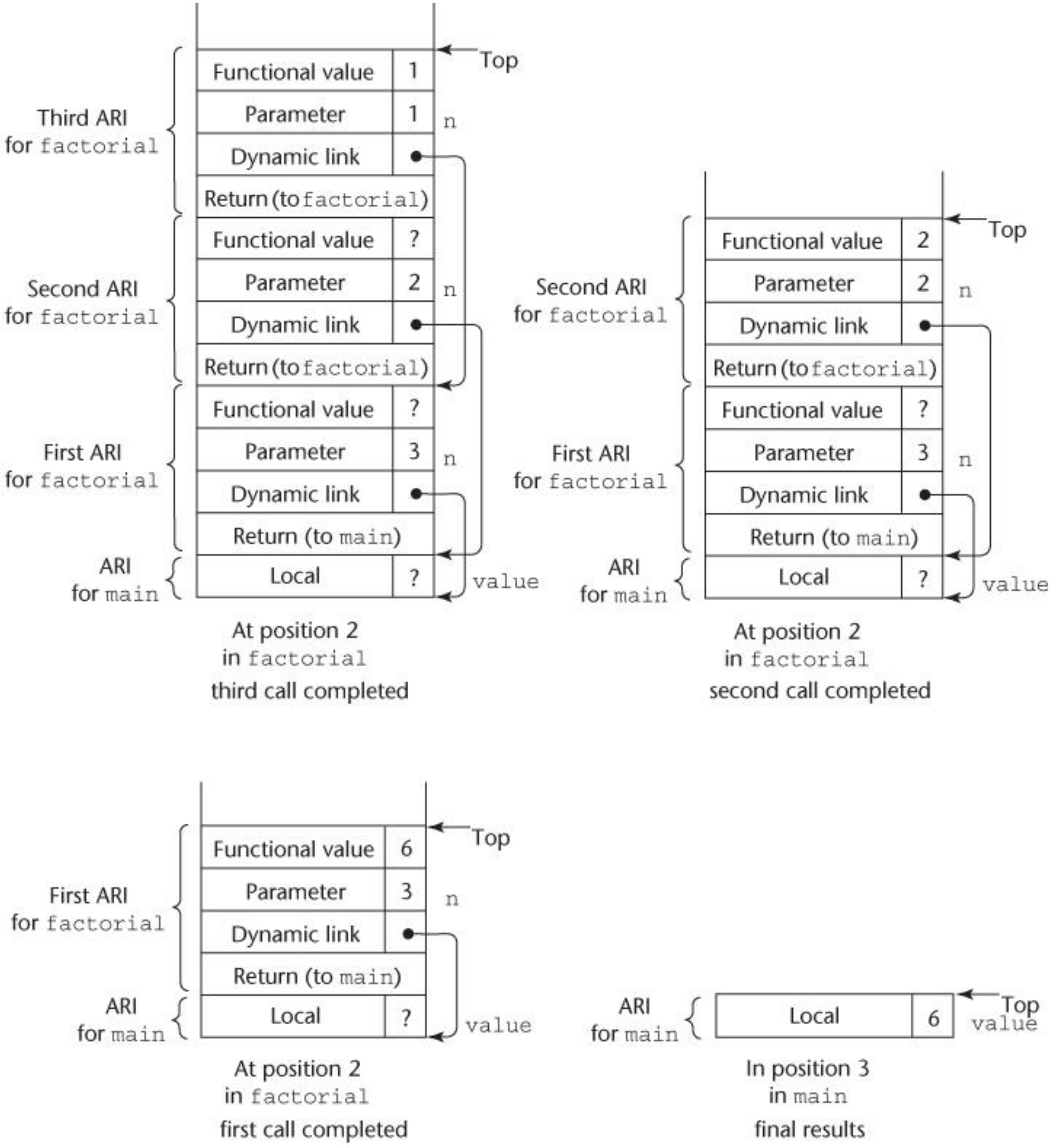


ARI = activation record instance

Stacks for returns from `factorial`



ARI = activation record instance



ARI = activation record instance

Scoping: How is scope maintained

- Scope: the accessibility of a variable or more generally the context in which an identifier is valid

- Static Scope
  - Parent scope in lexical sense – where the identifier is textually defined.
  - Each scope must have link to parent scope
    - A tree: one link per scope to parent

- Dynamic Scope
  - Parent scope in dynamic sense (sequence of function calls)
    - What is the scope of the calling function
    - Can be maintained directly using the dynamic link of the runtime stack

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

Statics Scope Design Concern : Nested Subprograms

- If embedded functions (embedded scopes) are not permitted, static scope is fairly easy to implement

- 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

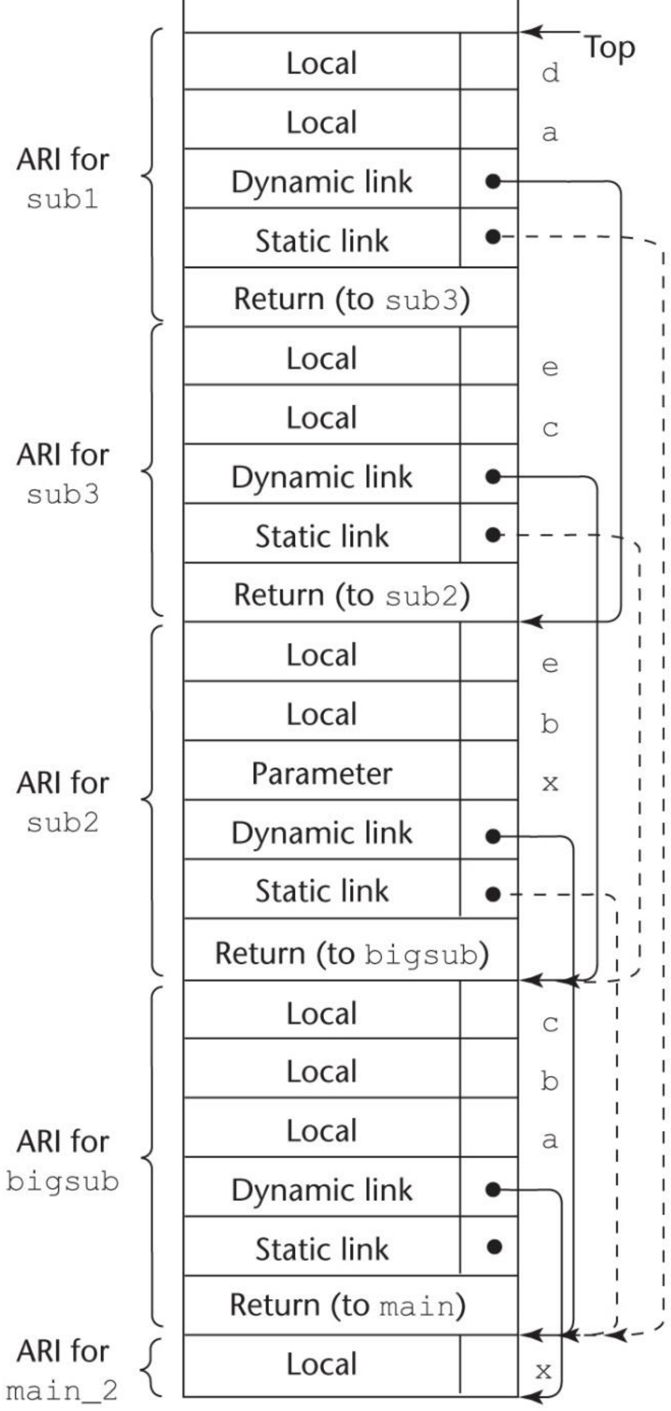
Example JavaScript Program

```
function main(){
 var x;
 function bigsub() {
  var a, b, c;
  function sub1 {
   var a, d;
    a = b + c; <------------------------------1
   ...
  }  // end of sub1
  function sub2(x) {
   var b, e;
   function sub3() {
     var c, e;
    ...
     sub1();
     ...
    e = b + a; <-----------------------------2
   } // end of sub3 ...
   sub3();
   ...
    a = d + e; <------------------------------3
  } // end of sub2
  ...
  sub2(7);
  ...
 } // end of bigsub
 ...
 bigsub();
 ...
} // end of main
```

Example JavaScript Program (continued)

- Call sequence for `main`

  `main` calls `bigsub`
  `bigsub` calls `sub2`
  `sub2` calls `sub3`
  `sub3` calls `sub1`



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

Static Scope Design Concern: 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

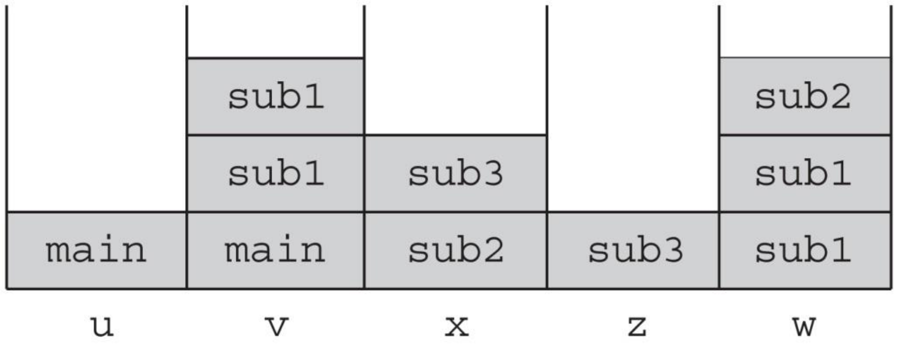
Implementing Dynamic Scoping

- *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.)

More scope examples: Simple Static Scope Example

- Static Scope Variable resolution
  - Search current scope
  - If not found, search enclosing (parent) scope.

- Output
  2
  3

```
int x=1;
main() {
  x = 3;
  {
    int x;
    {
      x = 2;
    }
    printf("%d\n",x);
  }
  printf("%d\n",x);
  return 0;
}
```

GEORGETOWN UNIVERSITY

Static vs. Dynamic Scope Example

- Dynamic Scope Variable resolution
  - Search current scope.
  - If not found, search scope of the caller (and, repeat as necessary).

- Static Scope Output
  2
  2

- Dynamic Scope Output
  3
  4

```
int x;

int main() {
  x = 2;
  f();
  g();
}

void f() {
  int x = 3;
  h();
}

void g() {
  int x = 4;
  h();
}

void h() {
  printf("%d\n",x);
}
```

Summary

- Subprogram linkage semantics requires many steps by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex and more versatile
- Subprograms with stack-dynamic local variables and nested subprograms have two components
  - actual code
  - activation record (potentially multiple active instances)

Summary

- 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