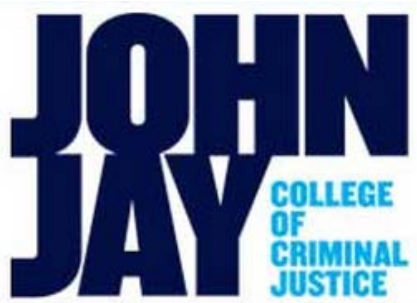


Programming Languages:

Lecture 12

Chapter 10: Implementing Subprograms

Jinwoo Kim
jwkim@jjay.cuny.edu

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Chapter 10 Topics

- The General Semantics of Calls and Returns
- Implementing “Simple” Subprograms
- Implementing Subprograms with Stack-Dynamic Local Variables
- Nested Subprograms
- Blocks
- Implementing Dynamic Scoping

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

The General Semantics of Calls and Returns

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

- A subprogram call has numerous actions associated with it
  - Implement parameter passing
  - Storage arrangement for local variables
  - Save execution status of calling program
  - Transfer of control
  - Necessary action if nested subprograms supported

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Implementing "Simple" Subprograms: Call Semantics

- Basic assumption of "Simple" subprogram
  - No nested subprograms
  - All local variables are static
- Save the execution status of the caller
- Carry out the parameter-passing process
- Pass the return address to the callee
- Transfer control to the callee

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Implementing "Simple" Subprograms: Return Semantics

- If pass-by-value-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

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Implementing "Simple" Subprograms: Storage required for Call & Return

- Status information about the caller
- Parameters
- Return address
- Return value for functions

Implementing "Simple" Subprograms: Parts

- Two separate parts of the simple subprogram
  - Actual code part
    - **Constant**
  - Noncode part
    - **local variables and data that can change**
  - They are fixed in size

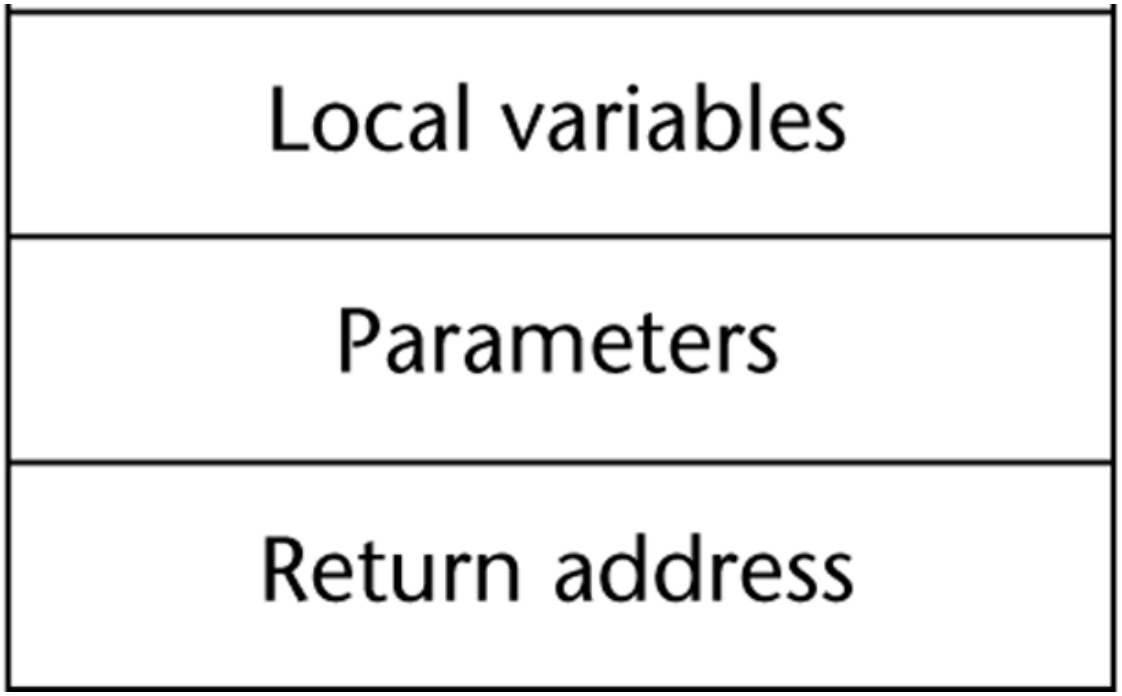
- The format, or layout, of the noncode part of an executing subprogram is called an *activation record*
  - The data it describes are relevant only during activation

- An *activation record instance* is a concrete example of an activation record
  - The collection of data for a particular subprogram activation

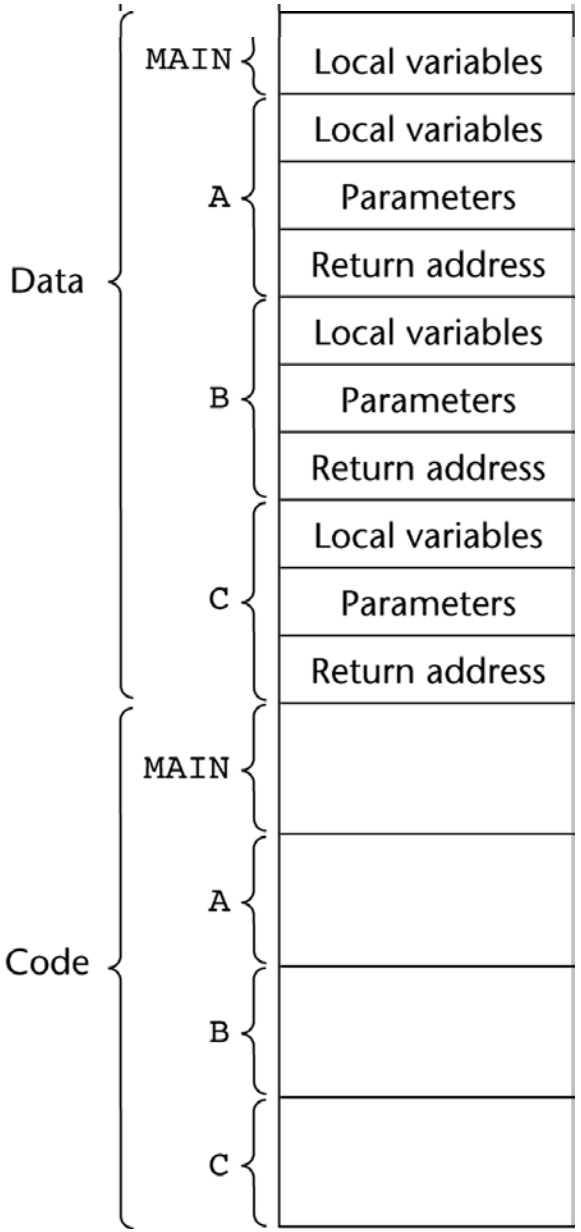
An Activation Record for "Simple" Subprograms

Since simple subprograms do not supports recursion, there can be only one active version of a given subprogram at a time

Single instance of activation record for a subprogram exists



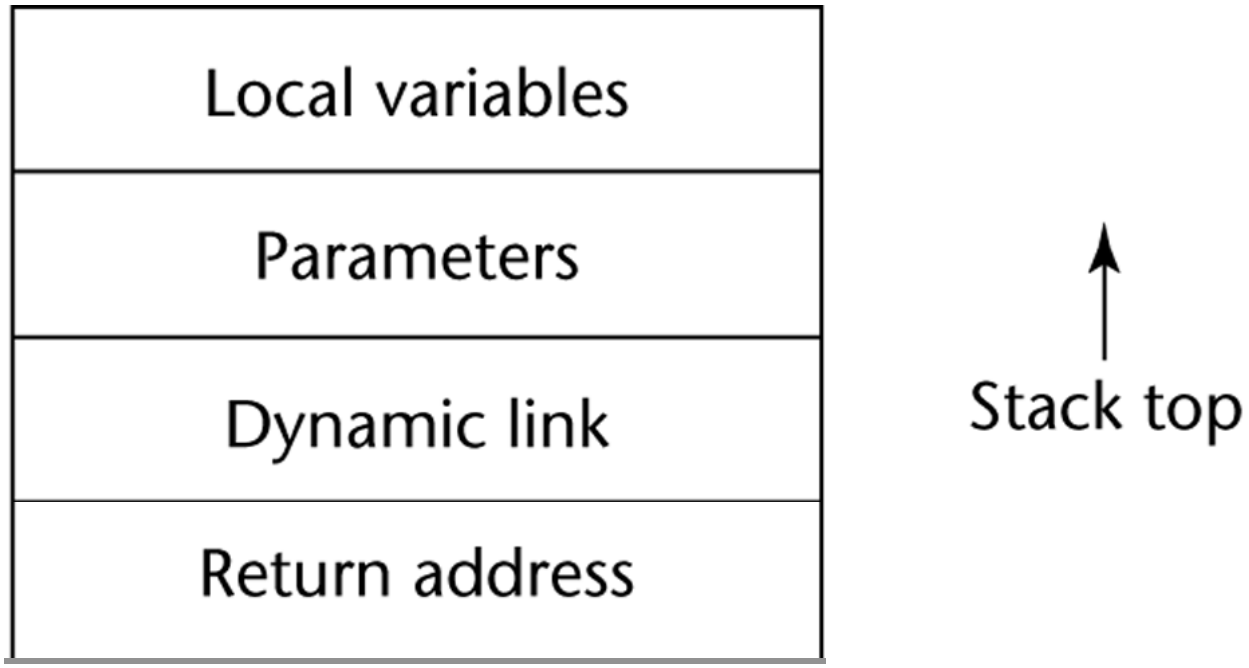
Code and Activation Records of a Program with "Simple" Subprograms



Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
  - The compiler must generate code to cause implicit allocation and de-allocation 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

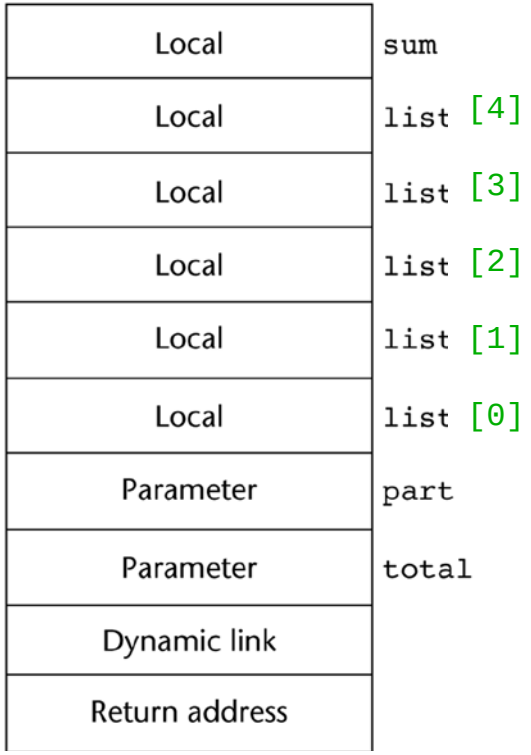


Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- 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
- Run-time stack

An Example: C Function

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

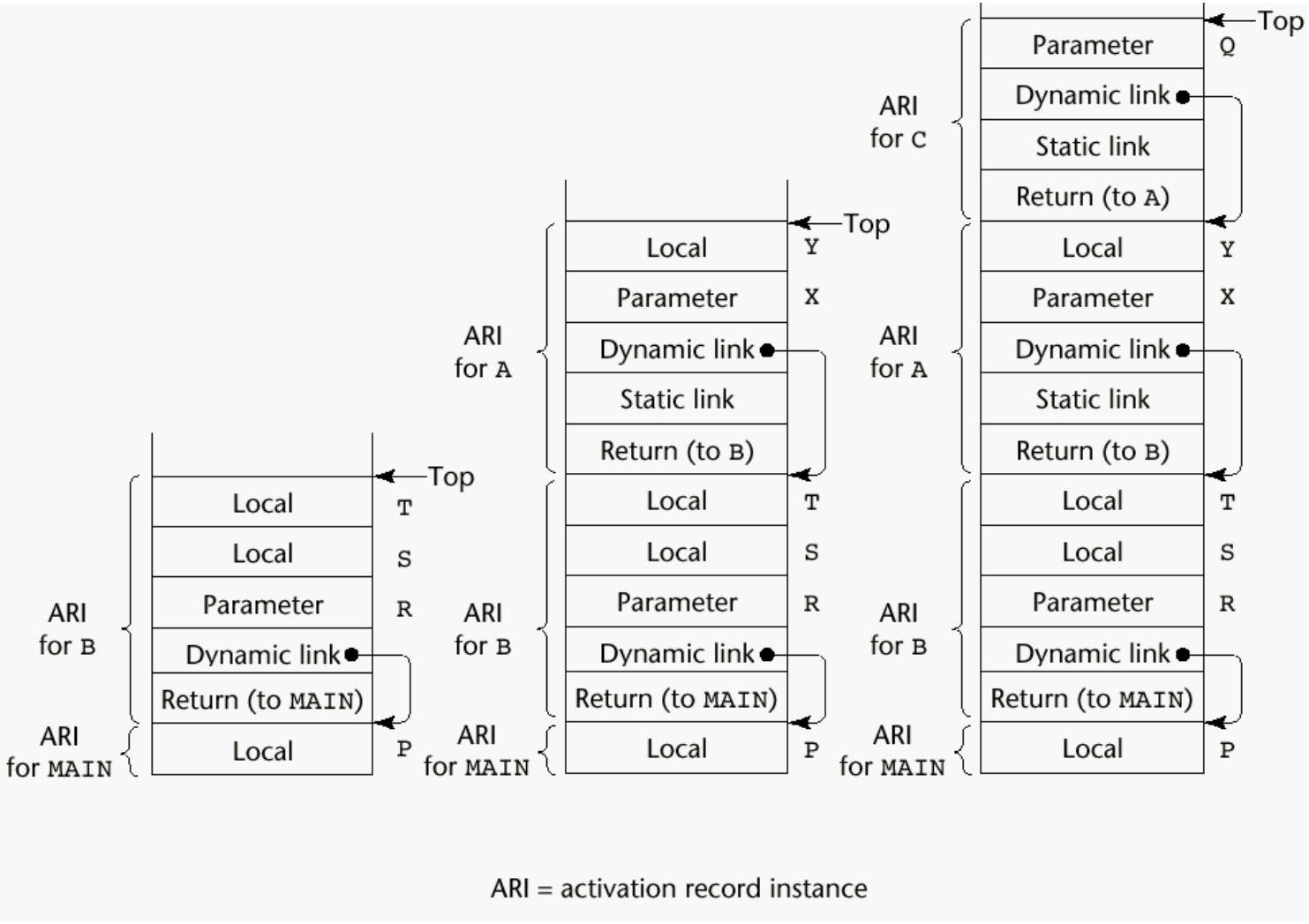


An Example Without Recursion

```
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);
    ...
}
```

main calls B
B calls A
A calls C

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
  - This offset is called the *local_offset*
- The local_offset of a local variable can be determined by the compiler at compile time

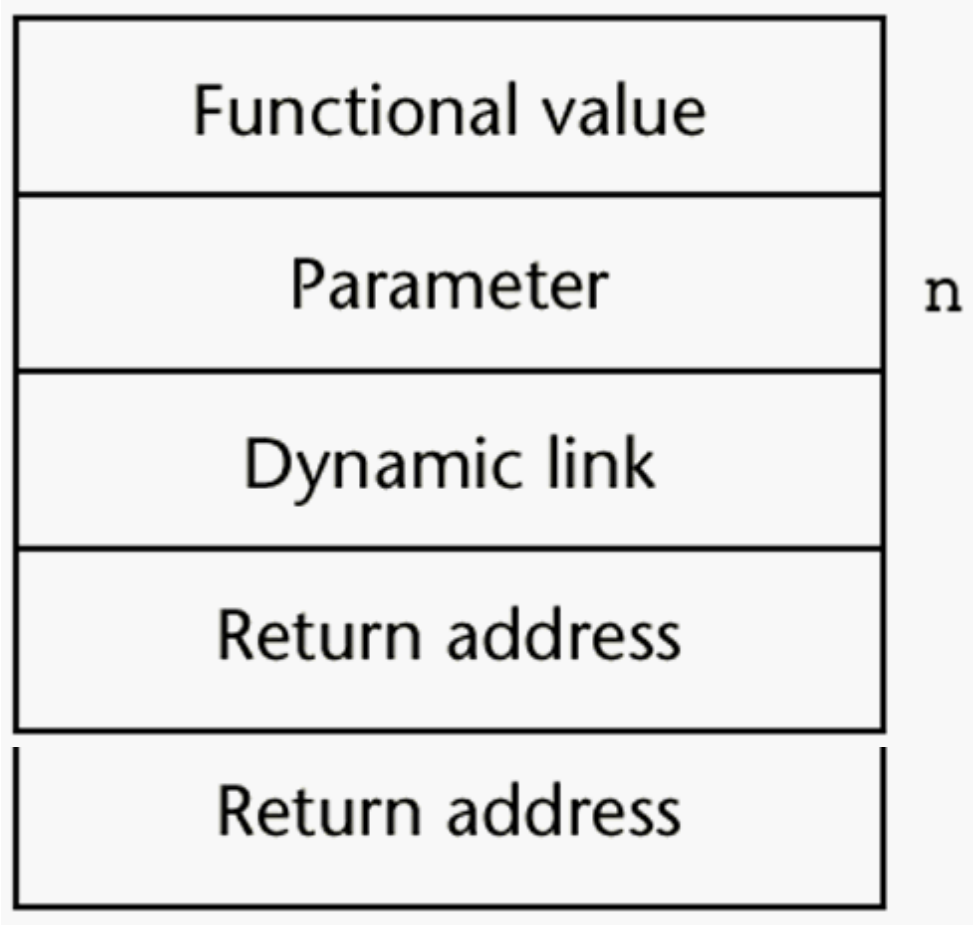
An Example With Recursion

- The activation record used in the previous example supports recursion, 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
}
```

Activation Record for `factorial`



JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, JavaScript) use stack-dynamic local variables and allow subprograms to be nested

- All variables that can be non-locally accessed reside in some activation record instance in the stack

- The process of locating a non-local reference:
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Locating a Non-local Reference

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

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Static Scoping

- 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

Example Pascal 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 }
```

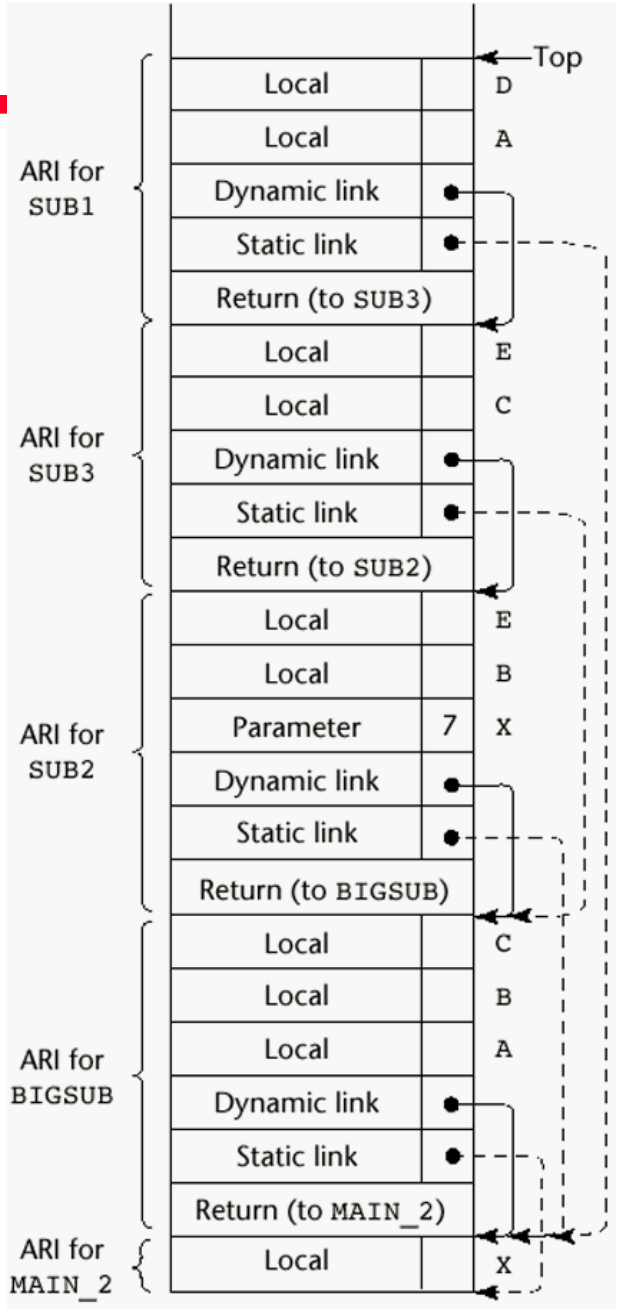
Example Pascal Program (continued)

- Call sequence for `MAIN_2`

  `MAIN_2` calls `BIGSUB`
  `BIGSUB` calls `SUB2`
  `SUB2` calls `SUB3`
  `SUB3` calls `SUB1`

Stack Contents at Position 1

```
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 }
```



Displays

- An alternative to static chains
- Static links are stored in a single array called a display
- The contents of the display at any given time is a list of addresses of the accessible activation record instances

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Blocks

- Blocks are user-specified local scopes for variables
- An example in C

```
{int temp;
 temp = list [upper];
 list [upper] = list [lower];
 list [lower] = temp
}
```

- The lifetime of `temp` in the above example begins when control enters the block
- An advantage of using a local variable like `temp` is that it cannot interfere with any other variable with the same name

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Implementing Blocks

- Two Methods:
  1. Treat blocks as parameter-less subprograms that are always called from the same location
     - **Every block has an activation record; an instance is created every time the block is executed**
  2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

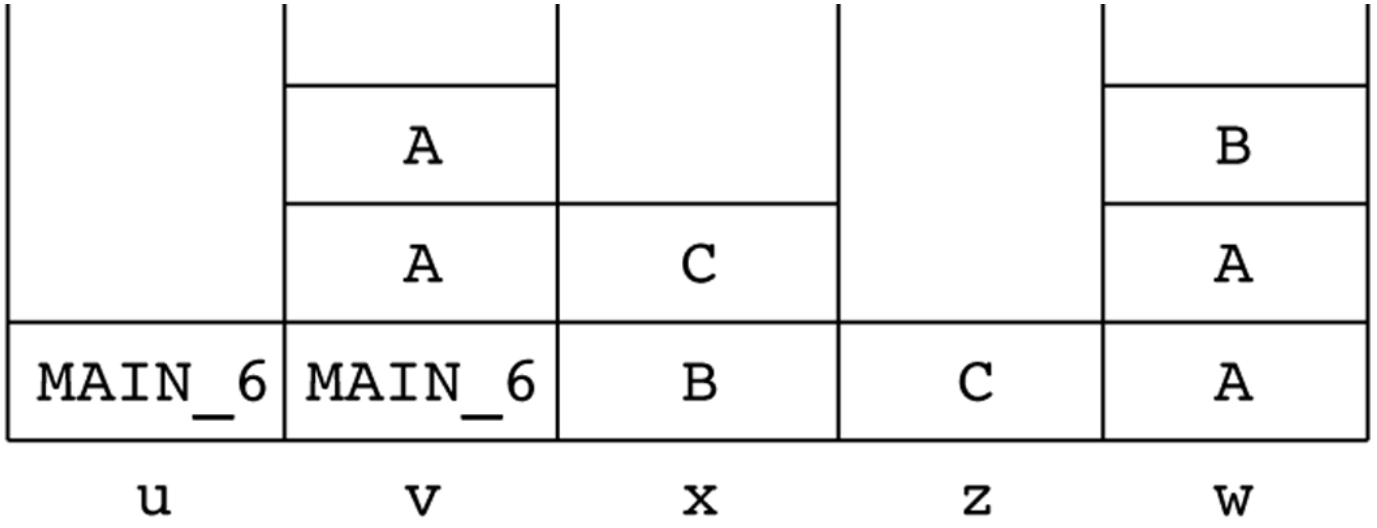
JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Implementing Dynamic Scoping

- *Deep Access*: non-local references are found by searching the activation record instances on the dynamic chain

- *Shallow Access*: put locals in a central place
  - One stack for each variable name
  - Central table with an entry for each variable name

Using Shallow Access to Implement Dynamic Scoping



(The names in the stack cells indicate the program units of the variable declaration.)

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Summary

- Subprogram linkage semantics requires many action by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex
- Subprograms with stack-dynamic local variables and nested subprograms have two components
  - actual code
  - activation record

JOHN JAY COLLEGE OF CRIMINAL JUSTICE

Summary (continued)

- Activation record instances contain formal parameters and local variables among other things
- Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or thru some central variable table method

Homework #7

- Problem Solving (P. 477 of class textbook)
  - 1, 3, 7, 8, 9

- Due date: One week from assigned date
  - Please hand in printed (typed) form
    - **I do not accept any handwritten assignment**
    - **Exception: pictures**