

## Ch.9: Implementing Subprograms

### I. General Semantics of Calls and Returns

#### A. What the *caller* of a subprogram must do:

1. Include the mechanism for the parameter-passing method used
2. Allocate storage on the stack for non-static local variables
3. Save the execution status of the calling program unit
4. Transfer control to called program and back again at end of subprogram
5. Provide a mechanism to access non-local variables if static scoping is used

#### B. What the *called* subprogram must do:

1. Copy values of out-mode parameters on the return
2. Deallocate storage used for local variables (move the return address and adjust stack pointer)
3. Return control to the calling program

### II. Implementing FORTRAN 77 Subprograms

#### A. Simplifying characteristics:

1. Non-local variables accessed through COMMON
2. No recursion
3. Subprogram variables statically declared

#### B. Semantics of a subprogram call

1. Save the execution status of the current program unit (Registers)
2. Carry out the parameter-passing process
3. Push the return address on the run-time stack
4. Transfer control to the callee (Load subprogram starting address into PC)

#### C. Semantics of a subprogram return:

1. Copy parameter values back to caller if pass-by-value-result parameters are used
2. If the subprogram is a function, make the resulting functional value accessible to caller
3. Restore caller's execution status (Registers)
4. Transfer control back to the caller (move the return address into the PC)

#### D. To implement subroutine calls & returns, storage on the stack is required for caller status (registers), parameters, return address, function result value. The layout for these items is called an *activation record*, since the data items are relevant during the activation of a subprogram.

#### E. When a program runs, there could be multiple *instances* of the *activation record* for a subprogram (e.g. recursion), though not in FORTRAN 77

#### F. FORTRAN 77 example: [See overhead]

1. Since there can only be one active version of a subprogram at a time, the activation records for all the functions can be allocated in memory statically.
2. Pieces can be compiled separately and put together by a linker/loader in order to run it.

### III. Implementing Subprograms in ALGOL-like Languages

#### A. More complex than in FORTRAN 77 because:

1. Parameters can be passed by reference, not just value
2. Subprogram local variables may be dynamically allocated
3. Recursion introduces the possibility of multiple activation records for a single subprogram at one time.
4. Static scoping used to access nonlocal variables

#### B. A typical activation record now includes:

1. *static link*: points to the bottom of the activation record instance of an activation of the static parent. It is used to access non-local variables
2. *dynamic link*: points to the top of an instance of the activation record of the caller. This is used in the destruction of the current activation record instance when the subprogram is completed, as there may have been other allocations on the stack.

- C. The order of items on the activation record is significant: return address, static & dynamic links, parameters, local variables. (This may be a bit different in practice...)
- D. See example of a simple activation record on the [overhead]
- E. An Example without recursion and nonlocal references [overhead]
- F. An Example *with* recursion [overhead]
- G. Mechanisms for Implementing Nonlocal References
  - 1. Background:
    - a. A two-step process:
      - (1). Find the instance of the activation record
      - (2). Use the local offset of the variable within the activation record instance.
    - b. Note that a procedure is callable only when all of its static ancestor program units are active.
    - c. As in the case of recursion, the parent scope activation record need not be adjacent to the child's activation record.
  - 2. Static Chain: a chain of static links connecting certain activation record instances in the stack
    - a. During execution, the static link of an ARI (Activation Record Instance) P points to an ARI of P's static parent unit. This continues for the parent, etc., giving a static chain connecting all the static ancestors of P.
    - b. At compile time the compiler can determine not only the length of the chain needed to access a nonlocal variable, but the offset within the ARI as well. This is done as follows:
      - (1). *Static depth* is the depth of nesting from the outermost scope, with the main program being depth 0.
      - (2). The *nesting depth* (a.k.a. *chain offset*) between a calling procedure and the called variable is the difference between the static depth of the calling procedure and the static depth of the procedure containing the declaration for the called variable.
      - (3). The local offset within an ARI can be determined at compile time as well, so actual variable names themselves need not be stored in the ARI
      - (4). These concepts are illustrated in the [overheads]
      - (5). Note that the (chain,local) offsets are determined statically, as the compiler keeps track of nesting information to verify scopes as compilation progresses.
    - c. Returning from a subprogram means simply removing the subprogram's ARI from the stack
    - d. Calling a subprogram means the most recent activation record instance of the parent scope must be found.
      - (1). This situation can be treated the same as non-local variable references explained above, giving a compile-time computed `static_depth` to be traversed when the subroutine is called at run-time. This traversal will give the ARI of the static scope parent to which the new subroutine's ARI's static link should point.
      - (2). To create the static links, first find the static distance (chain offset) between the *caller* and the *parent scope* of the subprogram being called. Then set the static link of the ARI of the subprogram being called to be the caller's ARI + the chain offset.
    - e. Problem: references to variables beyond the static parent are costly, since a potentially lengthy chain may need to be followed. (e.g. when there are many nested levels and a referenced variable is at the outermost level - the chain must be followed all the way.)

### 3. Displays

- a. Static links are collected in a single array, where at any one time the display contains a list of the accessible ARI's in the stack in the order in which they are nested.
- b. This gives essentially an ARI "look-up" table, which avoids the need to follow a static link chain. The `static_depth` of a subroutine corresponds exactly to its position in the display.
- c. The display contains only links for the subroutine currently executing and its static ancestors.
- d. There are three cases for adding an ARI, where we compare the caller's static depth (`sdcaller`) and the callee's static depth (`sdcallee`)
  - (1). `sdcaller = sdcallee` [Figure 9.10]
  - (2). `sdcaller > sdcallee` [Fig. 9.14] Note that the ARI for SUB3 need not be removed, as SUB1 cannot access SUB3's data because it is not in its scope.
  - (3). `sdcaller < sdcallee`
    - (a). [Figure 9.11] shows the generic situation, where a new ARI is simply added to the display, overwriting any previous value at that entry in the display
    - (b). Suppose SUB1 contains a nested Proc. SUB4 [overhead]
    - (c). In some cases it is necessary to save the old ARI. Consider the situation developing in [Fig. 9.12] and then [Fig. 9.13]. Note that successive calls to {SUB3, SUB1, SUB4, SUB2} in [Fig. 9.13] would require saving the old ARI addresses as the stack grows. These old ARI addresses would need to be stored somehow either on the run-time stack, or there would need to be some sort of stack associated with each display entry. (E.g. add an extra field to each ARI to store old display value).

### 4. Comparison between chains and displays

- a. For nonlocals more than one static level away, displays are faster
- b. Overall, displays are better if there are many distant nonlocal references
- c. Experiments indicate that nesting levels rarely exceed three in practice.

## IV. Blocks

- A. These may be implemented as parameterless procedures called from a fixed location in the program.
- B. Blocks may also be implemented by statically allocating space for block variables after local variables in the ARI [overhead]

## V. Implementing Dynamic Scoping

Deep and shallow *binding* have different semantics; deep and shallow *access* have the same semantics. (Remember: deep and shallow *binding* are choices to determine the referencing environment for passed subprograms.)

- A. Deep Access: references to nonlocals are resolved by following dynamic chain [see overhead]
  1. The chain must be traversed at run-time; There is no way chain length can be determined at compile time
  2. Names of variables must be stored in ARI's so matching can be done as chain is searched
- B. Shallow Access: have a separate stack for each variable name in a program. The top of the stack stores the current value for each.

## VI. Implementing Parameters that are Subprogram Names

### A. Static Chaining

-The caller and callee must share a static ancestor, which should be the static parent of the callee. The address of this static parent is then passed along with the subprogram name, where this address is then used to set the static link in the callee's ARI.

### B. Displays

- It is possible that the entire display must be stored and replaced, as the callee could have an entirely different path of static ancestors.