Chapter 10 Implementing Subprograms

- 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 of Simple Subprograms

Simple subprograms: subprograms cannot be nested and all local variables are static.

- Early versions of Fortran were examples of languages that had this kind of subprograms.

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-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

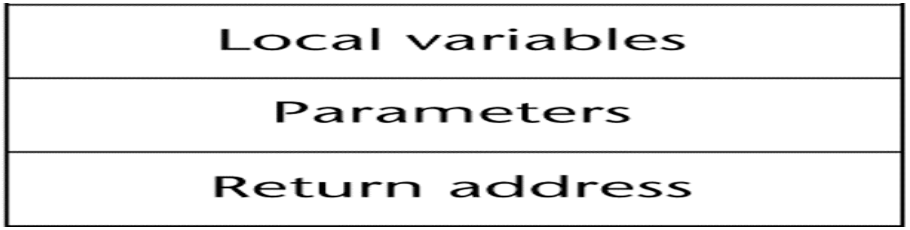
A simple subprogram consists of two separate parts:

- The actual code which is constant.
- The noncode part (local variables and data that can change) which also has fixed size.

- The format, or layout, of the noncode part of an executing subprogram is called an *activation record*

An Activation Record for "Simple" Subprograms

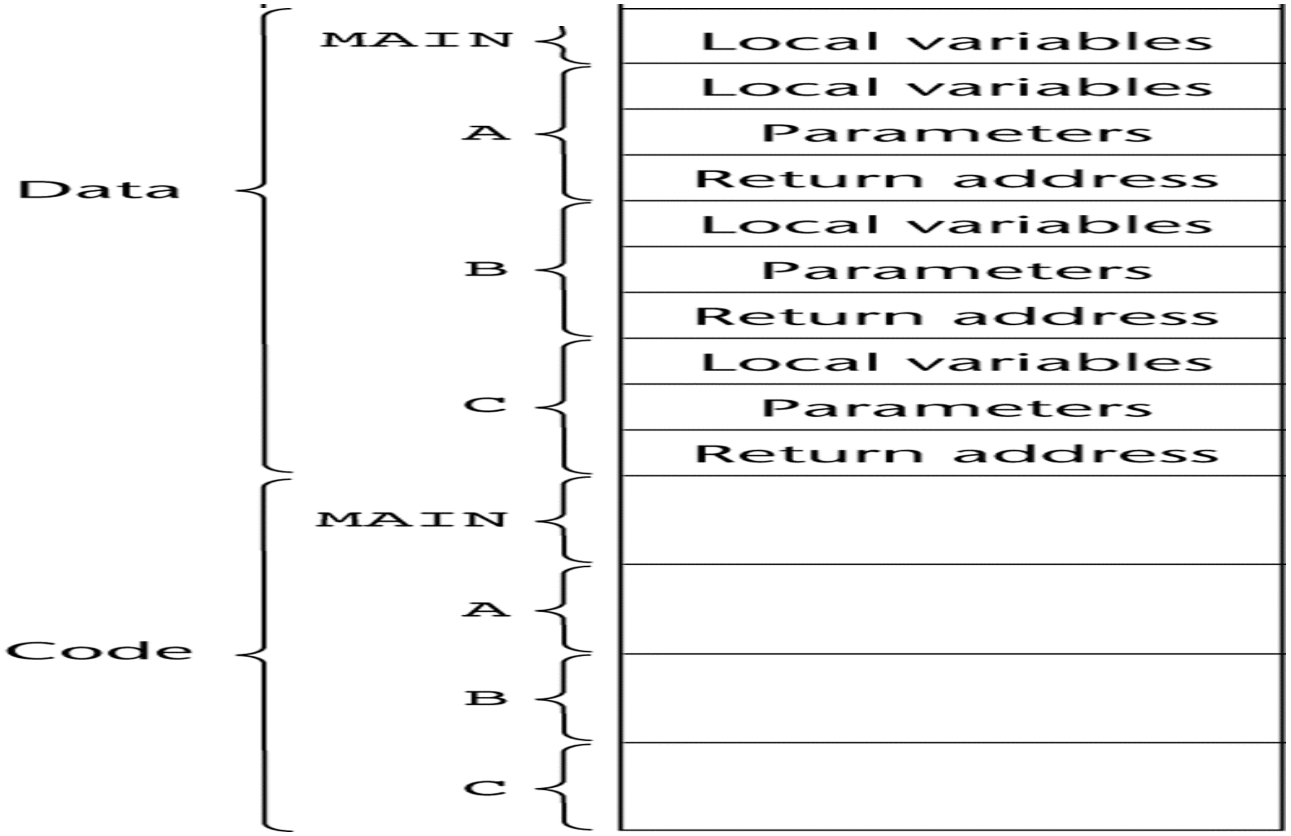
.



- An *activation record instance* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

- There can be only one active record instance of a given simple subprogram at a time: do not support recursion,

Activation record instances (A main program and three subprograms A, B and C)



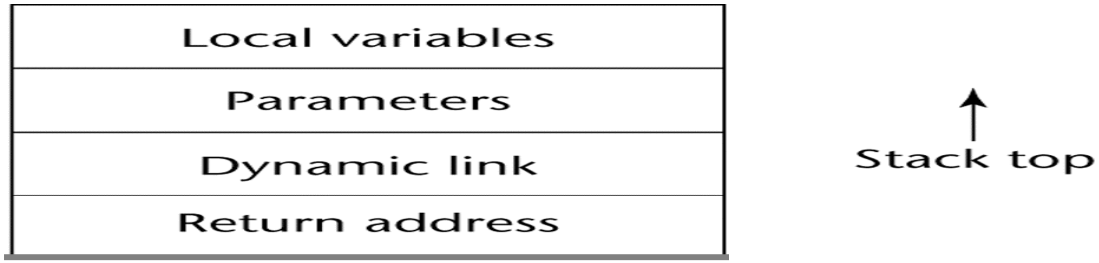
Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record

- The compiler must generate code to cause allocation and de-allocation of activation records

- Recursion must be supported since advantage of stack-dynamic local variable is support for recursion (adds the possibility of multiple simultaneous activations records of a subprogram)

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



Dynamic link: points to the top of an activation record of the caller

- In static-scoped languages, this link is used in the destruction of current activation record when the procedure completes its execution.

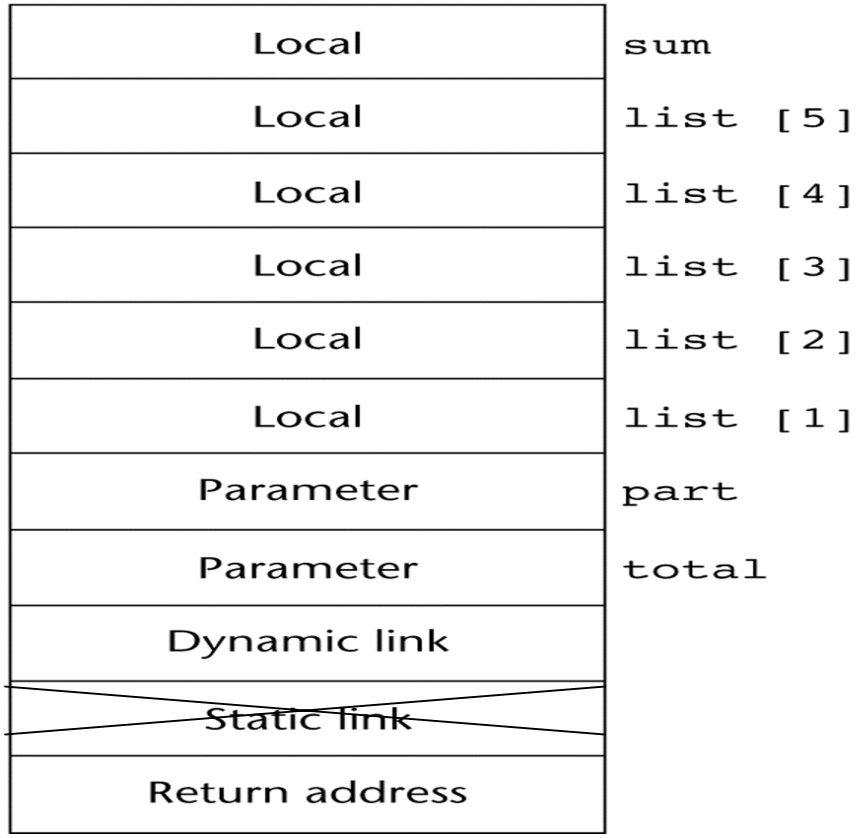
The collection of dynamic links in the stack at a given time is called the *dynamic chain*

Unlike simple subprogram, an activation record instance is dynamically created when a subprogram is called.

- Every recursive or non-recursive subprogram creates a new instance of an activation record on the stack.

An Activation Record Example: C Function

```
void sub(float total, int part){
    int list[5];
    float sum;
    …}
```

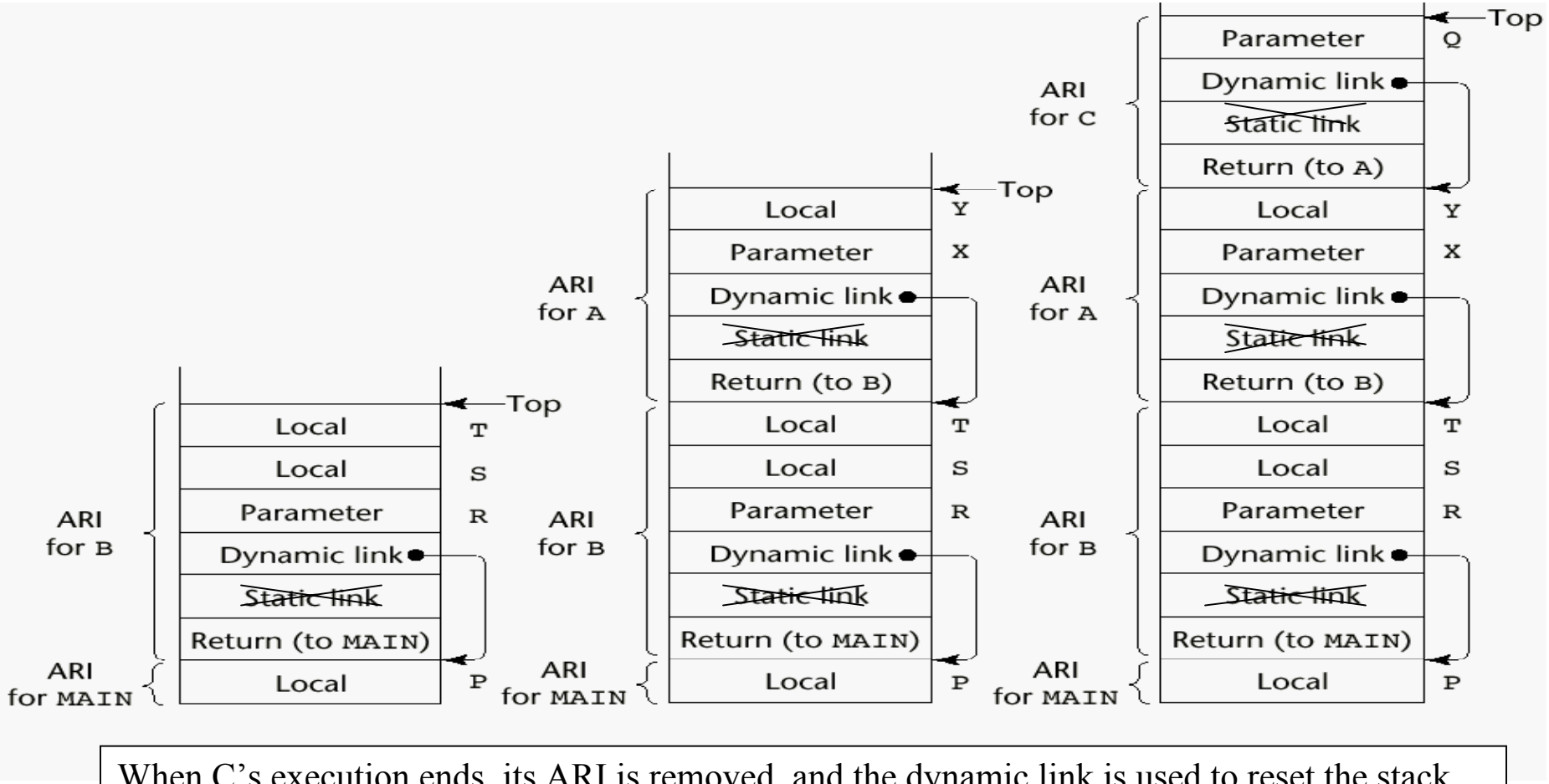


An Example Without Recursion

```
void A (int x) {
int y;
… <---------------------- point 2
  C(y);
…}

void B (float r) {
int s, t;
… <---------------------- point 1
  A(s);
…}

void C (int q) {
…… <---------------------- point 3
}

void main () {
float p;
…
  B(p);
…}
```

A call sequence:
main calls B
B calls A
A calls C

Activation records for position 1, 2 and 3 (ARI: activation record instance)



When C's execution ends, its ARI is removed, and the dynamic link is used to reset the stack top pointer. A similar process takes place when functions A and B terminates.

local_offset of variables

Reference to local variables can be represented in the code as offset from the beginning of the activation record of the local scope. Such an offset is called a **local_offset.**

• The first local variable declared has an offset of two (return address and dynamic link) plus the number of parameters.

local_offset of s in B is 3
local_offset of t in B is 4
local_offset of y in A is 3

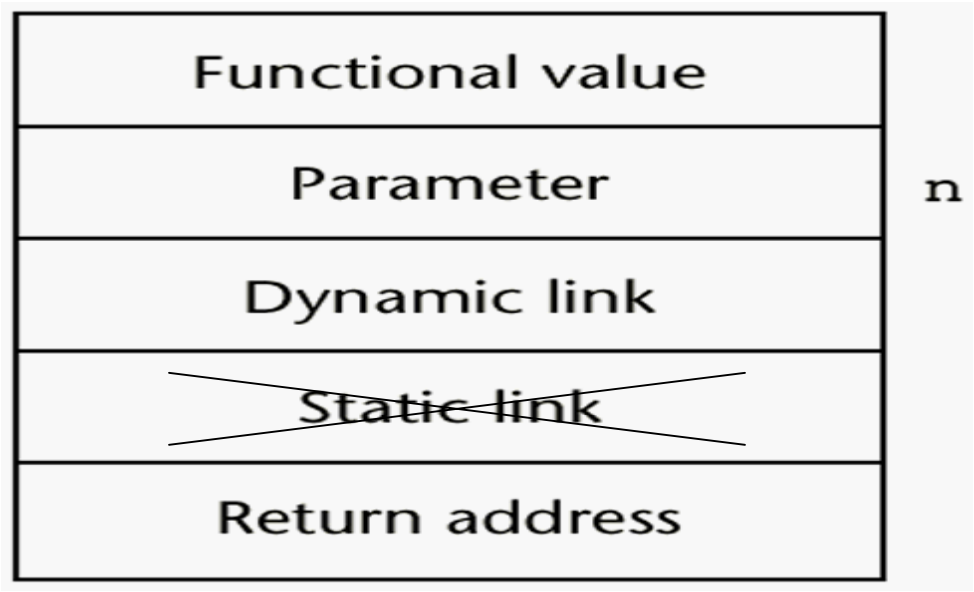
- The local_offset of a variable in an activation record can be determined at compile time, using the order, type, and sizes of variables declared in the subprogram associated with the activity record.

An Example With Recursion

The following example C program uses recursion to compute the factorial functions.

```
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
}
```

The activation record format has an additional entry for the returned value of the function



The Activation Records for the three times that execution reaches position 2

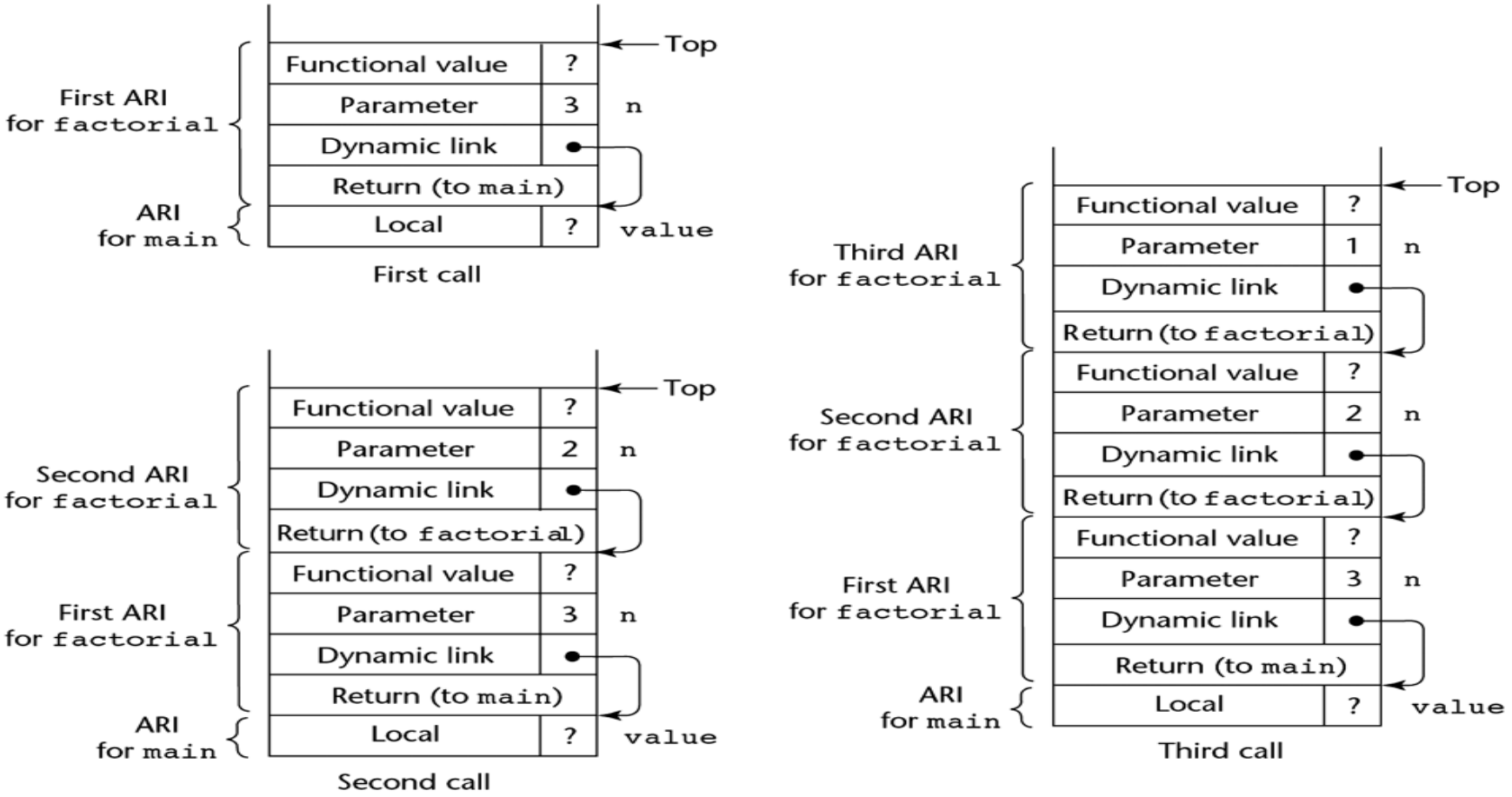
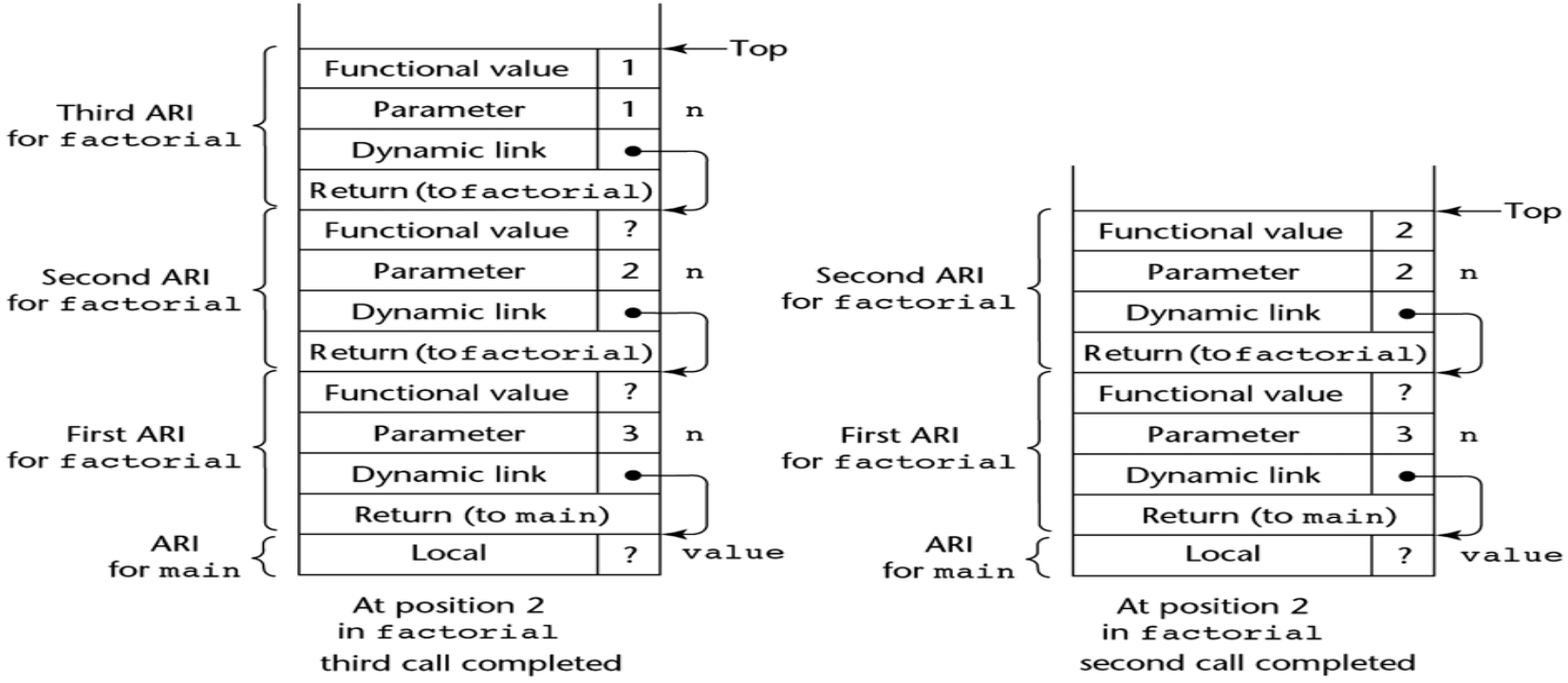


Figure 10.7

Stack contents at position 1 in `factorial`

Recall the code that the function multiplies the current value of the parameter n by the value returned by the recursive call to the function.

```
int factorial (int n) {
            <-----------------------------1
          if (n <= 1) return 1;
          else return (n * factorial(n - 1));
            <-----------------------------2
        }
```



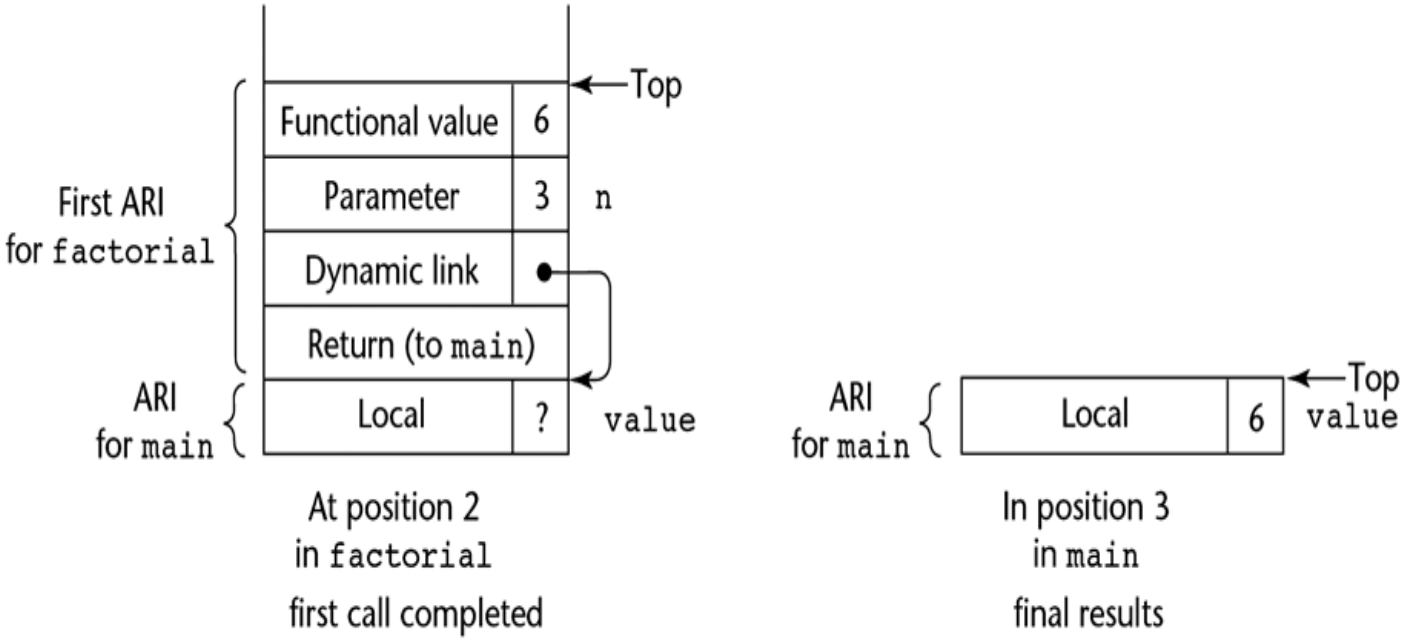


Figure 10.8

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