# Principles of Programming Languages(CS F301)


BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.

# Implementing Subprograms

## Ch.10 of T1; (covered Sec.1 to Sec. 3)


BITS Pilani
Hyderabad Campus

Prof R Gururaj

# Introduction

The subprogram *call* and *return* operations are together called *subprogram linkage*.

A subprogram call in a typical language has numerous actions associated with it.

The call process must include the implementation of whatever parameter passing method is used.

# Implementing subprograms

For simple subprograms without nesting, and with all variables static, it will be easy.

The semantics of a call to a simple subprogram requires the following.

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

(Last three actions of call are to be performed by caller.

Saving the state of caller can be done by either.)

The semantics of a return from a simple subprogram requires the following actions.

1. If there are pass-by-value-result or INOUT mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters.
2. If the subprogram is a function (returning a value), the functional value is moved to a place accessible to the caller.
3. The execution status of the caller is restored.
4. Control is transferred back to the caller.

( action 1,3, and 4 can be done by the called; action 2 can be done by either)

The call and return actions require storage space for the following.

1. Status info about the caller.
2. Parameters.
3. Return address.
4. Return value of functions.
5. Temporaries used by code of the subprograms.

The execution status includes everything needed to restore the execution of the caller.

Includes- registers, CPU status bits, Environment Pointer (EP)

The EP is used to access parameters and local variables during the execution of a program.

The linkage action for a subprogram occurs at the beginning of its execution is called the *prologue*.

And the actions that occur at the end is called as *epilogue*

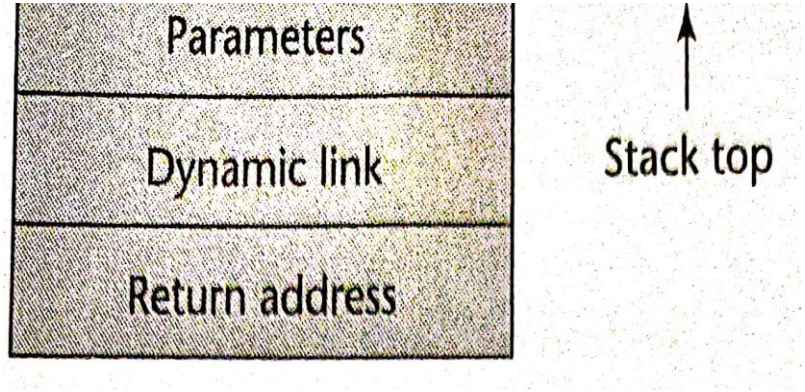
of subprogram linkage.

Simple subprograms – actual code, and local variables and parameters.

The format or layout of the non-code part of a subprograms is called an *Activation Record* (AR), because the data it describes are relevant only during the execution of subprogram. The concrete example of AR is called *Activation Record Instance* (ARI).

For simple subprograms it is of fixed size. Hence can be statically allocated.

## Activation record for simple subprograms.



# Implementing subprograms with stack-dynamic variables

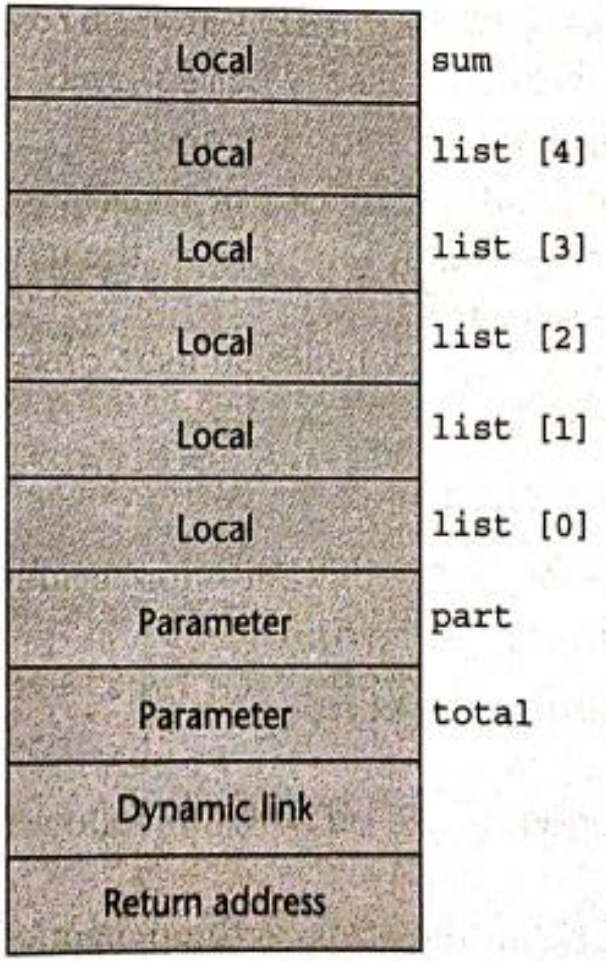


The stack dynamic variables can support recursion.

Linkage of subprograms with stack dynamic variables is complex when compared to simple subprograms, because:

1. Compiler must generate code to cause implicit allocation and deallocation of local variables.
2. Recursion adds the possibility of multiple simultaneous activations of the same subprogram, means that there can be more than one instance of a subprogram at a given time. The number of activations is limited by memory size of the machine. Each activation requires its activation record instance.

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




Activation record for sub()

*Dynamic link* in the stack is the pointer to the base of the activation record instance of the caller.

The format of the activation record is fixed at compile time.

This stack is the part of the runtime system, hence called the *runtime stack*.

Upon return from the subprogram, the stack top is set to the value of current EP minus one.

And the EP is set to the dynamic link for the activation record instance of the subprogram that has completed the execution.

Resetting the top of the stack effectively removes the top activation record instance.

A subprogram is active from the time it is called until the time that completes its execution.

At the time it becomes inactive the local scope ceases to exist and its referencing environment is no longer meaningful.

Therefore at that time its activation record instance can be destroyed.

The collection of dynamic links present in the stack at a given point of time is called the *dynamic chain* or *call chain*.

The reference to local variable is represented as offset from EP , and is called *local offset*.

The caller actions are as follows:

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.

The prologue actions of the called are as follows:

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

The epilogue actions of the called are as follows:

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. 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.
4. Restore the execution status of the caller.
5. Transfer control back to the caller.

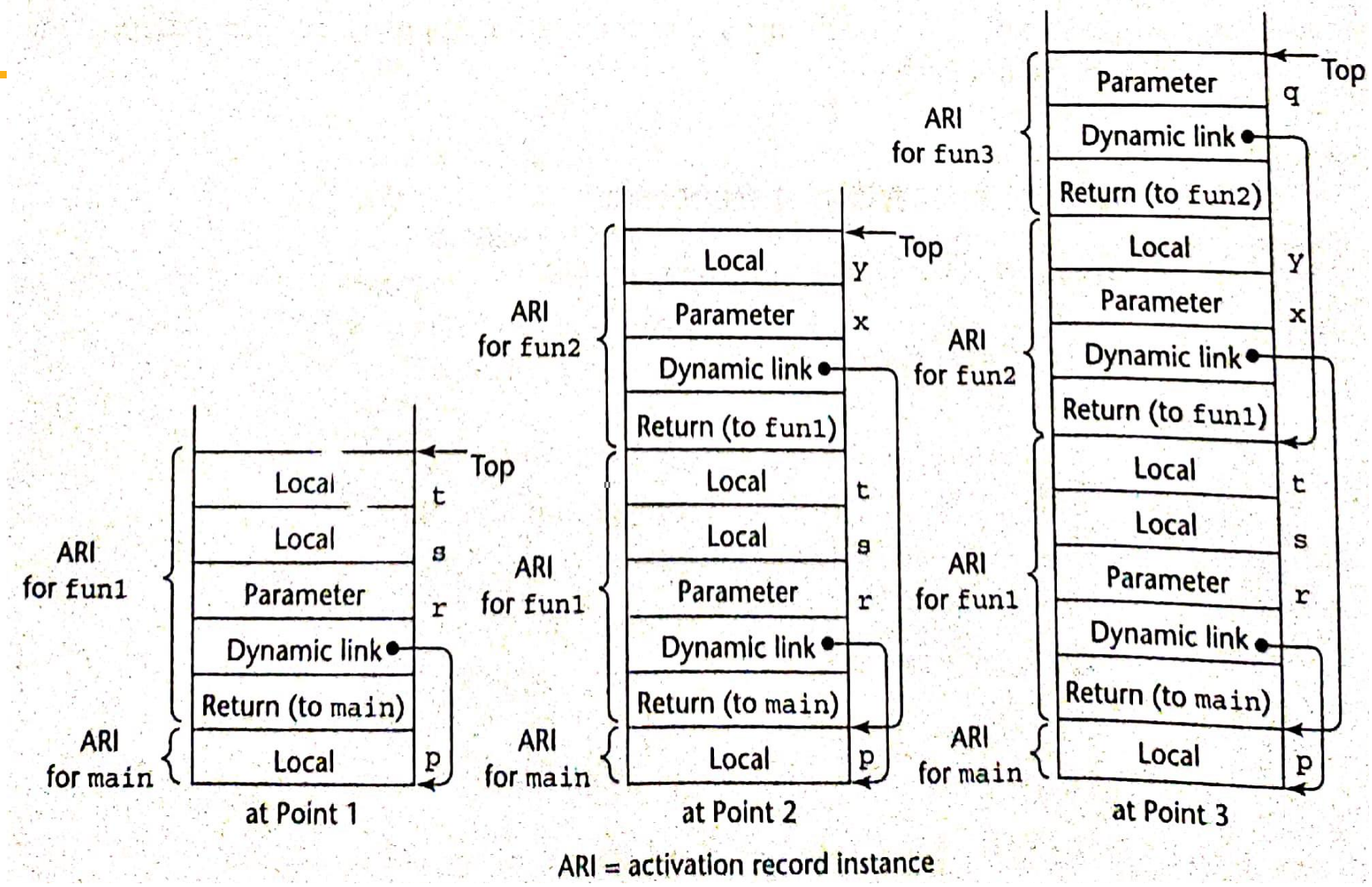
# An Example Without Recursion

Consider the following skeletal C program:

```
void fun1(float r) {
   int s, t;
   ...          <-------------- 1
   fun2(s);
   ...
}

void fun2(int x) {
   int y;
   ...          <-------------- 2
   fun3(y);
   ...
}

void fun3(int q) {
   ...          <-------------- 3
}

void main() {
   float p;
   ...
   fun1(p);
   ...
}
```

The sequence of function calls in this program is

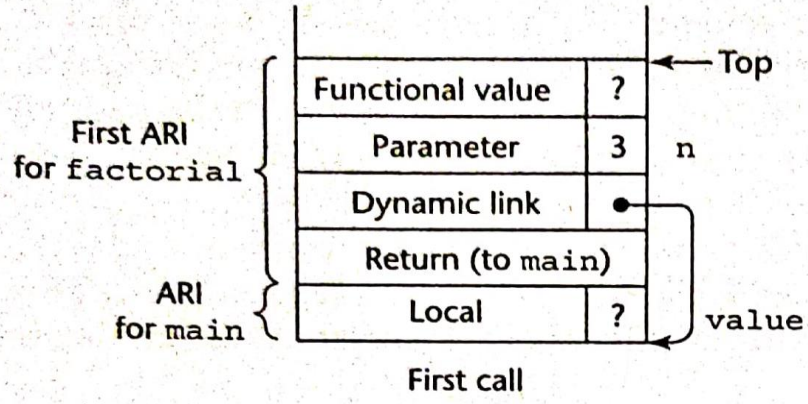
```
main calls fun1
fun1 calls fun2
fun2 calls fun3
```

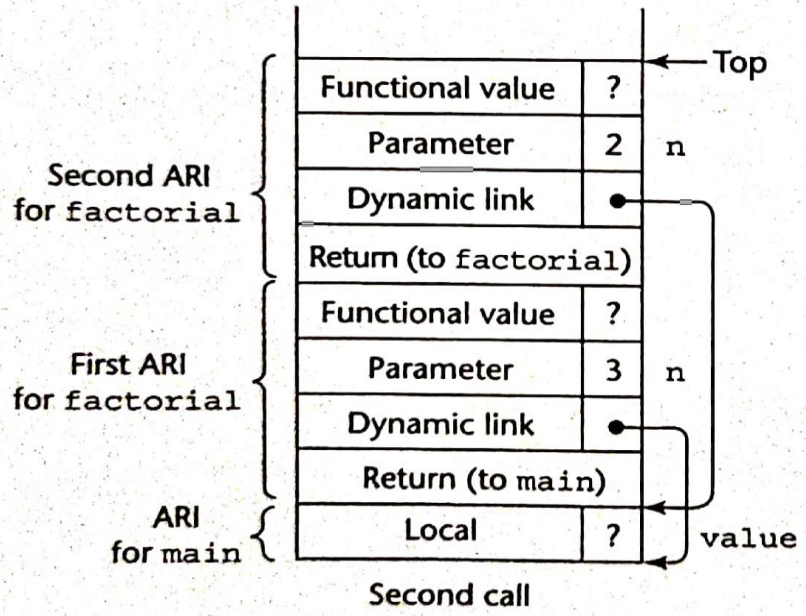

ARI for fun1
ARI for main
Local
Local
Parameter
Dynamic link
Return (to main)
Local
t
s
r
p
Top
at Point 1
ARI for fun2
ARI for fun1
ARI for main
Local
Parameter
Dynamic link
Return (to fun1)
Local
Local
Parameter
Dynamic link
Return (to main)
Local
y
x
t
s
r
p
Top
at Point 2
ARI for fun3
ARI for fun2
ARI for fun1
ARI for main
Parameter
Dynamic link
Return (to fun2)
Local
Parameter
Dynamic link
Return (to fun1)
Local
Local
Parameter
Dynamic link
Return (to main)
Local
q
y
x
t
s
r
p
Top
at Point 3
ARI = activation record instance

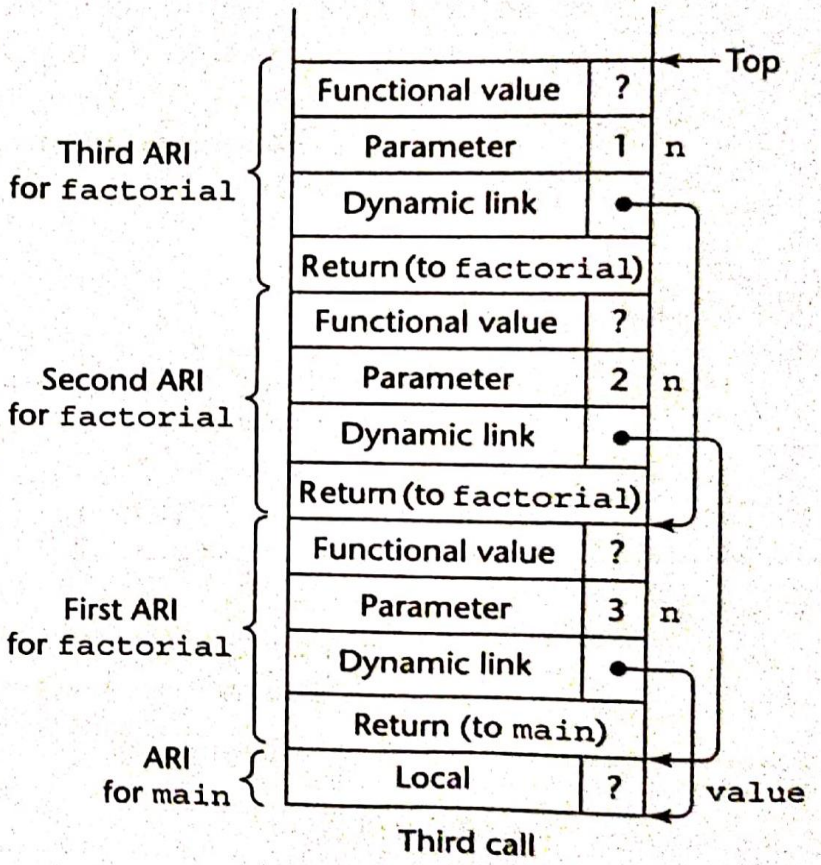
## Recursion

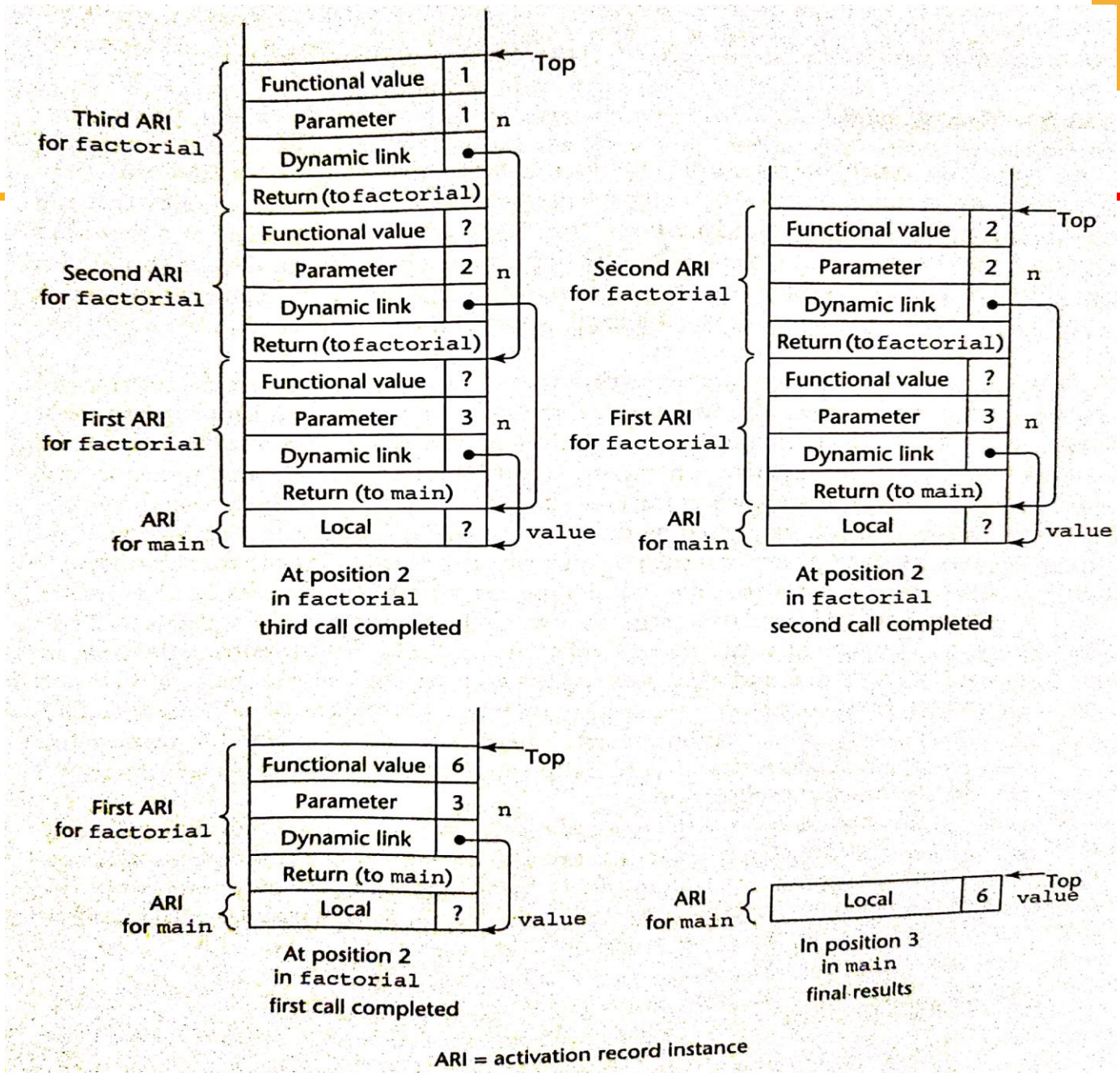
Consider the following example C program, which uses recursion to compute the factorial function:

```
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
}
```


Top
Functional value
?
Parameter
3
n
Dynamic link
Return (to main)
Local
?
value
First ARI for factorial
ARI for main
First call


Top
Functional value
?
Parameter
2
n
Dynamic link
Return (to factorial)
Functional value
?
Parameter
3
n
Dynamic link
Return (to main)
Local
?
value
Second ARI for factorial
First ARI for factorial
ARI for main
Second call


Top
Functional value
?
Parameter
1
n
Dynamic link
Return (to factorial)
Functional value
?
Parameter
2
n
Dynamic link
Return (to factorial)
Functional value
?
Parameter
3
n
Dynamic link
Return (to main)
Local
?
value
Third ARI for factorial
Second ARI for factorial
First ARI for factorial
ARI for main
Third call


Third ARI for factorial
Second ARI for factorial
First ARI for factorial
ARI for main
Functional value
Parameter
Dynamic link
Return (to factorial)
Local
Top
n
value
At position 2 in factorial third call completed
At position 2 in factorial second call completed
At position 2 in factorial first call completed
In position 3 in main final results
ARI = activation record instance

# Summary of Ch.10

## (covered Sec.1 to Sec. 3)



1. Introduction
2. Subprogram linkage
3. Implementing simple subprograms
4. Activation record instance
5. Implementing subprograms with stack-dynamic variables.
6. Call prologue/epilogue
7. Implementing recursive subprograms.