Chapter 9

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*
- A subprogram call has numerous actions associated with it
  - Parameter passing methods
  - Static local variables
  - Execution status of calling program
  - Transfer of control
  - Subprogram nesting

IMPLEMENTING "SIMPLE" SUBPROGRAMS: CALL SEMANTICS

- Save the execution status of the caller
- Carry out the parameter-passing process
- Pass the return address to the callee
- Transfer control to the callee

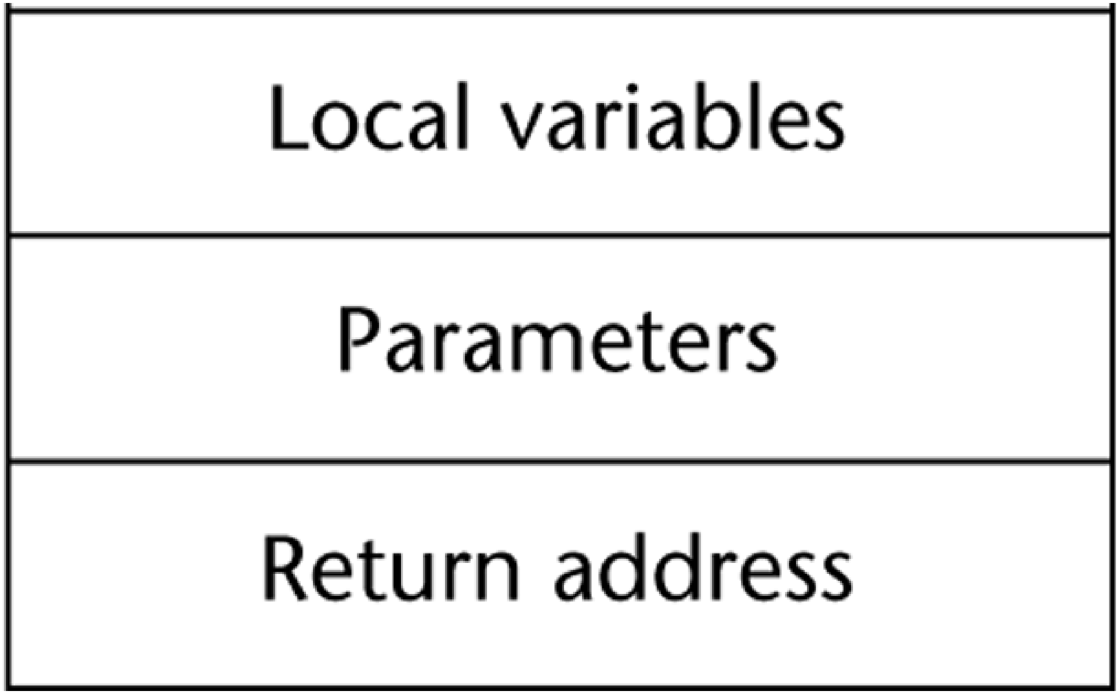
Implementing "Simple" Subprograms: Return Semantics

- If pass-by-value-result 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

Implementing "Simple" Subprograms: Parts

- Two separate parts: the actual code and the noncode part (local variables and data that can change)
- The format, or layout, of the noncode 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

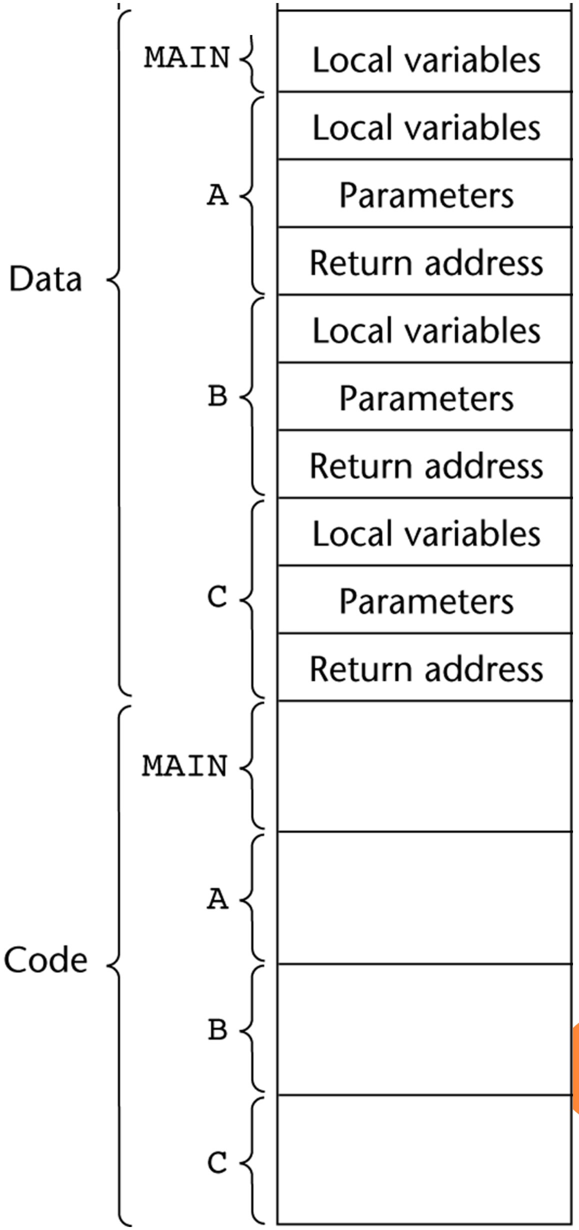


Local variables

Parameters

Return address

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



Data

MAIN

Local variables

A

Local variables

Parameters

Return address

B

Local variables

Parameters

Return address

C

Local variables

Parameters

Return address

Code

MAIN

A

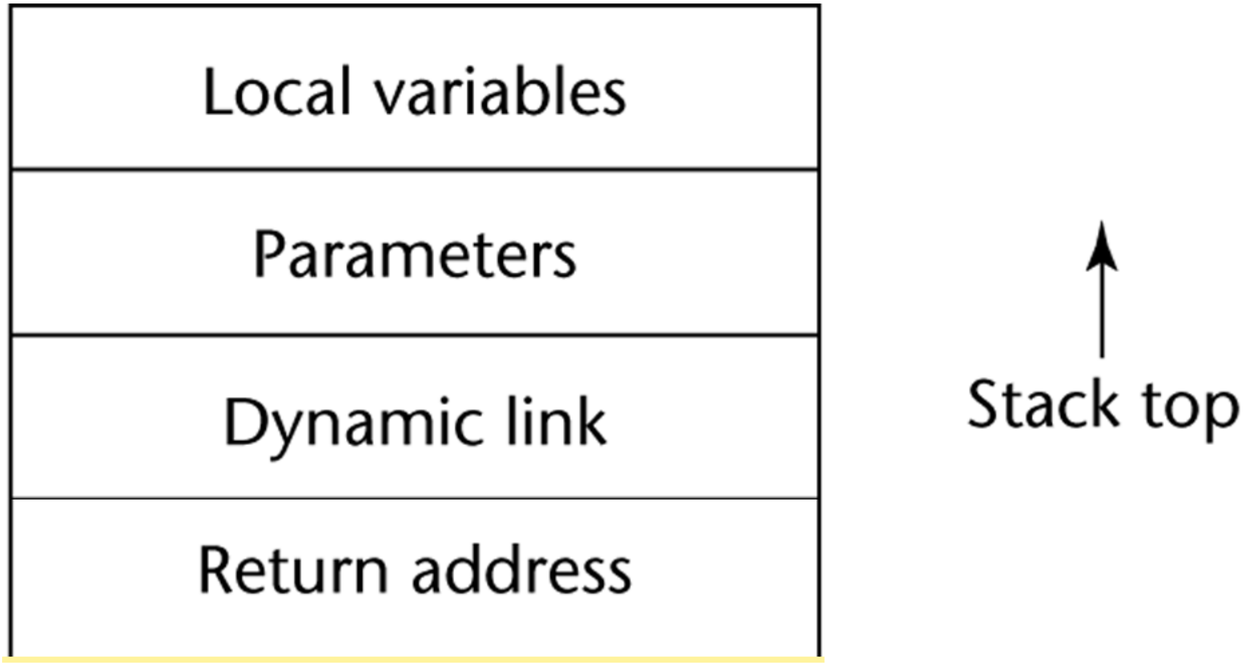
B

C

Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
  - The compiler must generate code to cause implicit allocation and de-allocation 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



Local variables

Parameters

Dynamic link

Return address

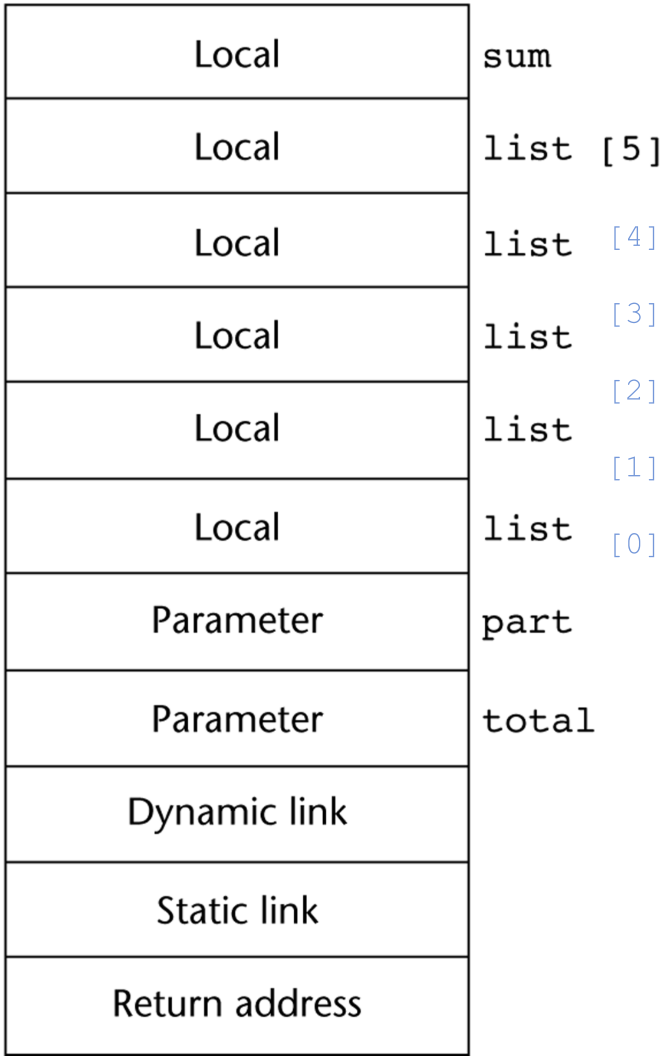
Stack top

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
- Run-time stack

AN EXAMPLE: C FUNCTION

```
void sub(float total, int part)
{
  int list[6];
  float sum;
  …
}
```

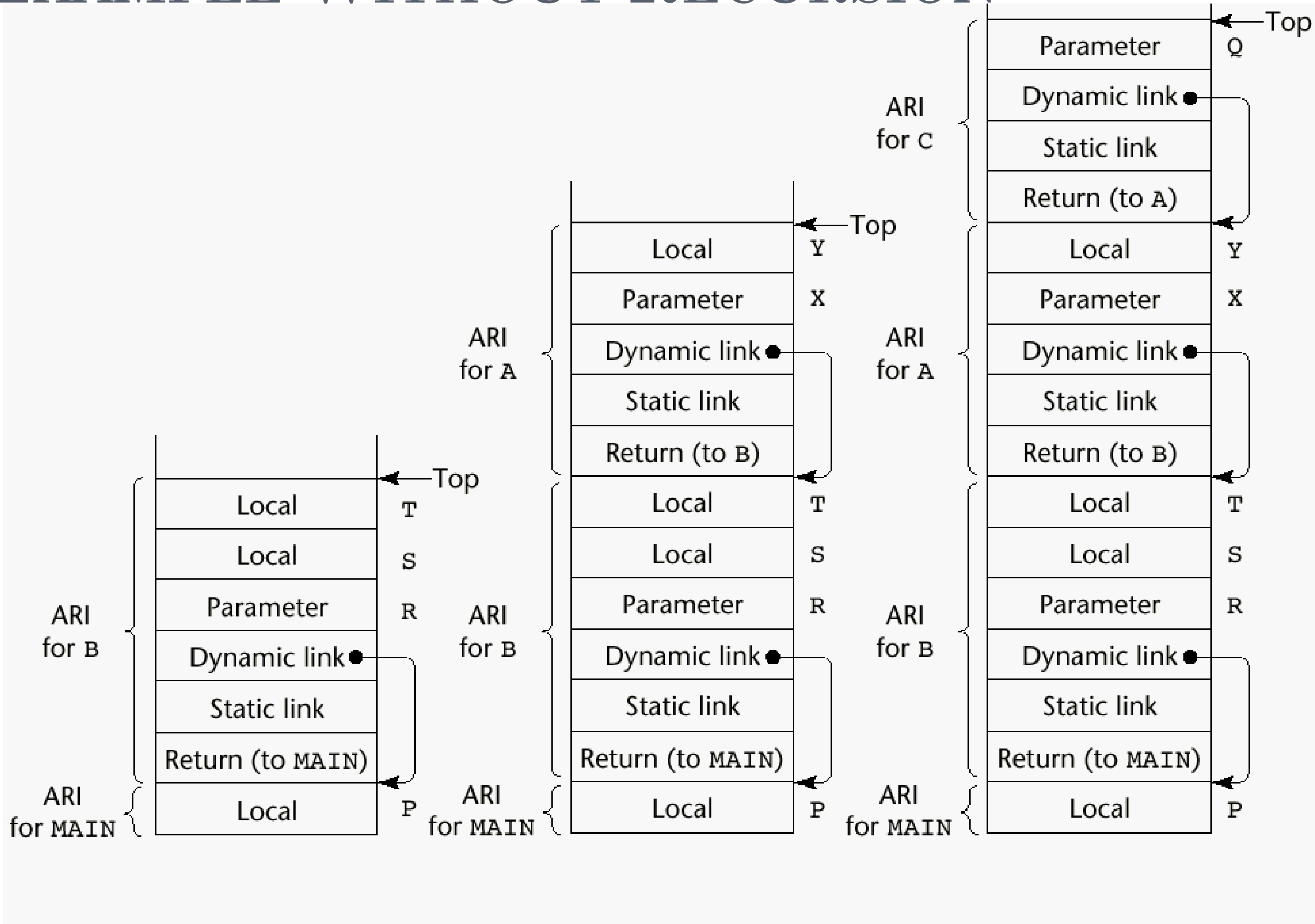


An Example Without Recursion

```
void A(int x) {
   int y;
   ...
   C(y);
   ...
}
void B(float r) {
   int s, t;
   ...
   A(s);
   ...
}
void C(int q) {
   ...
}
void main() {
   float p;
   ...
   B(p);
   ...
}
```

main calls B
B calls A
A calls C

An Example Without Recursion



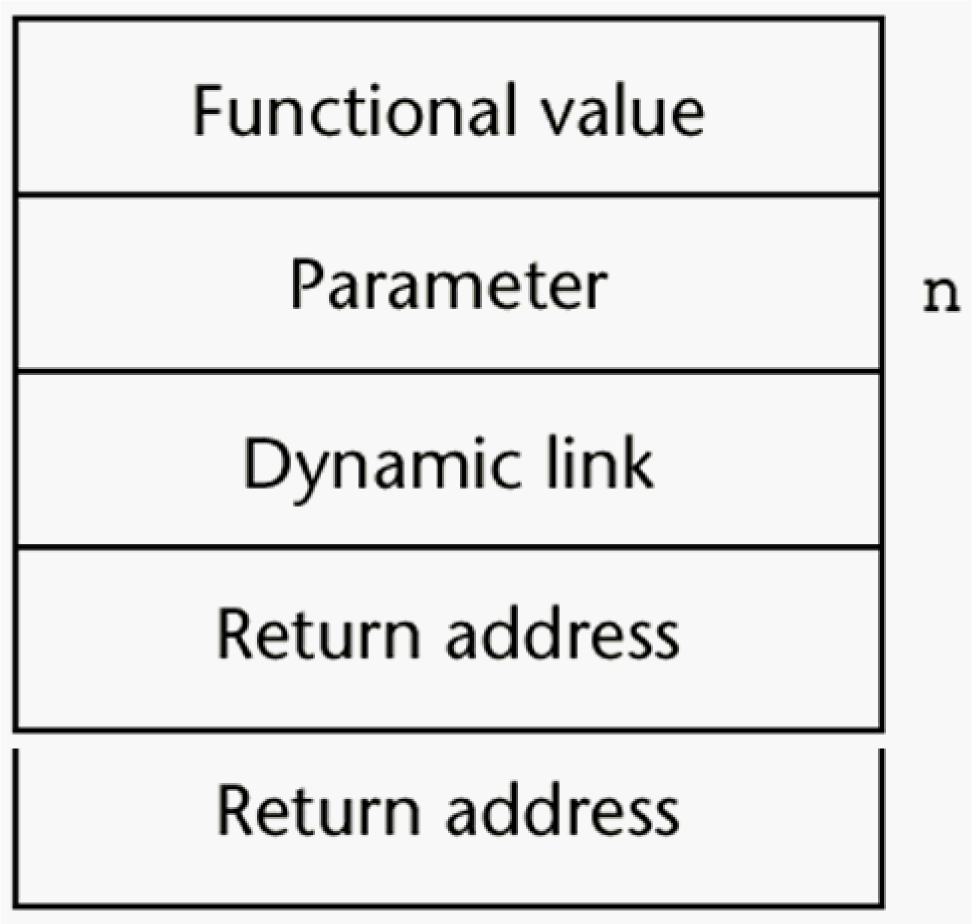
ARI = activation record instance

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
}
```

Activation Record for factorial



Functional value

Parameter

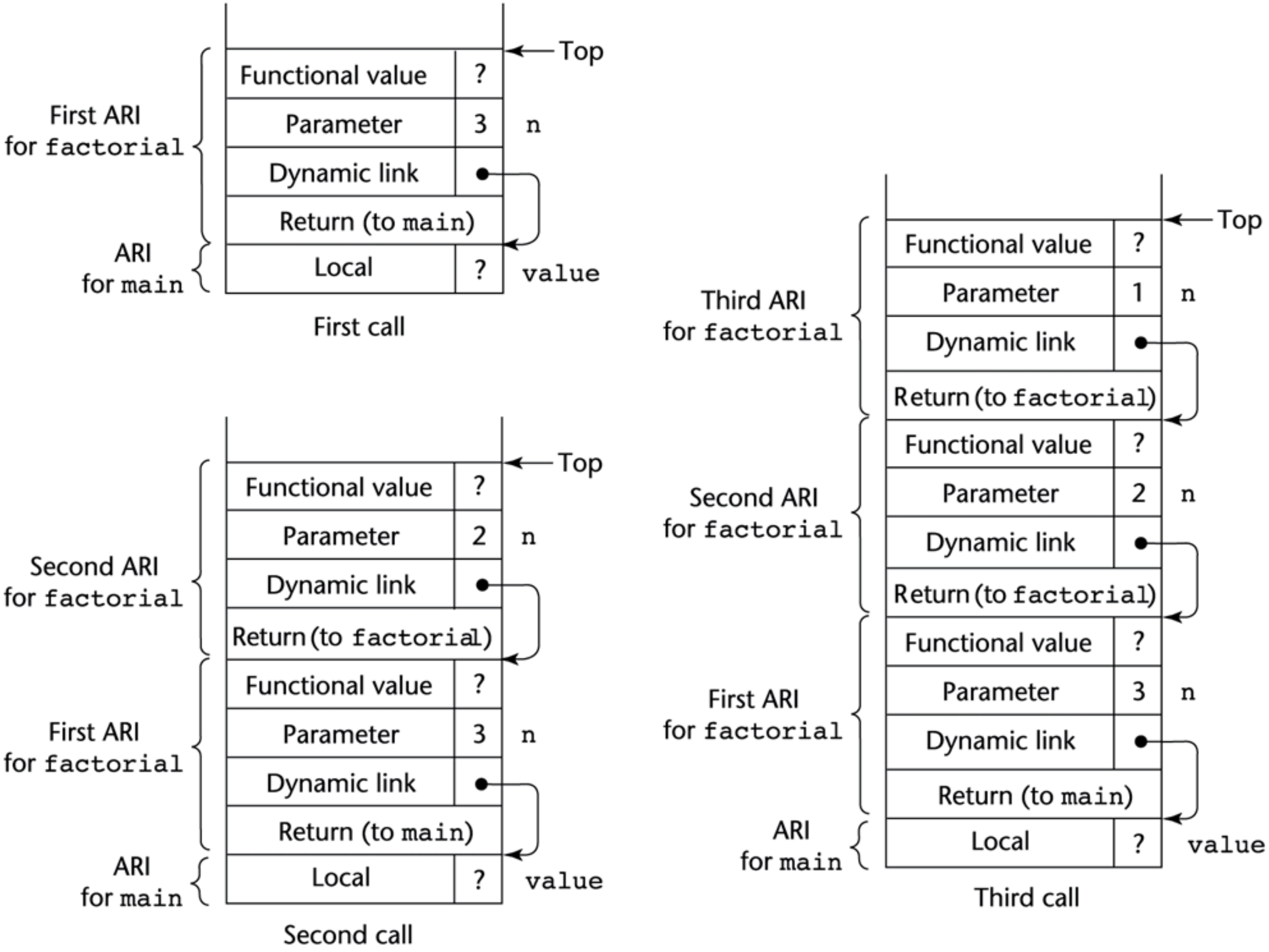
n

Dynamic link

Return address

Return address

EXAMPLE WITH RECURSION (PART I)



First ARI for factorial
- Functional value ?
- Parameter 3 n
- Dynamic link ●
- Return (to main)

ARI for main
- Local ? value

Top

First call

Second ARI for factorial
- Functional value ?
- Parameter 2 n
- Dynamic link ●
- Return (to factorial)

First ARI for factorial
- Functional value ?
- Parameter 3 n
- Dynamic link ●
- Return (to main)

ARI for main
- Local ? value

Top

Second call

Third ARI for factorial
- Functional value ?
- Parameter 1 n
- Dynamic link ●
- Return (to factorial)

Second ARI for factorial
- Functional value ?
- Parameter 2 n
- Dynamic link ●
- Return (to factorial)

First ARI for factorial
- Functional value ?
- Parameter 3 n
- Dynamic link ●
- Return (to main)

ARI for main
- Local ? value

Top

Third call

ARI = activation record instance

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

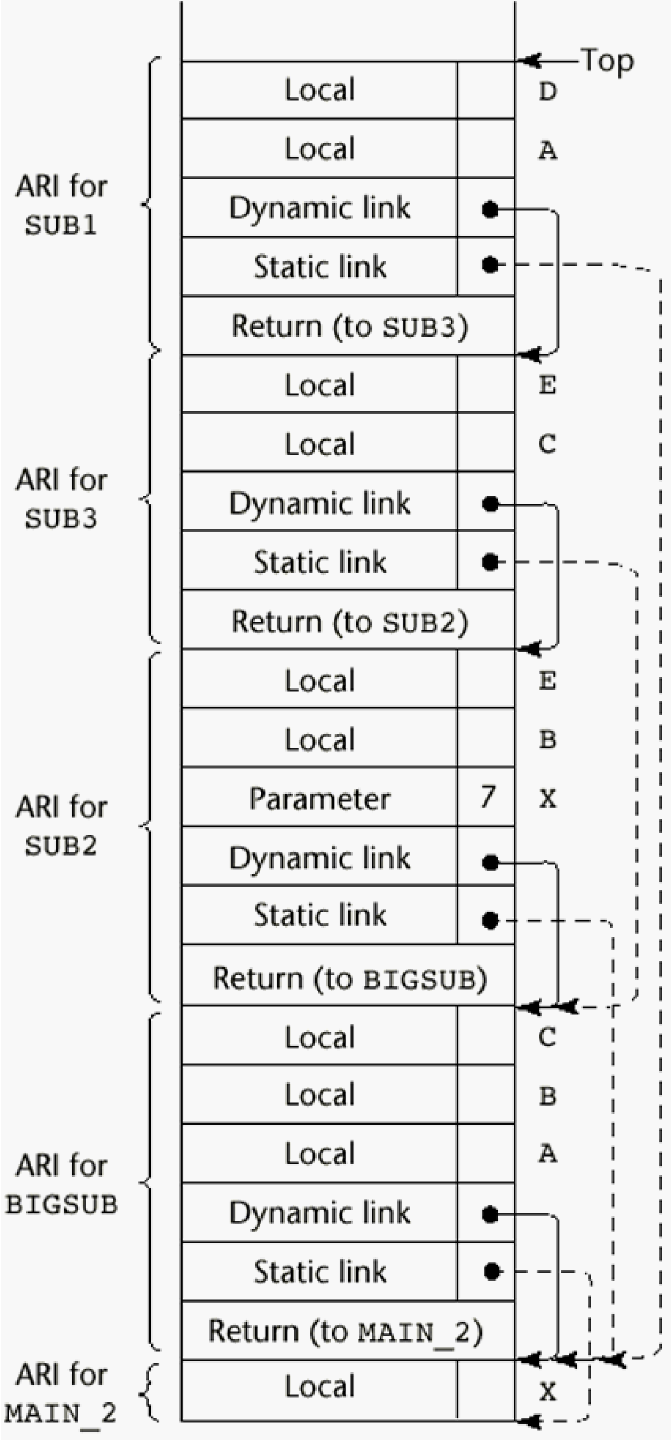
```
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <------------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        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



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

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