Implementing Subprograms

BBM 301 – Programming Languages

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 subprogram calls
  - 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:
  - Out 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

- Simple Programs:
  - Subprograms cannot be nested
  - All local variables are static
  - Ex: Early versions of Fortran
- Call Semantics:
  - Save the execution status of the caller (caller/callee)
  - Pass the parameters (caller)
  - Pass the return address to the subprogram (caller)
  - Transfer control to the subprogram (caller)

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 (callee)
  - If it is a function, move the functional value to a place the caller can get it (callee)
  - Restore the execution status of the caller (caller/callee)
  - Transfer control back to the caller (callee)

Storage required for call/return actions

- Required storage:
  - Status information about the caller
  - Parameters
  - return address
  - return value for functions

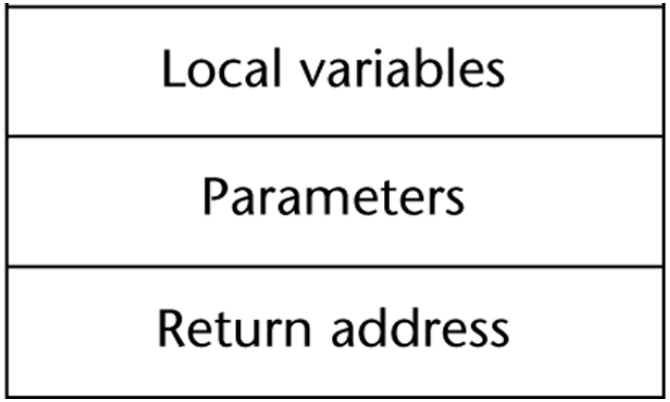
- These along with local variables and the subprogram code, form the complete collection of information a subprogram needs to execute and then return to the caller

Implementing "Simple" Subprograms: Parts

- Two separate parts: the actual code (constant) 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*
- The form of an activation record is static.
- An *activation record instance (ARI)* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

An Activation Record for "Simple" Subprograms

Because languages with simple subprograms do not support recursion, there can be only one active version of a given subprogram at a time

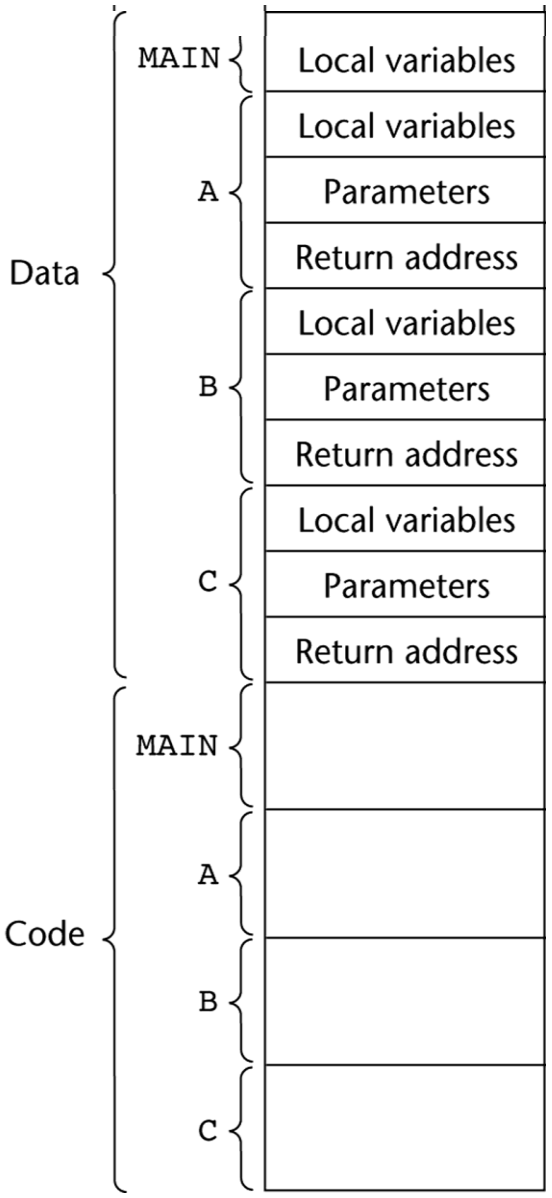


Therefore, there can be only a single instance of the activation record for a subprogram

Since activation record instance of a simple subprogram has fixed size it can be statically allocated. It could be also attached to the code part

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

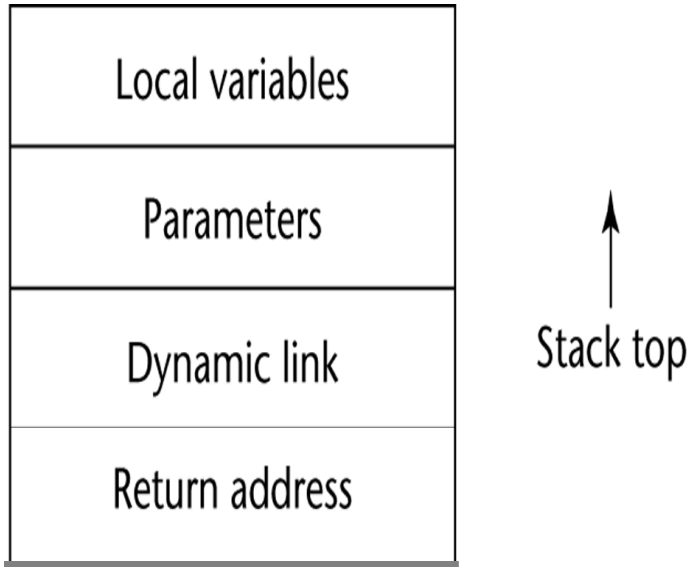
- Note that code could be attached to ARIs
- Also, the four program units could be compiled at different times
- Linker put the compiled parts together when it is called for the main program



Implementing Subprograms with Stack-Dynamic Local Variables

- Advantage of Stack-Dynamic Local Variables
  - Support for recursion
- 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)
  - Each activation needs its own activation record and the number of activations is limited only by the memory size of the machine.

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



Return address: pointer to the code segment of the caller and an offset address in that code segment of the instruction following the call

Dynamic link: pointer to the base of the activation record instance of the caller

In static scoped languages this link is used to provide traceback information when a run-time error occurs.

In dynamic-scoped languages, the dynamic link is used to access non-local variables

Because the return address, dynamic link, and parameters are placed in the activation record instance by the caller, these entries must appear first.

Implementing Subprograms with Stack-Dynamic Local Variables

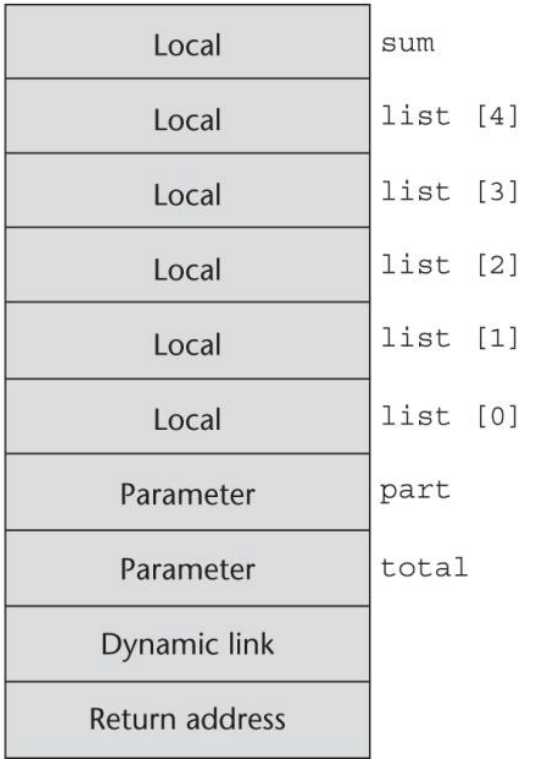
- In most languages,
  - The format of the activation record is known at compile time
  - In many cases, the size is also known, because all the local data are of fixed size.
  - In some languages, like Ada, the size of a local array can depend on the value of an actual parameter, so in those cases the format is static, but the size can be dynamic.

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- An activation record instance is dynamically created when a subprogram is called
- Because the call and return semantics specify that the subprogram last called is the first to complete, it is reasonable to use a stack, so
  - Activation record instances reside on the run-time stack
- Every subprogram activation (recursive or non-recursive) creates a new instance of an activation record on the 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

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

Revised Semantic Call/Return Actions

Before (Prologue)

- Save the old EP to the stack as the dynamic link and create the new value
- Allocate local variables

After (Epilogue)

- 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 stack pointer by setting it to the value of the current EP minus one 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);
   ...
}
```

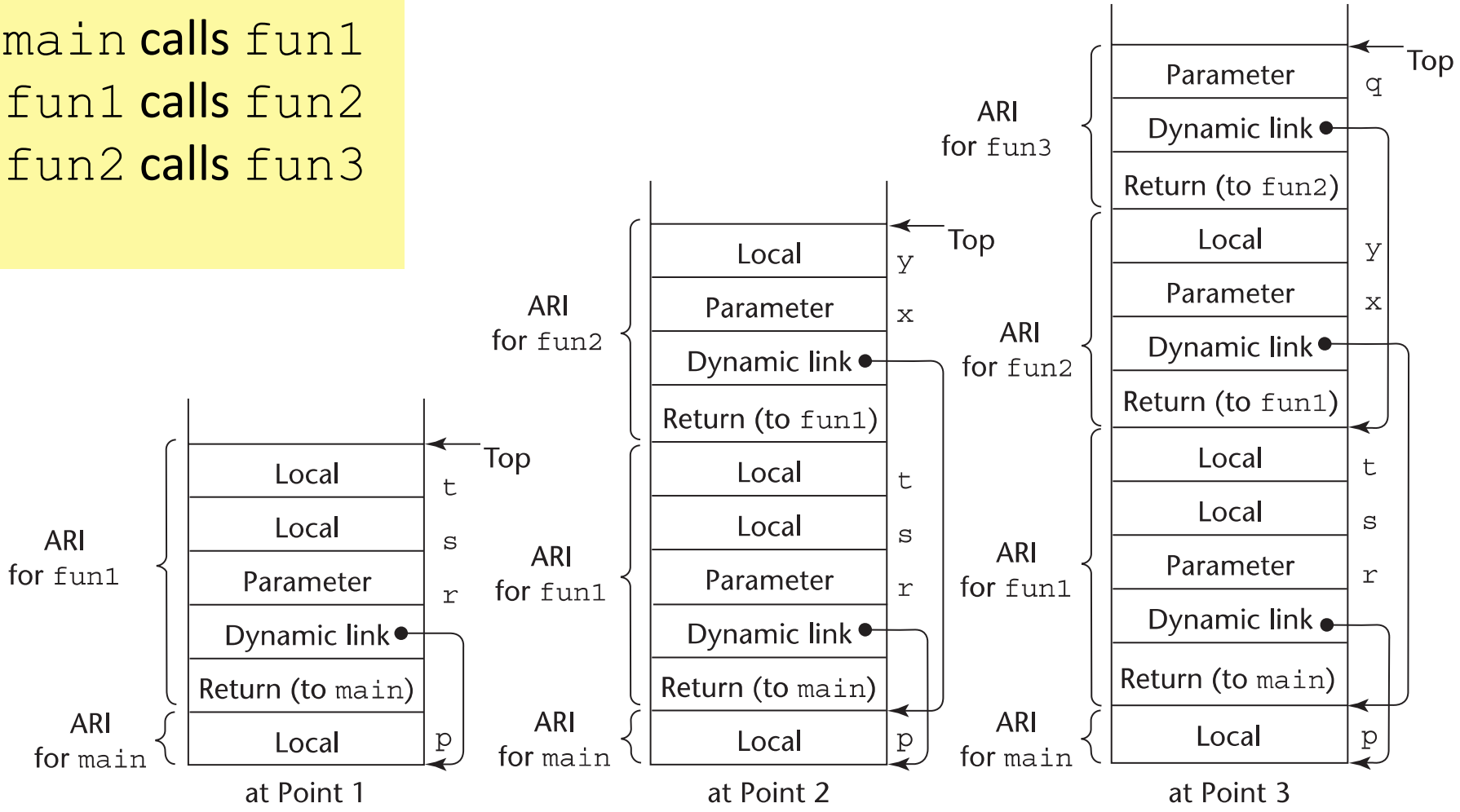
`main` calls `fun1`

`fun1` calls `fun2`

`fun2` calls `fun3`

An Example Without Recursion

`main` calls `fun1`
`fun1` calls `fun2`
`fun2` calls `fun3`



ARI = activation record instance

1-18

Dynamic Chain and Local Offset

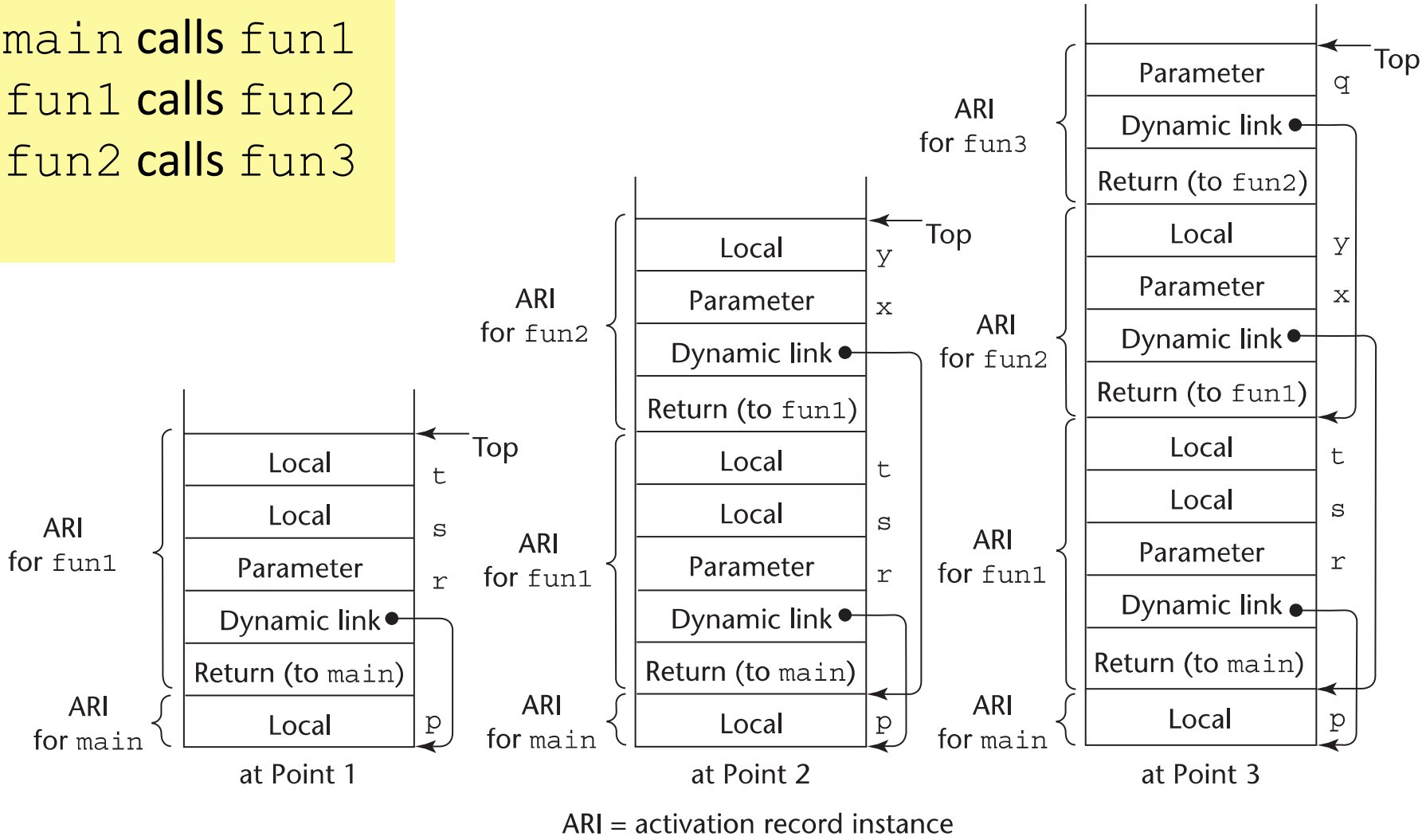
- *Dynamic chain (call chain):* The collection of dynamic links in the stack at a given time
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the *local_offset*
- The local_offset of a local variable can be determined by the compiler at compile time
  - How?

Local offset

- The local_offset of a variable in an activation record can be determined at compile time, using the order, types, and sizes of variables.
- For simplicity, we assume that all variables take one position in the activation record.

An Example Without Recursion

`main` calls `fun1`
`fun1` calls `fun2`
`fun2` calls `fun3`



ARI = activation record instance

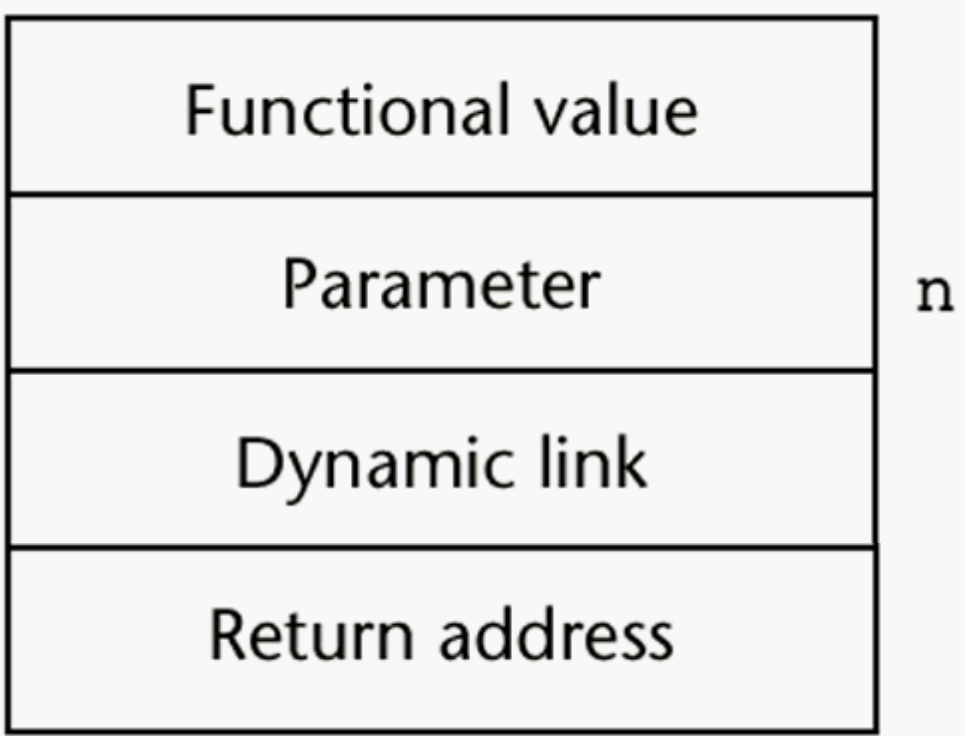
In fun1, the local_offset of `s` is 3; for `t` it is 4.
In fun2, the local_offset of `y` is 3.

1-21

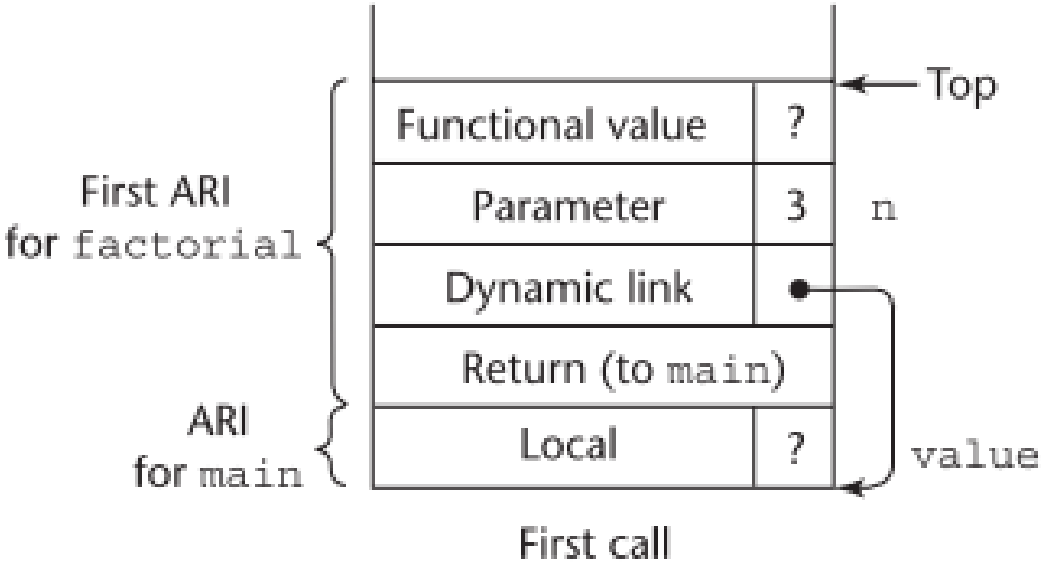
Recursion

```
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
}
```

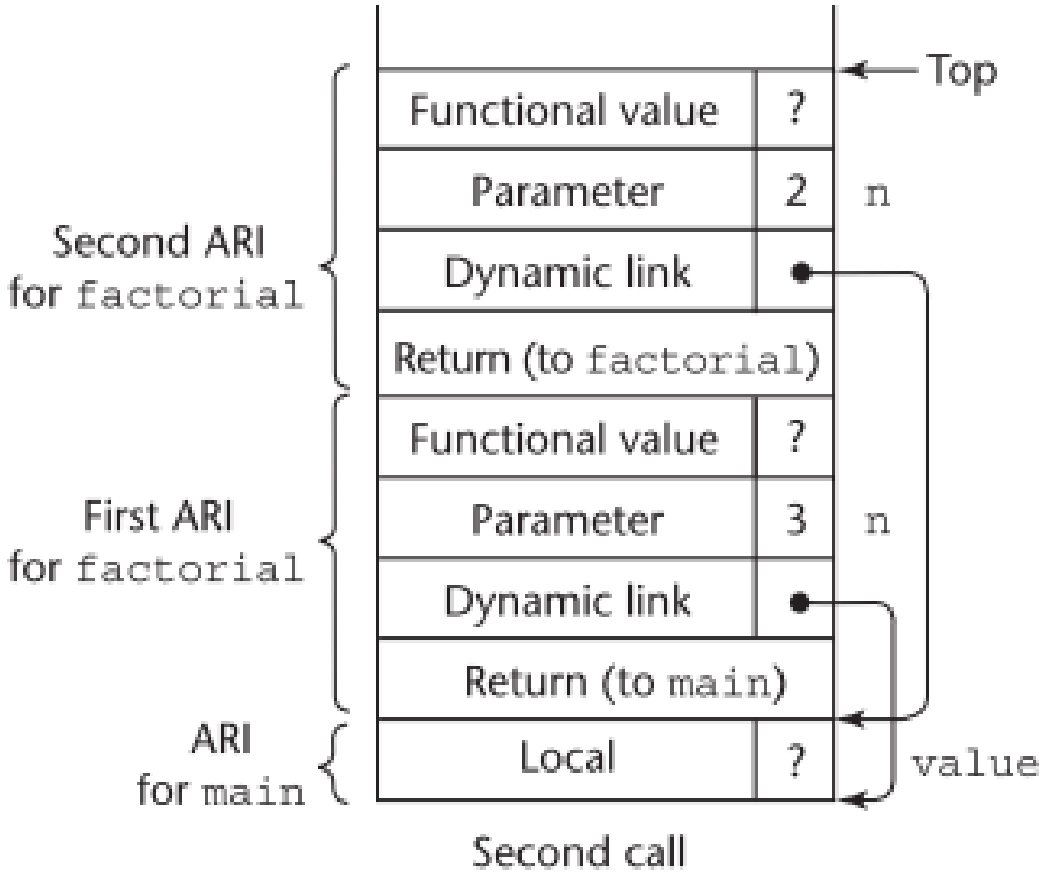
Activation Record for `factorial`



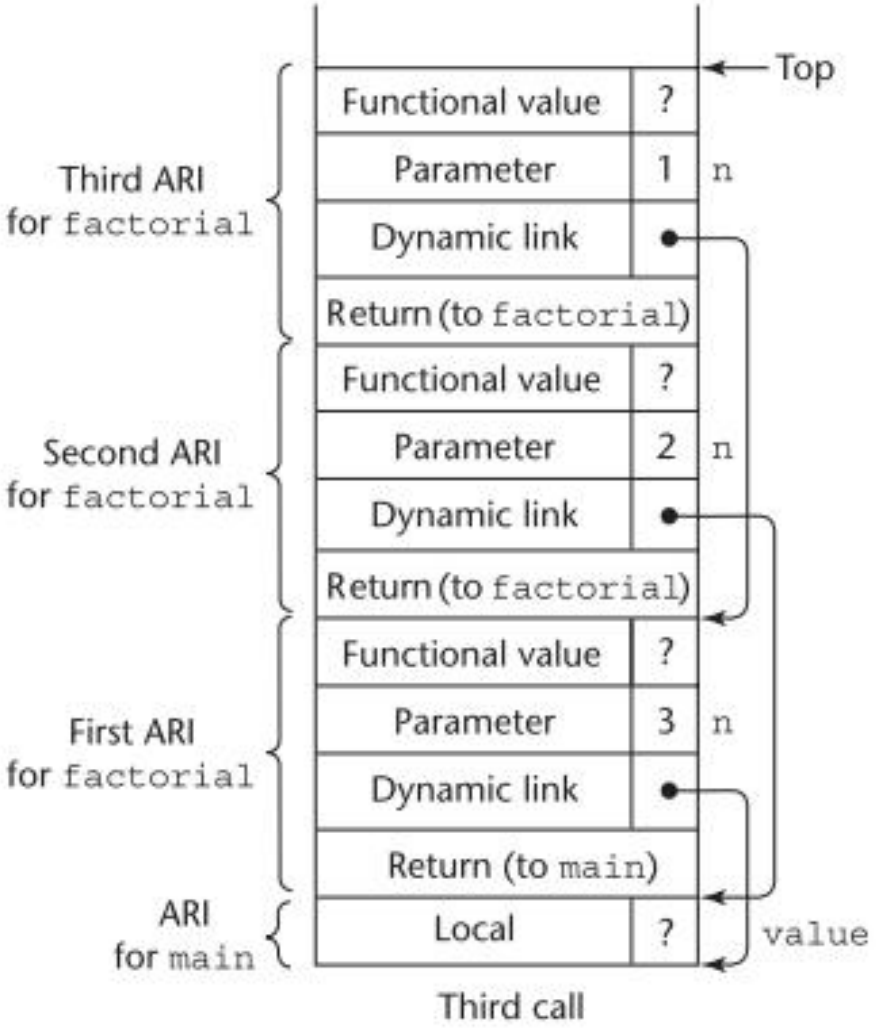
Stack contents during execution of `main` and `factorial`



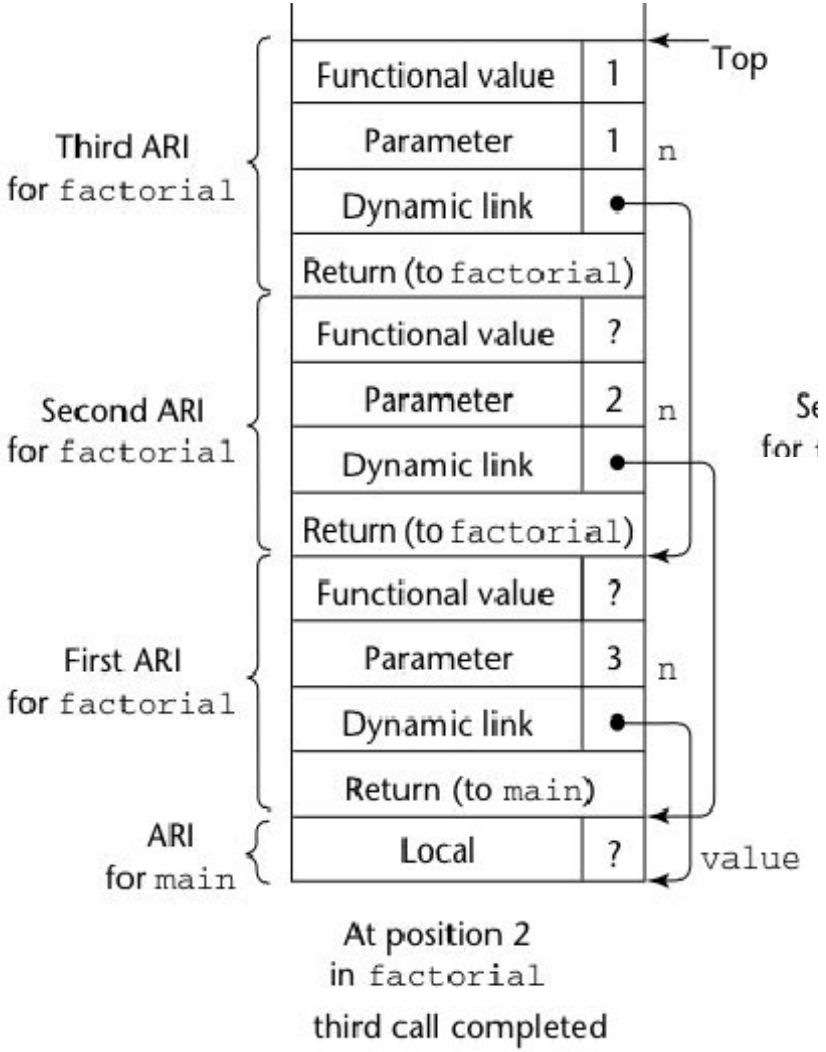
Stack contents during execution of `main` and `factorial`



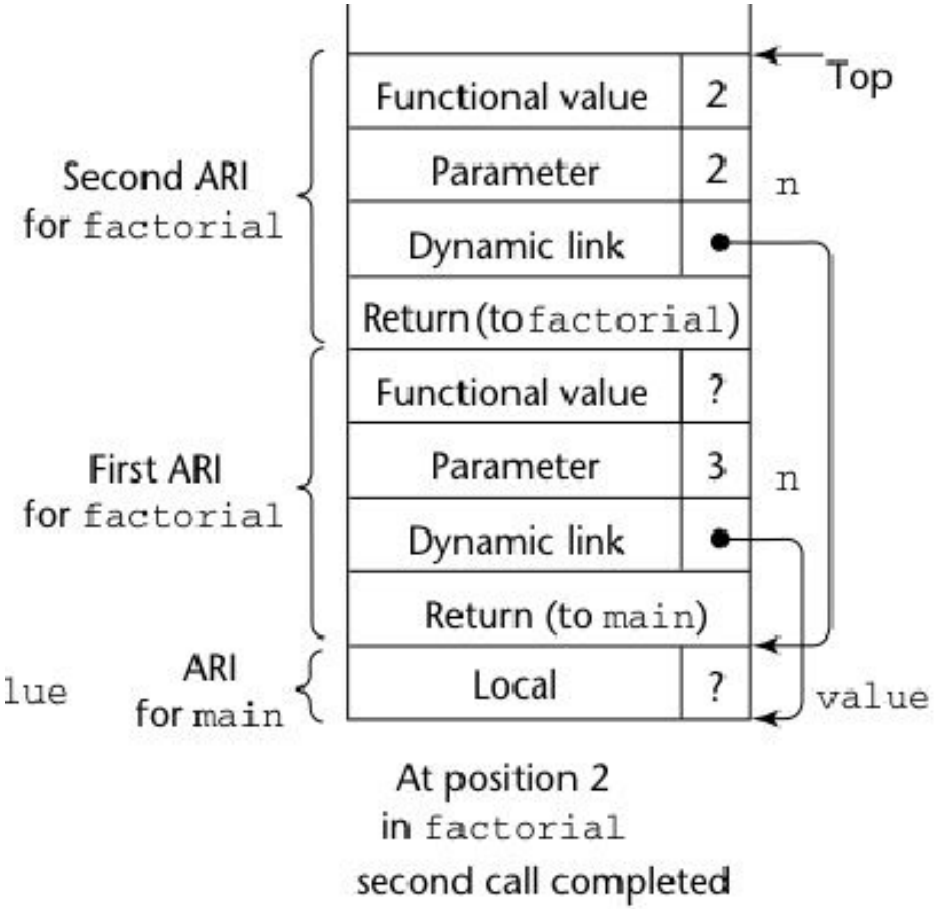
Stack contents during execution of `main` and `factorial`



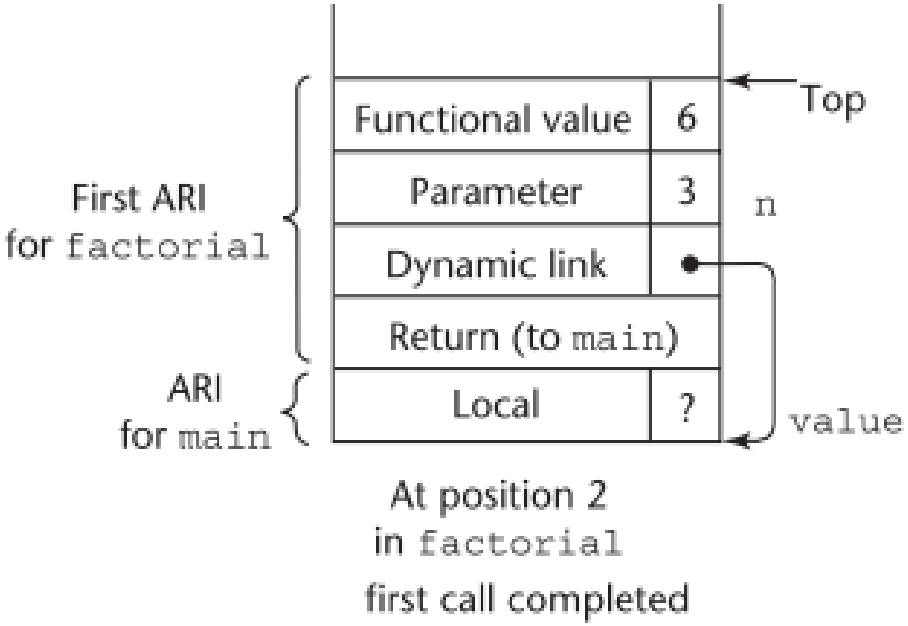
Stack contents during execution of `main` and `factorial`



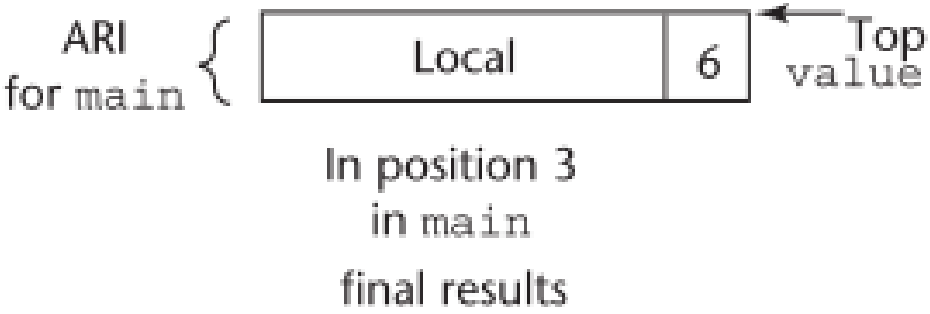
Stack contents during execution of `main` and `factorial`



Stack contents during execution of `main` and `factorial`



Stack contents during execution of `main` and `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

Implementing Static Scoping

- The *static link* in an activation record instance for subprogram A points to the bottom of one of the activation record instances of A's static parent

- A *static chain* is a chain of static links that connects certain activation record instances

- 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 => indicates how deeply it is nested in the outermost scope

Static Scoping (continued)

- A program unit that is not nested inside any other unit has a static_depth of 0.
- A subprogram A is defined in a nonnested program unit, its static_depth is 1.
- If subprogram A contains the definition of a nested subprogram B , then B ’s static_depth is 2.

- The *chain_offset* or *nesting_depth* of a nonlocal reference is the difference between the static_depth of the reference and static_depth of the procedure containing its declaration

- 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

```
# Global scope
. . .
def f1():
    def f2():
        def f3():
                . . .
       # end of f3
     . . .
   # end of f2
. . .
# end of f1
```

Static depth of f1 = 0

Static depth of f2 = 1

Static depth of f3 = 2

Example Ada Program*

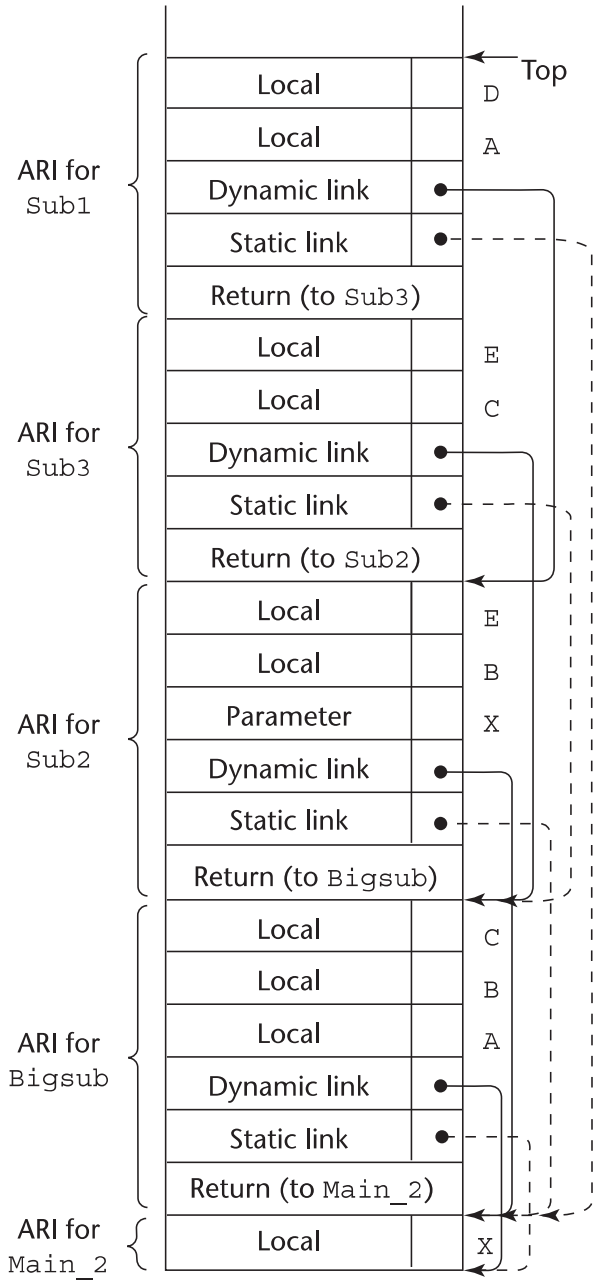
```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
      A := B + C;  <-----------------------1
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
  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



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 }
```

Variable	Chain offset	Local offset
A at position 1		
B at position 1		
C at position 1		
E at position 2		
B at position 2		
A at position 2		
A at position 3		
D at position 3		
E at position 3		

Evaluation of Static Chains

- Problems:
  1. A nonlocal reference 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

- Recall that 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

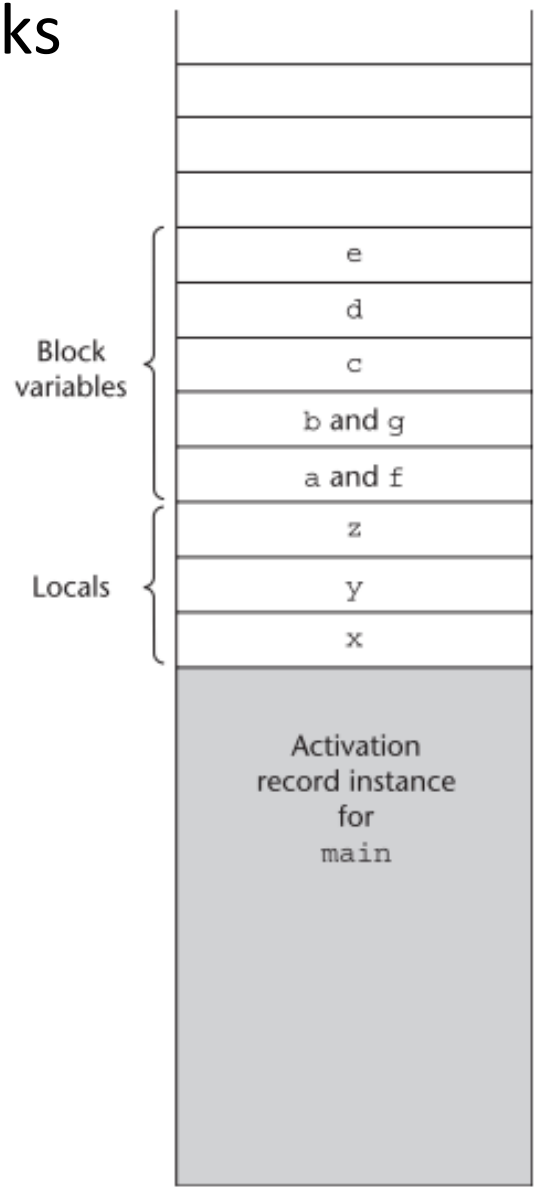
Implementing Blocks

- 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 Blocks

- Block variable storage when blocks are not treated as parameterless procedures

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

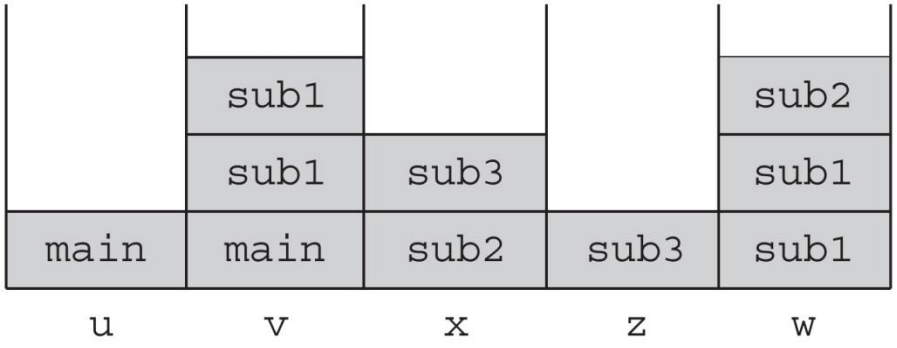


Implementing Dynamic Scoping

- *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.)

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3

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 table method