Topics

- The General Semantics of Calls and Returns
- Implementing "Simple" Subprograms
- Implementing Subprograms with Stack-Dynamic Local Variables

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

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

- Subprograms cannot be nested and all local variables are static
- Early versions of Fortran
- Call Semantics:
  - Save the execution status of the caller
  - Pass the parameters
  - Pass the return address to the called
  - Transfer control to the called

Implementing "Simple" Subprograms (continued)

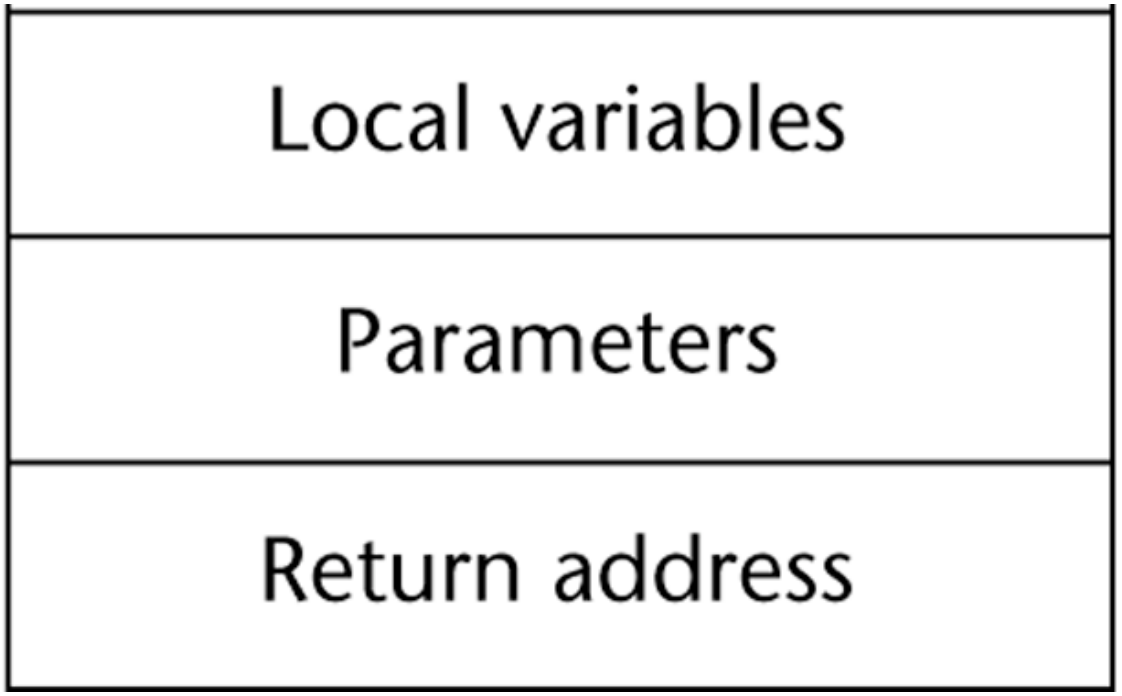
- Return Semantics:
  - If pass-by-value-result or pass-by-result parameters are used, move the current values of those parameters to their corresponding actual parameters
  - If it is a function, move the functional value to a place the caller can get it
  - Restore the execution status of the caller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

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

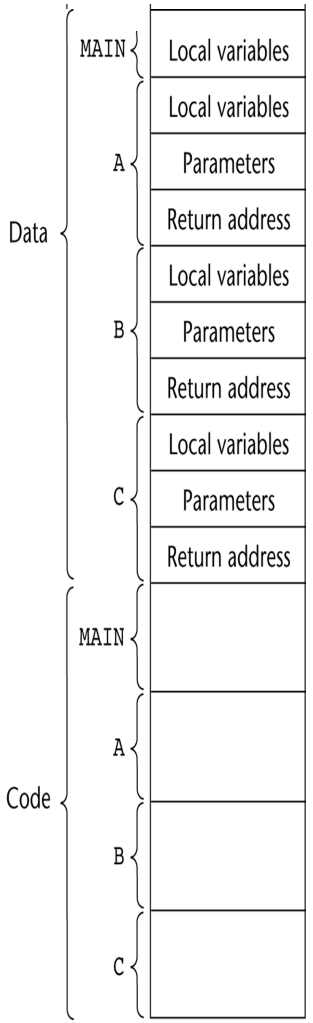
Implementing "Simple" Subprograms (continued)

- 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



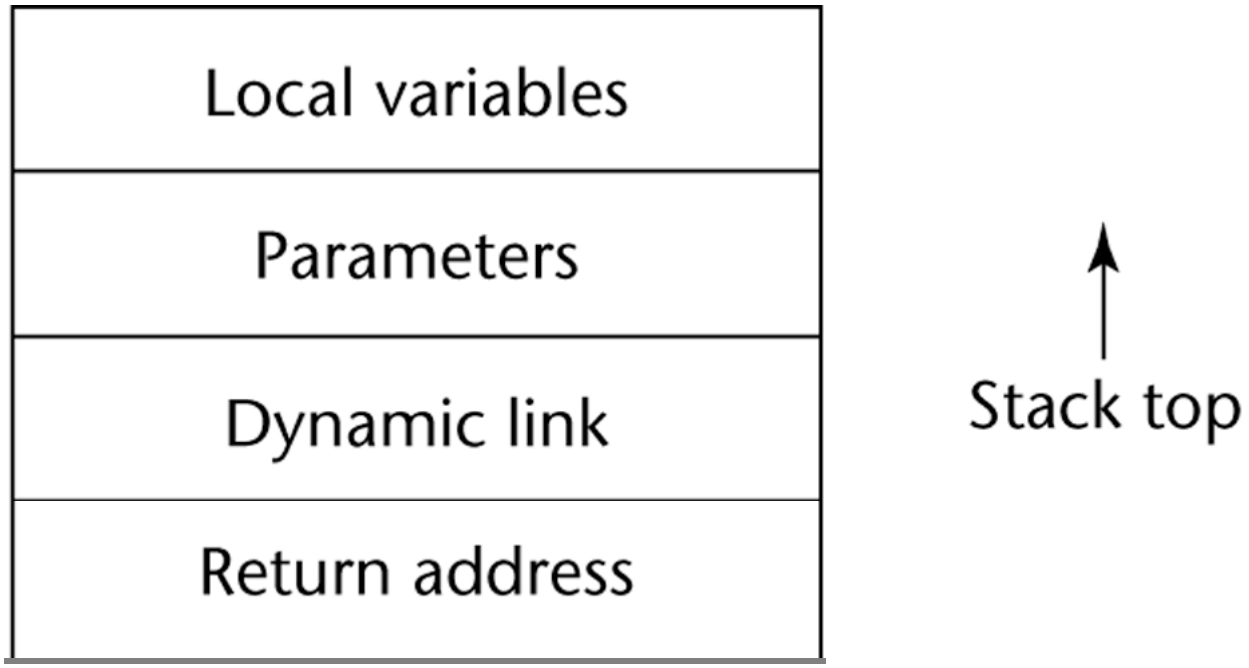
Code and Activation Records of a Program with "Simple" Subprograms



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 must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

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

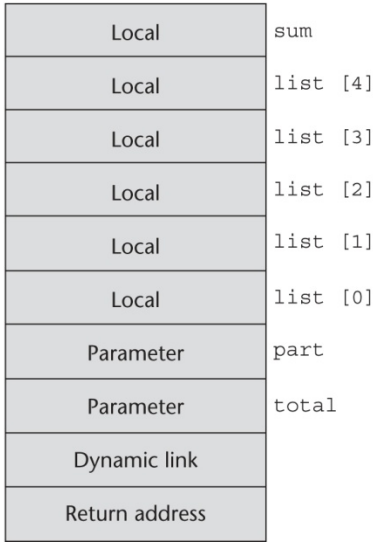


Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- 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:
  - Create an activation record instance
  - Save the execution status of the current program unit
  - Compute and pass the parameters
  - Pass the return address to the called
  - Transfer control to the called
- Prologue actions of the called:
  - Save the old EP in the stack as the dynamic link and create the new value
  - Allocate local variables

Revised Semantic Call/Return Actions (continued)

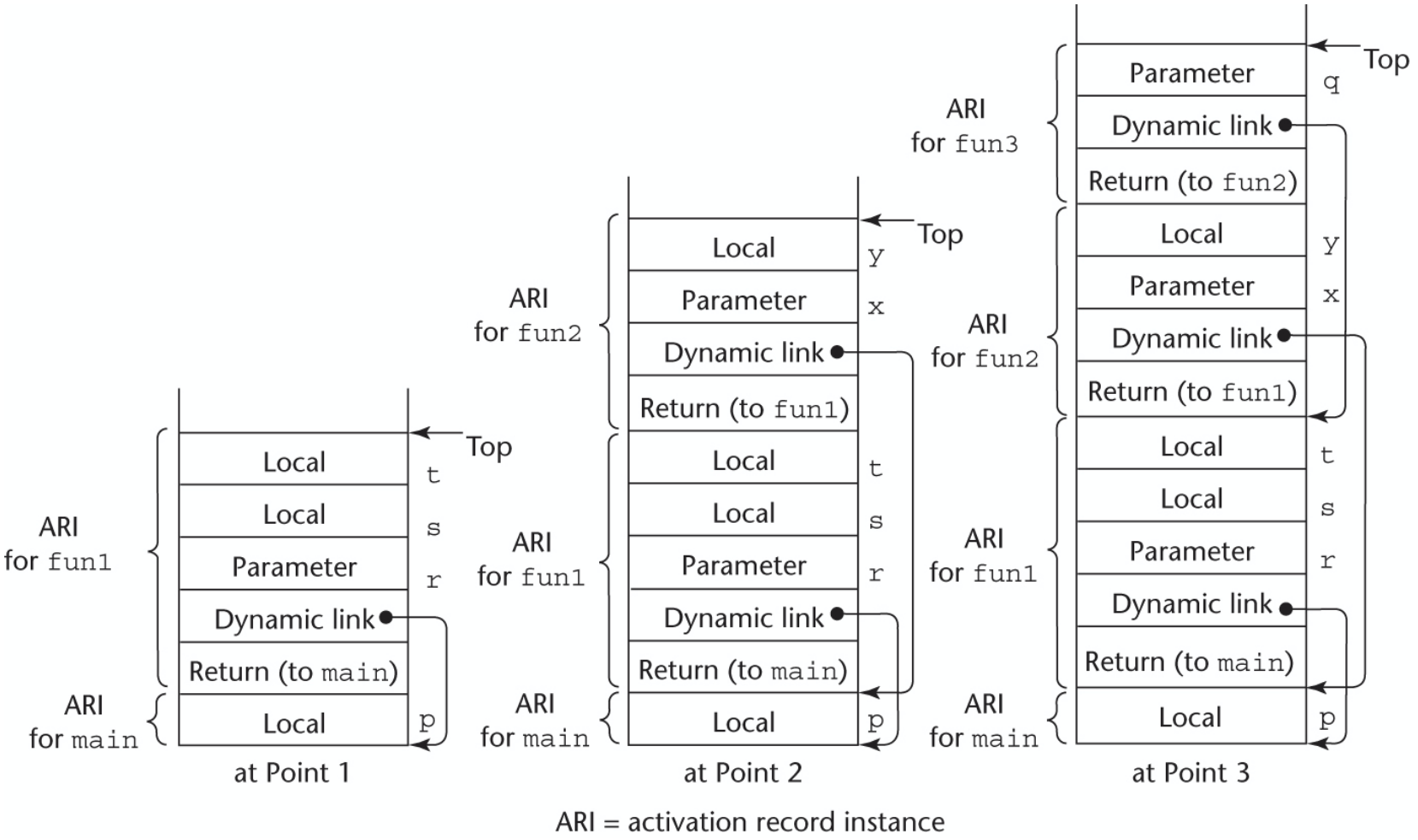
- Epilogue actions of the called:
  - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
  - If the subprogram is a function, its value is moved to a place accessible to the caller
  - Restore the stack pointer by setting it to the value of the current EP-1 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



Dynamic Chain and Local Offset

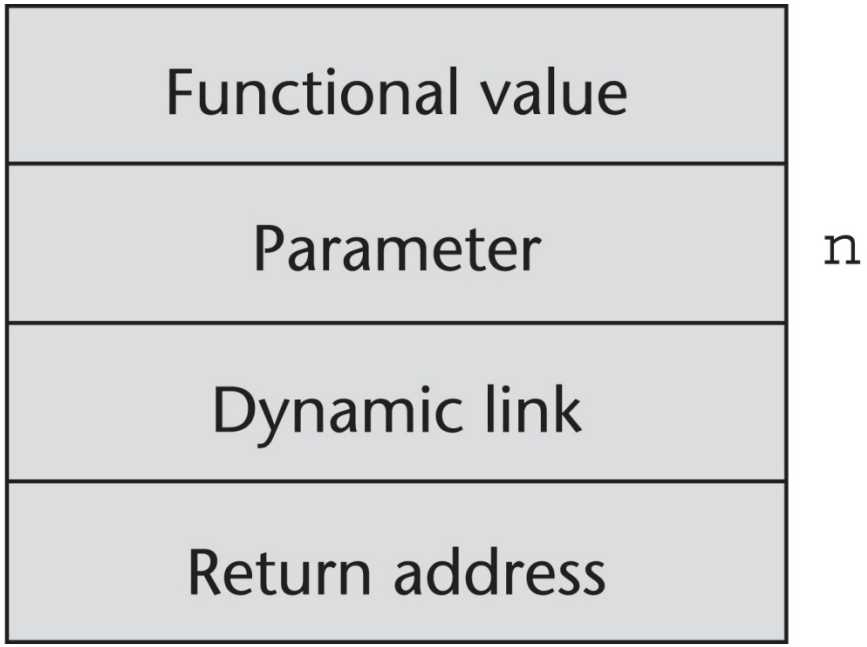
- The collection of dynamic links in the stack at a given time is called the *dynamic chain*, or *call chain*
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the *local_offset*
- The local_offset of a local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

An Example With Recursion

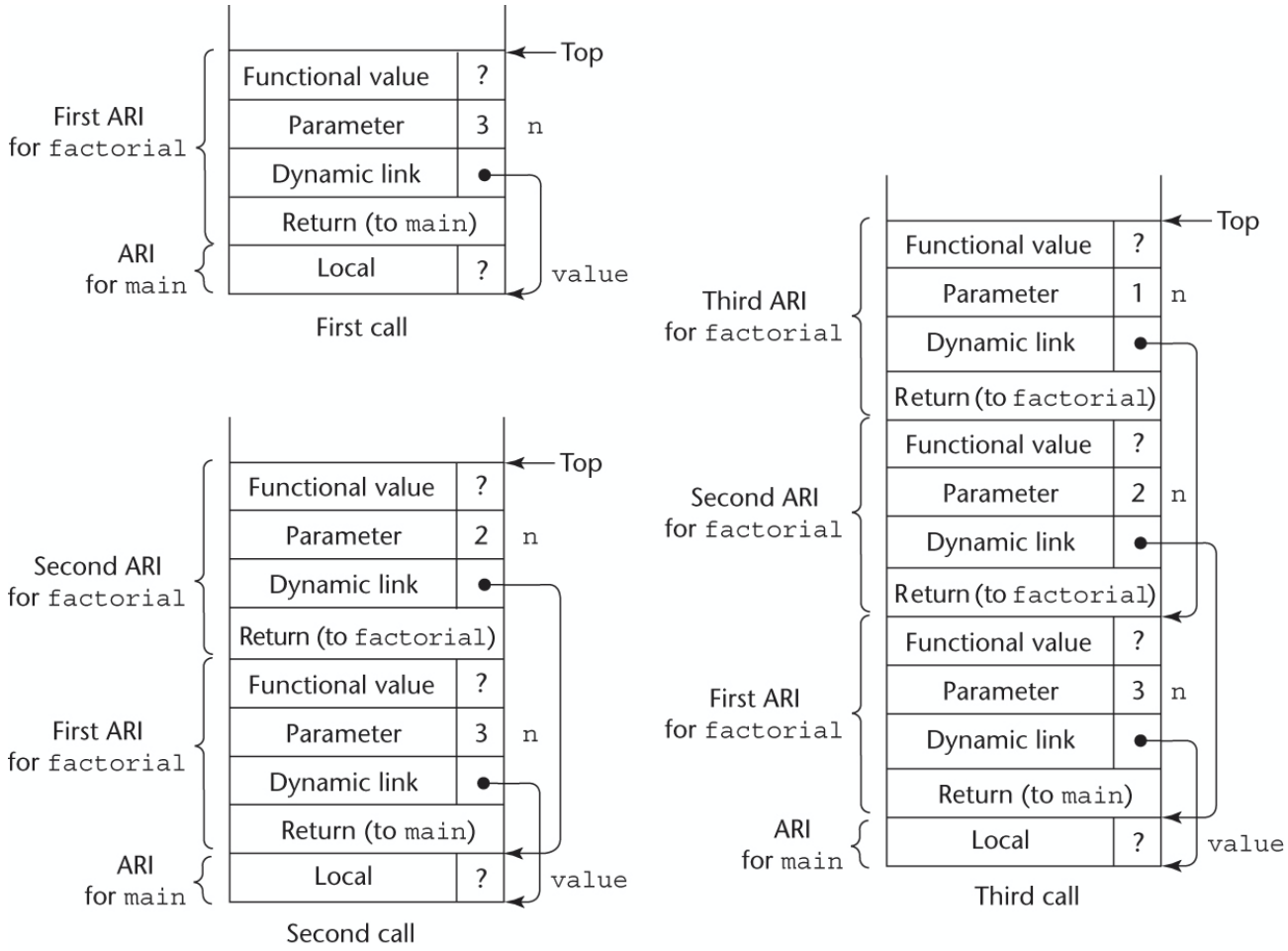
- The activation record used in the previous example supports recursion

```
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`



Stacks for calls to `factorial`



ARI = activation record instance

Stacks for returns from `factorial`