Chapter 9

Def: The subprogram call and return operations of a language are together called its *subprogram linkage*

Implementing FORTRAN 77 Subprograms

Call Semantics:
1. Save the execution status of the caller
2. Carry out the parameter-passing process
3. Pass the return address
4. Transfer control to the callee

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

Required Storage:
- Status information of the caller, parameters, return address, and functional value (if it is a function)

Chapter 9

The format, or layout, of the noncode 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)

FORTRAN 77 subprograms can have no more than one activation record instance at any given time

The code of all of the program units of a FORTRAN 77 program may reside together in memory, with the data for all units stored together elsewhere

The alternative is to store all local subprogram data with the subprogram code

Implementing Subprograms in ALGOL-like Languages

- *This is more complicated than implementing FORTRAN 77 subprograms because:*
  - Parameters are often passed by two methods
  - Local variables are often dynamically allocated
  - Recursion must be supported
  - Static scoping must be supported

Chapter 9

- A typical activation record for an ALGOL-like language:

Local variables
Parameters
Dynamic link
Static link
Return address

stack top
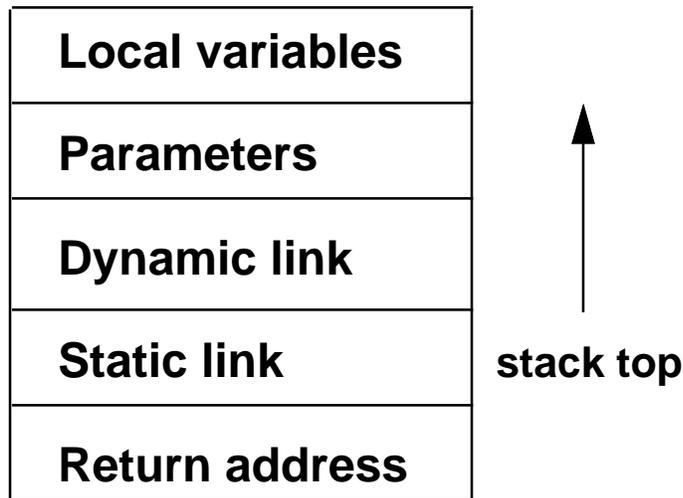
- The activation record format is static, but its size may be dynamic

- The static link points to the bottom of the activation record instance of an activation of the static parent (used for access to nonlocal vars)

- 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

Chapter 9

- An example program:

```
program MAIN_1;
  var P : real;
  procedure A(X : integer);
    var Y : boolean;
    procedure C(Q : boolean);
      begin { C }
      ... <--------------------------3
      end; { C }
    begin { A }
    ... <--------------------------2
    C(Y);
    ...
    end; { A }
    procedure B(R : real);
      var S, T : integer;
      begin { B }
      ... <------------------------1
      A(S);
      ...
      end; { B }
  begin { MAIN_1 }
  B(P);
  end. { MAIN_1 }
```

Note that: MAIN_1 **calls** B
          B **calls** A
          A **calls** C

---> **SHOW FIGURE 9.5 (p. 385)**

Chapter 9

- 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. This offset is called the *local_offset*

- The local_offset of a local variable can be determined by the compiler
  - Assuming all stack positions are the same size, the first local variable declared has an offset of three plus the number of parameters

  e.g. In MAIN_1, the local_offset of Y in A is 4

- The activation record used in the previous example supports recursion

```
e.g. 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
     }
```

Chapter 9

--> **SHOW FIGUREs 9.7 and 9.8 (p. 387 and p. 388)**

Nonlocal References - Static Scoping

Observation: **All variables that can be nonlocally accessed reside in some activation record instance in the stack**

The process of locating a nonlocal reference:

1. **Find the correct activation record instance**
2. **Determine the correct offset within that activation record instance**

Finding the offset is easy!

Finding the correct activation record instance:
 - **Static semantic rules guarantee that all nonlocal variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made**

Chapter 9

Technique 1 - Static Chains

A *static chain* is a chain of static links that connects certain activation record instances

The static link in an activation record instance for subprogram A points to one of the activation record instances of A's static parent

The static chain from an activation record instance connects it to all of its static ancestors

To find the declaration for a reference to a nonlocal variable:

You could chase the static chain until the activation record instance (ari) that has the variable is found, searching each ari as it is found

Def: *Static_depth* is an integer associated with a static scope whose value is the depth of nesting of that scope

Chapter 9

e.g.   main ----- static_depth = 0
            A   ----- static_depth = 1
              B ----- static_depth = 2
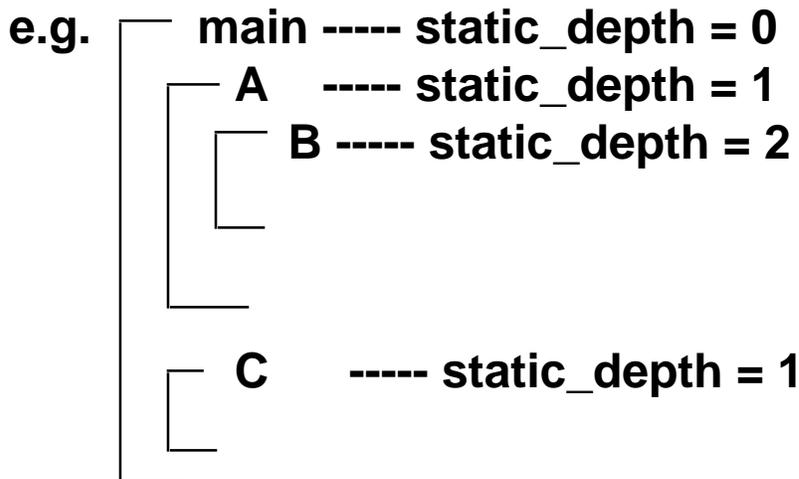
          C   ----- static_depth = 1

Def: The *chain_offset* or *nesting_depth* of a nonlocal reference is the difference between the static_depth of the reference and that of the scope where it is declared

A reference can be represented by the pair:

(chain_offset, local_offset)

where local_offset is the offset in the activation record of the variable being referenced

Chapter 9

```
Example Program:

program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
      A := B + C;  <----------------------1
      end;  { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        SUB1;
        E := B + A:   <--------------------2
        end; { SUB3 }
      begin { SUB2 }
      SUB3;
      A := D + E;  <----------------------3
      end; { SUB2 }
    begin { BIGSUB }
    SUB2(7);
    end; { BIGSUB }
  begin
  BIGSUB;
  end. { MAIN_2 }
```

Chapter 9

Call sequence for `MAIN_2`
  `MAIN_2` **calls** `BIGSUB`
  `BIGSUB` **calls** `SUB2`
  `SUB2` **calls** `SUB3`
  `SUB3` **calls** `SUB1`

`--->` **SHOW FIGURE 9.9 (p. 392)**

At position 1 in `SUB1`:
  `A` **- (0, 3)**
  `B` **- (1, 4)**
  `C` **- (1, 5)**

At position 2 in `SUB3`:
  `E` **- (0, 4)**
  `B` **- (1, 4)**
  `A` **- (2, 3)**

At position 3 in `SUB2`:
  `A` **- (1, 3)**
  `D` **- an error**
  `E` **- (0, 5)**

Chapter 9

Static Chain Maintenance

At the call (assume there are no parameters that are subprograms and no pass-by-name parameters:
   - The activation record instance must be built
   - The dynamic link is just the old stack top pointer
   - The static link must point to the most recent ari of the static parent (in most situations)

 Two Methods:

   1. Search the dynamic chain until the first ari for the static parent is found--easy, but slow

   2. Treat procedure calls and definitions like variable references and definitions (have the compiler compute the nesting depth, or number of enclosing scopes between the caller and the procedure that declared the called procedure; store this nesting depth and send it with the call)

      e.g. Look at `MAIN_2` and Figure 9.9
         At the call to `SUB1` in `SUB3`, this nesting depth is 2, which is sent to `SUB1` with the call. The static link in the new ari for `SUB1` is set to point to the ari that is pointed to by the second static link in the static chain from the ari for `SUB3`

Chapter 9

Evaluation of the Static Chain Method

Problems:

1. A nonlocal reference is slow if the number of scopes between the reference and the declaration of the referenced variable is large

2. Time-critical code is difficult, because the costs of nonlocal references are not equal, and can change with code upgrades

Technique 2 - Displays

The idea: Put the static links in a separate stack called a *display*. The entries in the display are pointers to the ari's that have the variables in the referencing environment.

Represent references as

(display_offset, local_offset)

where display_offset is the same as chain_offset

Chapter 9

Mechanics of references:

- Use the display_offset to get the pointer into the display to the ari with the variable

- Use the local_offset to get to the variable within the ari

Display maintenance (assuming no parameters that are subprograms and no pass-by-name parameters)

- Note that display_offset depends only on the static depth of the procedure whose ari is being built. It is exactly the static_depth of the procedure.

- There are k+1 entries in the display, where k is the static depth of the currently executing unit (k=0 is for the main program)

- For a call to procedure P with a static_depth of k:
  - a. Save, in the new ari, a copy of the display pointer at position k
  - b. Put the link to the ari for P at position k in the display

Chapter 9

At an exit, move the saved display pointer from the ari back into the display at position k

To see that this works:

- Let Psd be the static_depth of P, and
    Qsd be the static_depth of Q
  - Assume Q calls P

- There are three possible cases:
  1. Qsd = Psd
  2. Qsd < Psd
  3. Qsd > Psd

Example skeletal program:
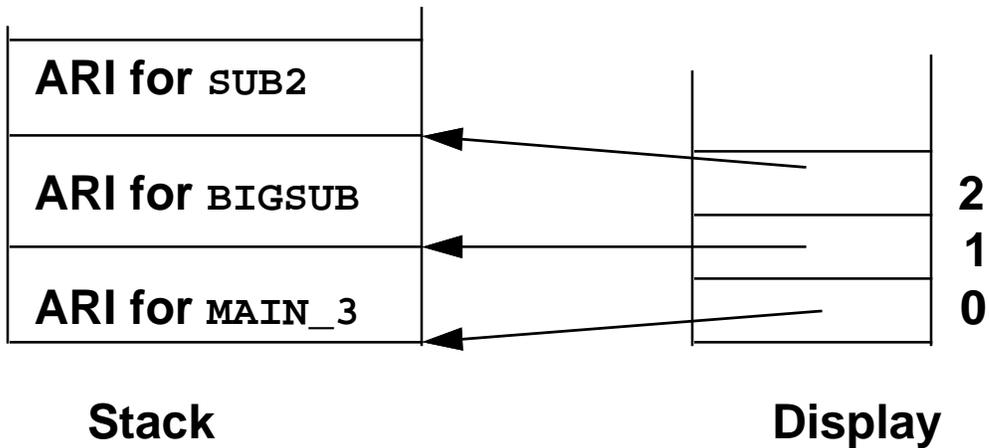
```
program MAIN_3;
   procedure BIGSUB;
      procedure SUB1;
         end; { SUB1 }
      procedure SUB2;
         procedure SUB3;
            end; { SUB3 }
         end; { SUB2 }
      end; { BIGSUB }
   end. { MAIN_3 }
```
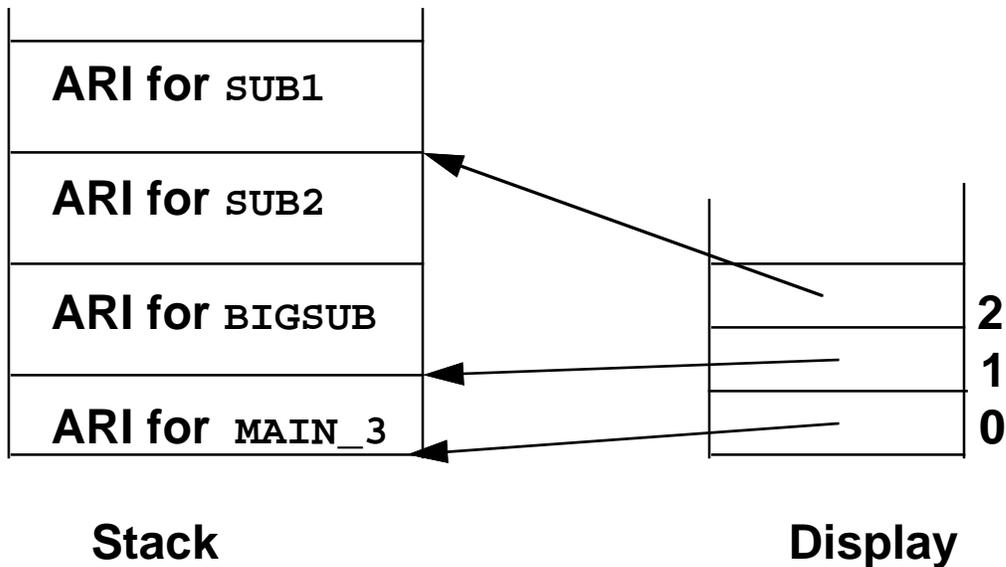
`MAIN_3` can illustrate all three cases

Chapter 9

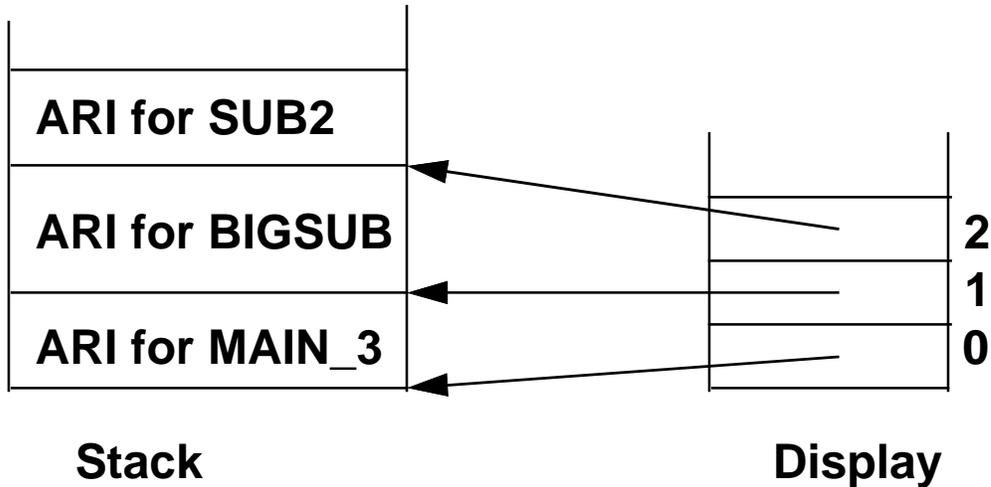**Case 1:** `SUB2` **calls** `SUB1`

Before the call, we have:

ARI for `SUB2`	
ARI for `BIGSUB`	2
	1
ARI for `MAIN_3`	0

Stack   **Display**

After the call, we have:

ARI for `SUB1`	
ARI for `SUB2`	
ARI for `BIGSUB`	2
	1
ARI for `MAIN_3`	0

Stack   **Display**

Chapter 9

Case 2: SUB2 **calls** SUB3

Before the call, we have:

Stack		Display
ARI for SUB2		
ARI for BIGSUB		2
		1
ARI for MAIN_3		0

Stack   **Display**

After the call, we have:

Stack		Display
ARI for SUB3		
ARI for SUB2		3
ARI for BIGSUB		2
		1
ARI for MAIN_3		0

Stack   **Display**
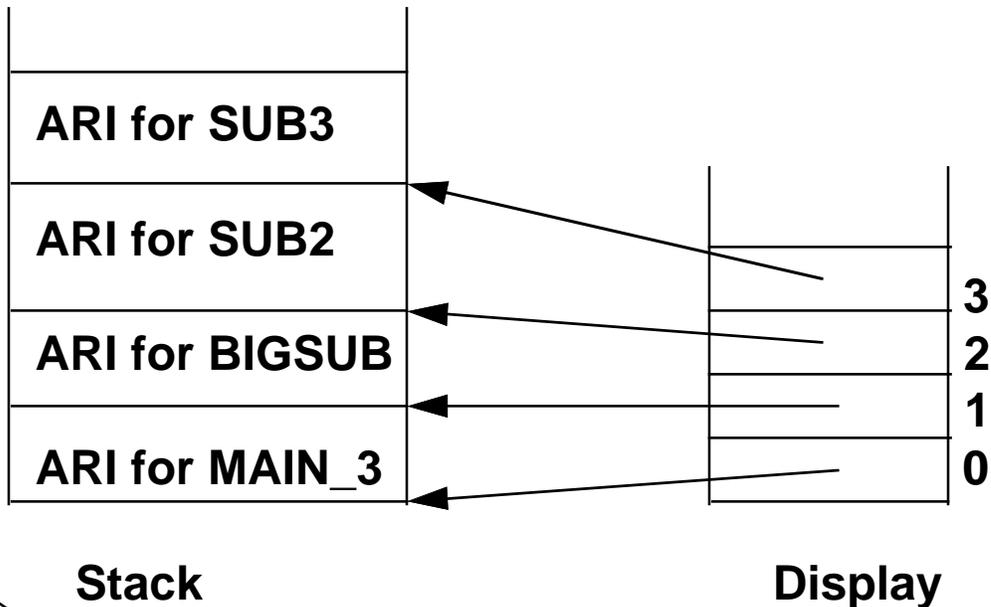
Case 3: SUB3 **calls** SUB1
  Before the call, we have:

ARI for SUB3
ARI for SUB2
ARI for BIGSUB
ARI for MAIN_3

Stack   **Display**

3
2
1
0

After the call, we have:

ARI for SUB1
ARI for SUB3
ARI for SUB2
ARI for BIGSUB
ARI for MAIN_3

Stack   **Display**

3
2
1
0
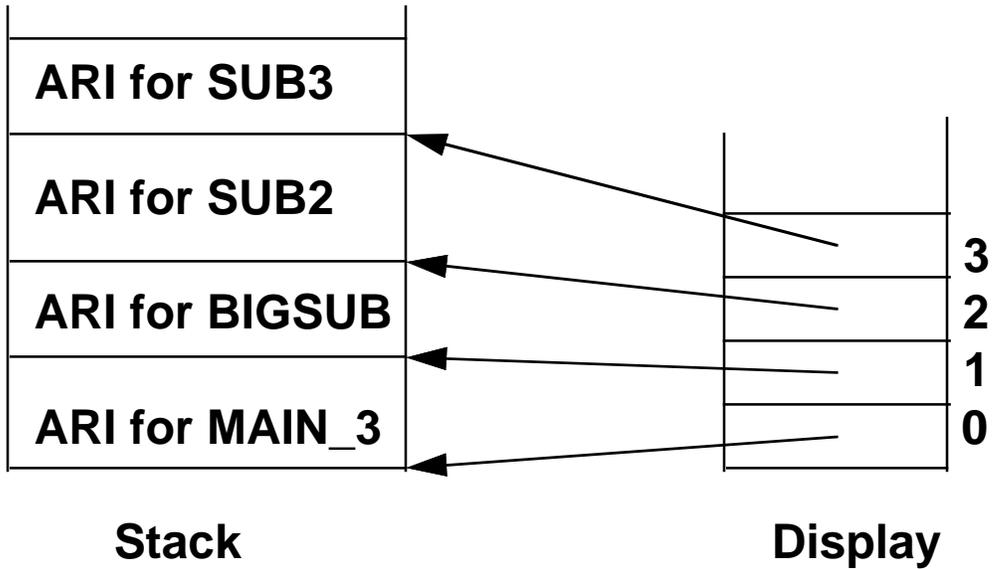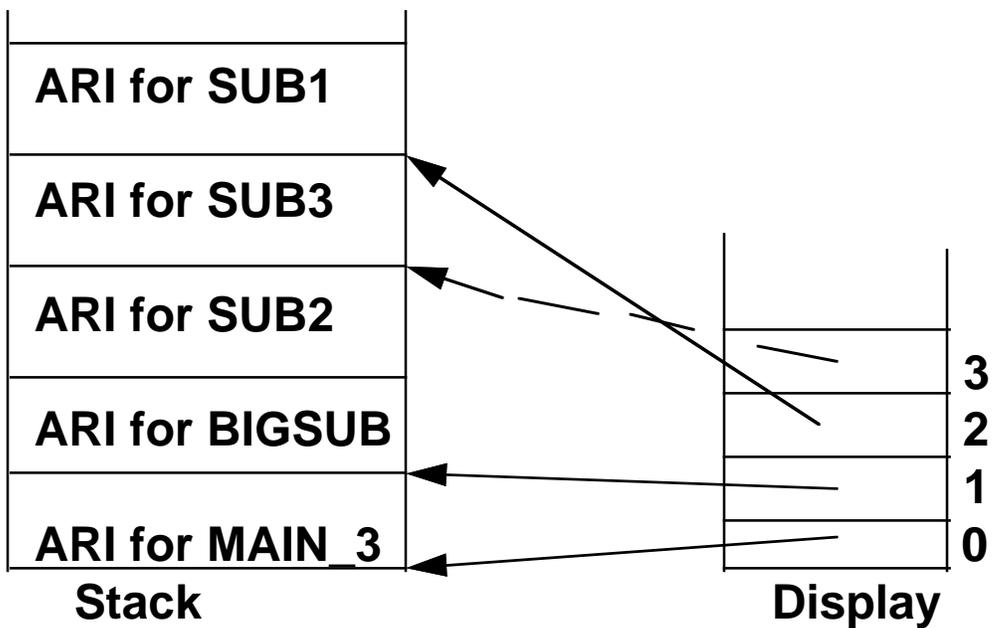
Chapter 9

The display can be kept in registers, if there are enough--it speeds up access and maintenance

Comparing the Static Chain and Display Methods

1. References to locals
  - Not much difference

2. References to nonlocals
   - If it is one level away, they are equal
   - If it is farther away, the display is faster
   - Display is better for time-critical code, because all nonlocal references cost the same

3. Procedure calls
   - For one or two levels of depth, static chain is faster
   - Otherwise, the display is faster

4. Procedure returns
   - Both have fixed time, but the static chain is slightly faster

Overall: Static chain is better, unless the display can be kept in registers

Chapter 9

Implementing Blocks

- Two Methods:
  1. Treat blocks as parameterless subprograms
     - Use activation records

  2. Allocate locals on top of the ari of the subprogram
     - Must use a different method to access locals
     - A little more work for the compiler writer

Implementing Dynamic Scoping

1. *Deep Access* - nonlocal references are found by searching the activation record instances on the dynamic chain
   - Length of chain cannot be statically determined
   - Every activation record instance must have variable names

2. *Shallow Access* - put locals in a central place

   Methods:
   a. One stack for each variable name (see p. 403)
   b. Central table with an entry for each variable name

Chapter 9

Implementing Parameters that are Subprogram Names (deep binding)

1. Static chain
- Compiler simply passes the link to the static parent of the parameter, along with the parameter

2. Display
- All pointers to static ancestors must be saved, because none are necessarily in the environment of the parameter

- In many implementations, the whole display is saved for calls that pass subprogram parameters

Chapter 9

Another look at the referencing environment for a passed subprogram

Consider the following skeletal program:

```
program MAIN_7;
  procedure SUB1;
    begin  { SUB1 }
    ...
    end;   { SUB1 }
  procedure SUB2(procedure SUBX);
    var SUM : real;
    procedure SUB3;
      begin  { SUB3 }
      SUM := 0.0;
      ...
      end;   { SUB3 }
    begin  { SUB2 }
    SUBX;
    SUB2(SUB3);
    ...
    end;   { SUB2 }
  begin  { MAIN_7 }
  ...
  SUB2(SUB1);
  ...
  end.   { MAIN_7 }
```

Which activation of SUB2 has the SUM that SUB3 assigns?