Implementing Subprograms

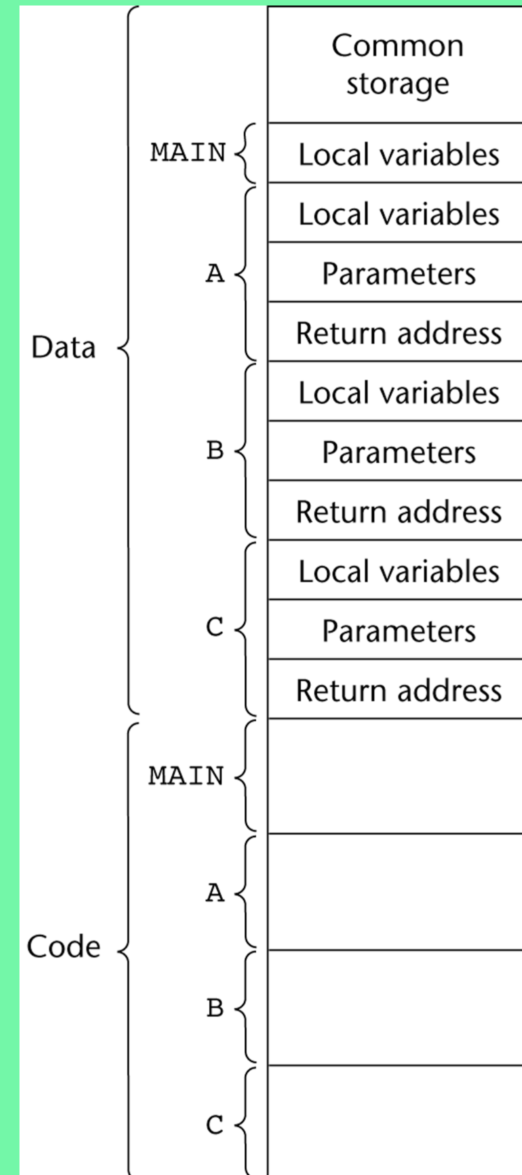
- What actions must take place when subprograms are called and when they terminate?
  - calling a subprogram has several associated actions:
    - calling subprogram’s environment must be saved
      - subprogram’s local variables, execution status return location
    - handle parameter passing to called subprogram
    - allocation and storage of local variables
    - transfer control to subprogram
  - subprogram termination then has the following actions:
    - return parameters that are out-mode
    - deallocation of local variables
    - return value from subprogram (if it is a function)
    - restore calling subprogram’s environment
    - transfer control to calling subprogram at the proper place
- Generically, this is known as *subprogram linkage* and in most languages, this is performed through the run-time stack by using *activation record instances*

FORTRAN I-77

- We first examine FORTRAN first since it is simplest
  - early FORTRAN had no recursion, all memory allocation was set up at compile time
    - so all variable and parameter names had specific memory locations known at compile time
  - subprogram call
    - save execution status of current program unit
    - carry out parameter passing
      - pass by copy required copying parameters
      - pass by reference required substituting an address for the value
    - pass return address to called subprogram
    - start execution of the subprogram
  - subprogram return
    - if pass-by-result parameters, copy current values to corresponding actual parameters and if subprogram is a function, value is moved to a place accessible by the caller
    - execution status of caller is restored
    - control is transferred back to caller

Activation Records (AR)

- In order to simplify the tasks of saving an environment and passing parameters
  - compiler generates ARs for each subprogram
- The AR for FORTRAN will store:
  - status of caller which include local variables
  - parameters
  - return address
  - functional value (for functions)
- The compiler sets this up providing storage space for each subprogram's AR
  - parameter passing is now merely a matter of copying values from one area of memory to another
  - transferring control is merely a matter of changing the PC based on subprogram start addresses or calling subprogram return addresses
  - memory access is efficient because all addresses are known at compile time

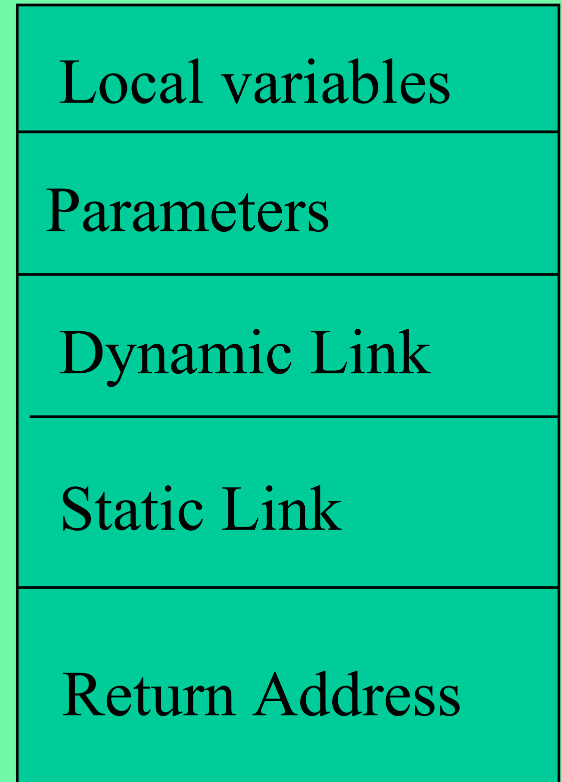


ALGOL-like Activation Records

- ALGOL used the run-time stack to permit recursion
  - the single activation record approach of FORTRAN does not permit multiple copies of local variables and parameters needed for recursion, and with only one copy, newer values would replace older values
  - the run-time stack, used in FORTRAN only to denote where to return to after subprograms terminate, will now also be used to store local variables and parameters
- The ALGOL approach was to have the compiler generate an AR for every subprogram
  - the AR is a *template*
  - upon subprogram call, an instance of the template is generated and pushed onto the run-time stack, an instance is called an activation record instance (ARI) and contains:
    - local variables
    - parameters
    - return location
    - return value (if a function)
    - links to connect to rest of run-time stack

ALGOL ARI

- Static Link
  - points to bottom of ARI for static parent
    - used for non-local variable access
- Dynamic Link
  - points to top of ARI of caller used to destroy current ARI end of subprogram
- Parameters
  - Store space for every parameter (whether a value or a pointer)
- Local variables store space for each local variable
  - stack-dynamic storage allocated at run-time but compile-time type checked
  - recursion available by creating an instance for each recursion call



NOTE: In C-languages, the static link will point to main's data since there are no other static parents

Example of ARI for C function

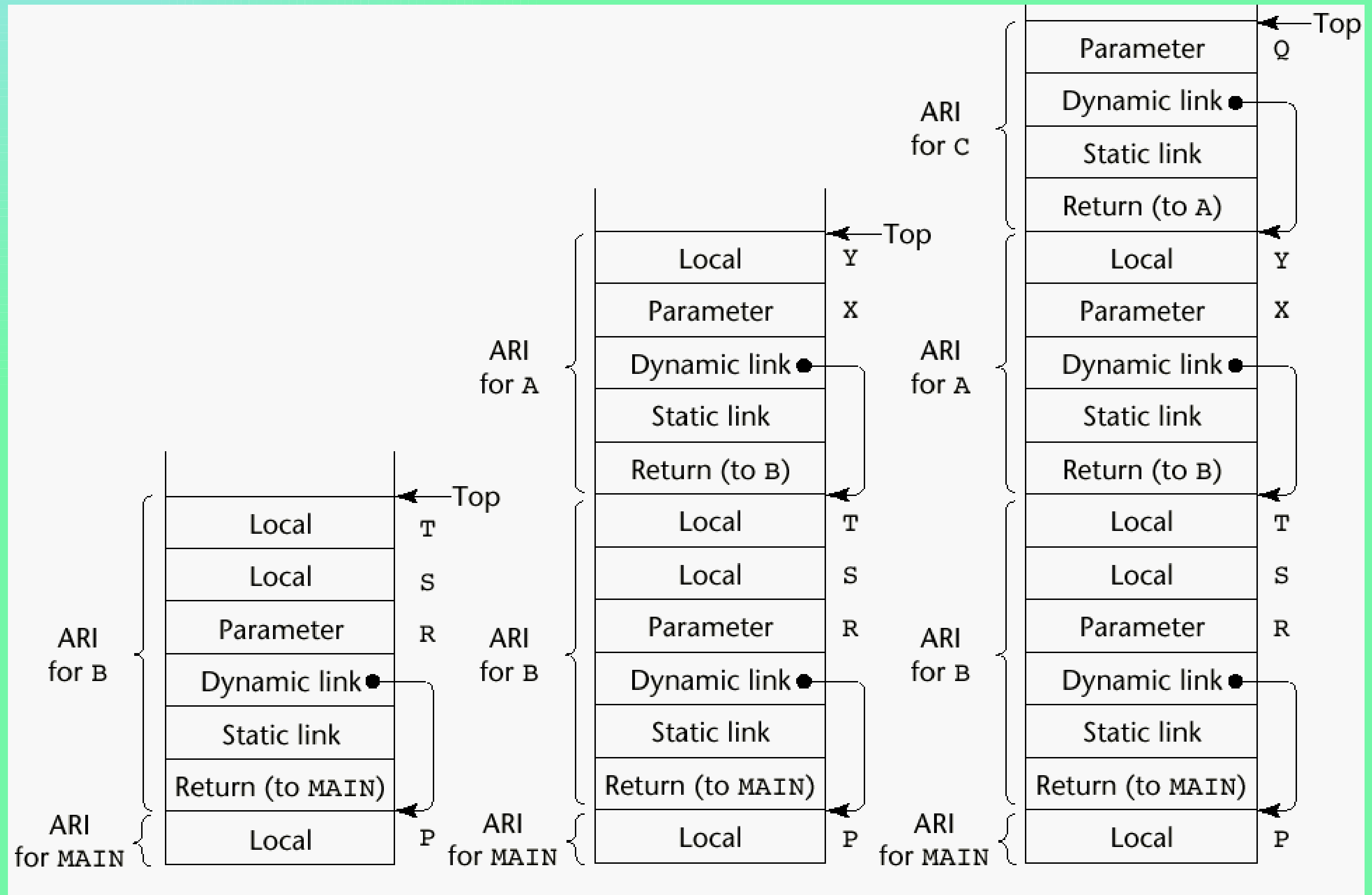
```
void sub(float total, int part)
{
    int list[4];
    float sum;
    …
}
```

Need space for
- 2 parameters (a float and an int)
- 2 variables, a 4-element int array and a float
- Return address (where in the code to return to when sub terminates)
- Dynamic link points to next ARI on stack so that this ARI can be popped off
- Static link points to static parent (main) for non-local variable references
- Since this is a void function, no space needed for return value

Local	sum
Local	list [5]
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Parameter	part
Parameter	total
Dynamic link	
Static link	
Return address	

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);
      ...
}
```

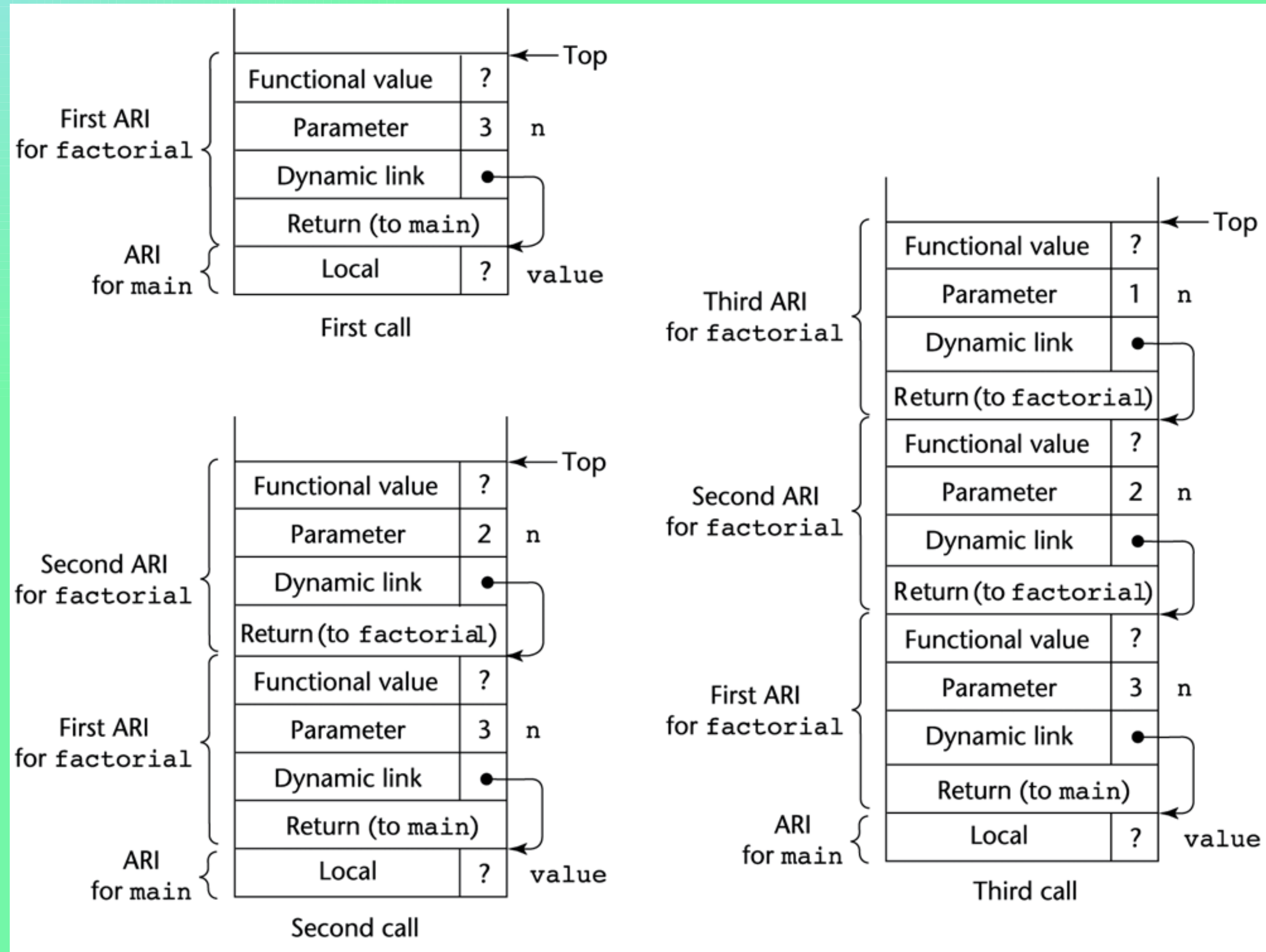


Stack after: main calls B B calls A A calls C

Example With Recursion (part I)

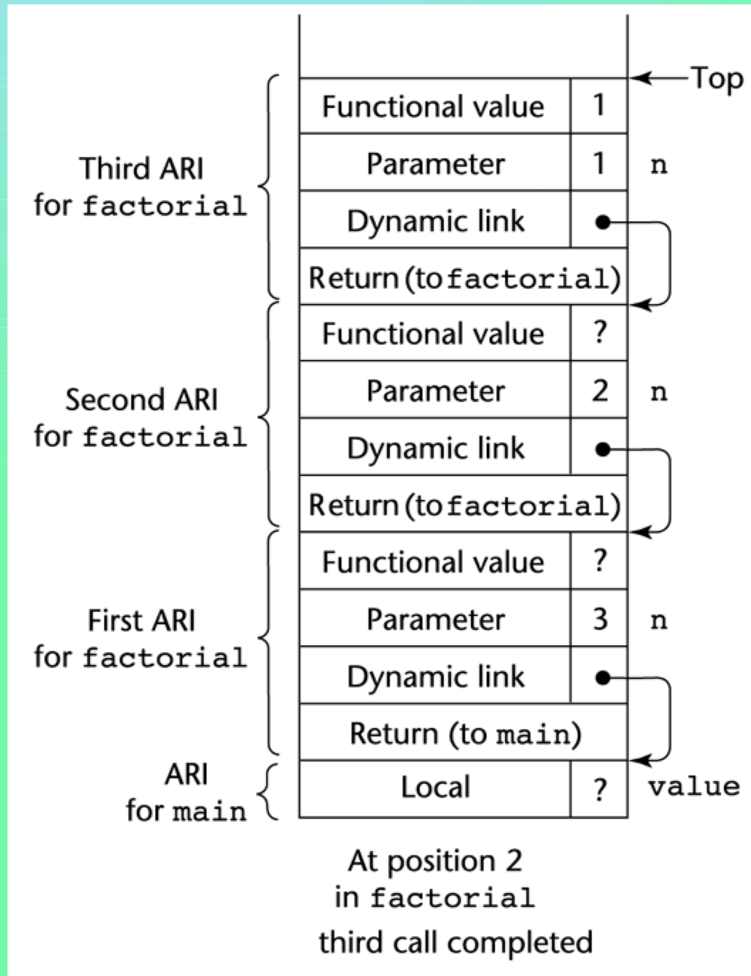
```
int factorial(int n)
{          <- Point 1
   if(n<=1) return 1;
   else return
        n*factorial(n-1);
}          <- Point 2

void main( )
{
   int value;
   value = factorial(3);
}          <- Point 3
```

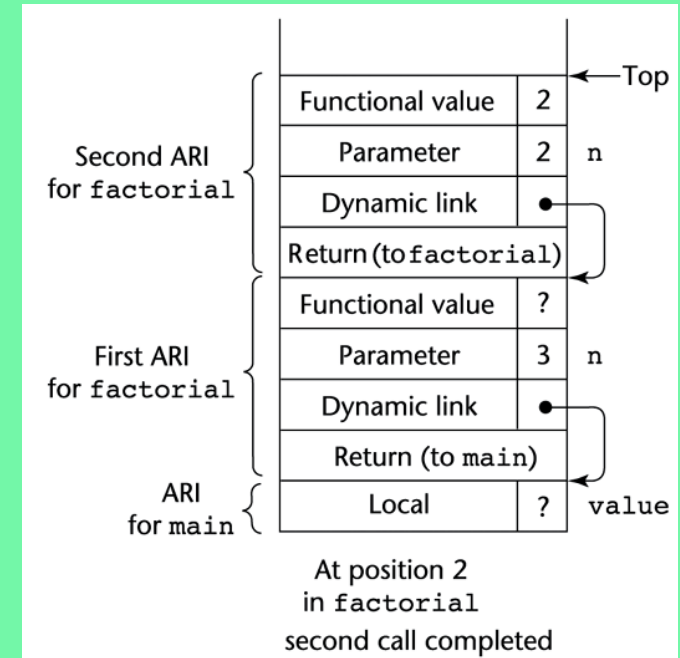


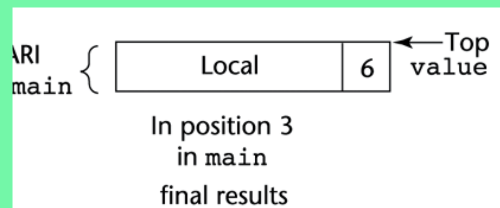
Stack contents at point 1 during each recursive call

Example With Recursion (part II)

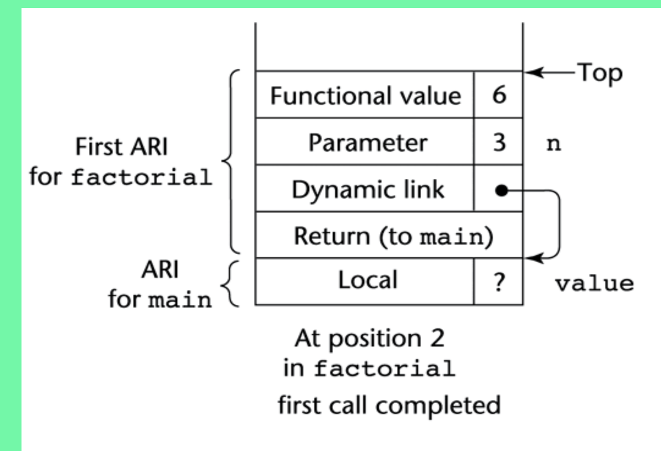


Stack contents at point 2 as each recursive call completes





Stack contents at point 3



Non Local Variable References

- Assume in some subprogram, a reference is made to a non-local variable
  - how do we determine what is being referenced?
  - non-local variables will be stored somewhere either in static memory (if the variable is global or declared static) or on the run-time stack
    - if on the run-time stack, which ARI do we check?
  - a top-down search through the ARIs would be inefficient, instead, the compiler can determine where the variable is stored using scope rules, and set up a pointer directly
    - recall in C/C++/Java, subprograms are not nested, so the only static parent a C function can have is main
    - but in non-C languages (notably Ada/Pascal-like languages), subprograms can be nested, the nestedness of the subprograms provides the information needed to find the non-local variable
- Two methods: Static Chains, Displays

Static Chains

- The compiler can determine for any given subprogram, which subprogram is it's static parent
  - the static link in the ARI points to this static parent
- Further, the compiler can determine how many static links must be taken to track down a given reference
  - for instance, assume
    - Main contains SubA – so SubA has a static link to Main
    - SubA contains SubB – so SubB has a static link to SubA
    - SubB has a static parent of subA which has a static parent of Main
  - now assume that Main has declared a variable x and neither SubA nor SubB has declared a variable x
    - if SubB references x, then x is found by following 2 links to reach Main on the run-time stack
- A static chain is the information needed to resolve the reference and consists of:
  - chain offset – the distance in terms of number of static links
    - this is equivalent to the nestedness between the reference and where it is declared using static scope rules
  - local offset – the position in the ARI of the variable (starting from the bottom of this ARI)