Principles of Programming Languages

subprograms implementation

Semantics of Calls and Returns

- General semantics of subprogram calls
  - Pass parameters
  - Allocate storage of local variables and bind them
  - Save the execution status of calling subprogram
  - Transfer of control and arrange for the return
- General semantics of subprogram returns:
  - Return values of out- and inout-mode parameters to the corresponding actual parameters
  - Deallocate storage of local variables
  - Restore the execution status
  - Return control to the caller

Outline

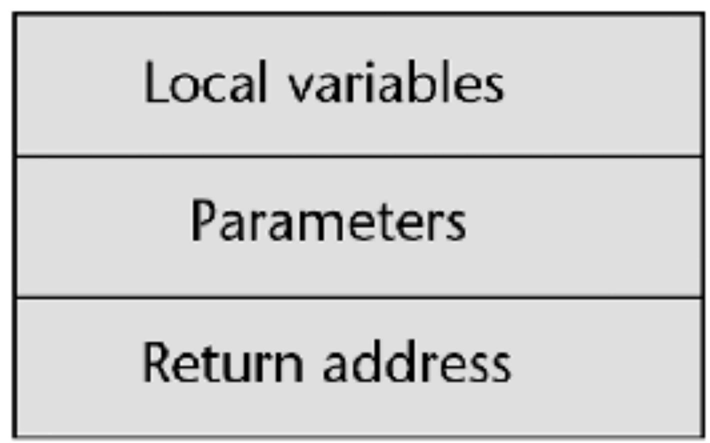
- Semantics of Calls and Returns (Sec. 10.1)
- Implementing "Simple" Subprograms (Sec. 10.2)
- Implementing Subprograms with Stack-Dynamic Local Variables (Sec. 10.3)
- Nested Subprograms (Sec. 10.4)
- Blocks (Sec. 10.5)
- Implementing Dynamic Scoping (Sec. 10.6)

Implementing “Simple” Subprograms

- “Simple” subprograms:
  - Subprograms cannot be nested
  - All local variables are static
- Required storage for calls and returns:
  - Status information of caller, parameters, return address, return value for functions
- A “simple subprogarm” consists of two parts:
  - Subprogram code
  - Non-code part (local variables and above data for calls and returns)

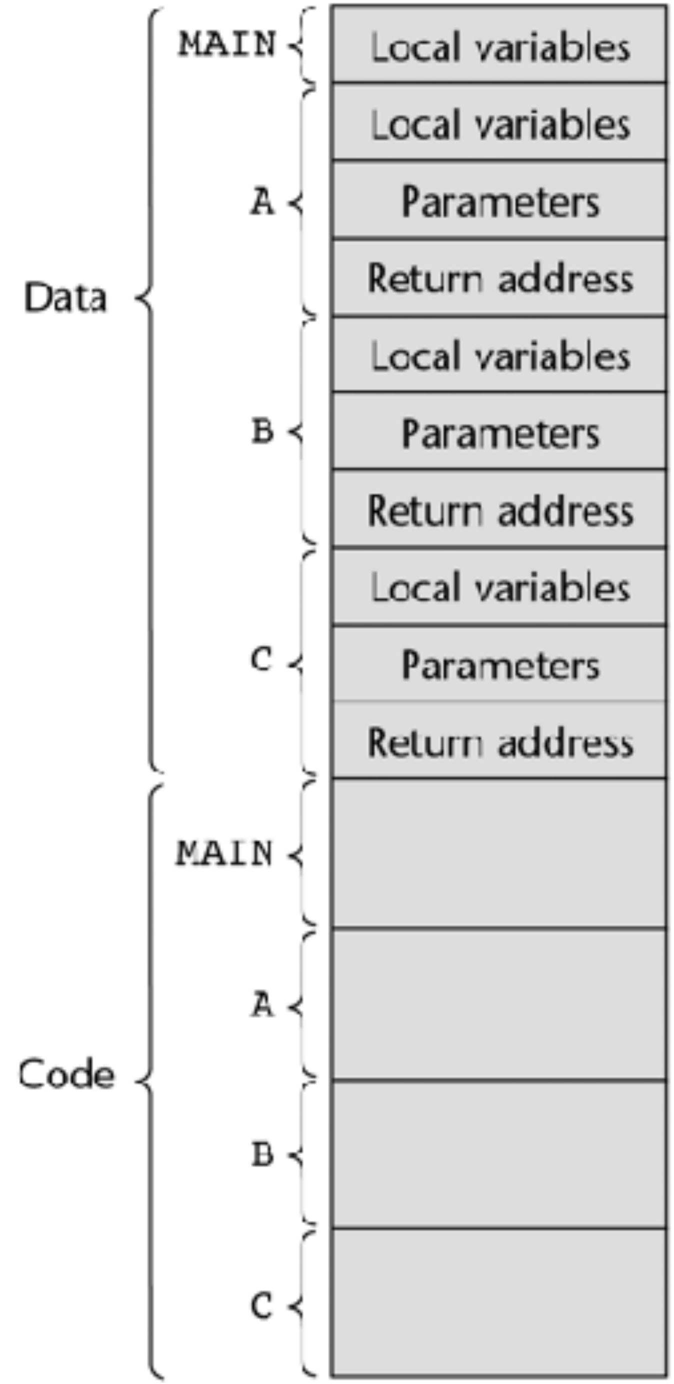
Implementing "Simple" Subprograms

- Format, or layout, of non-code part of an executing subprogram is called activation record (AR)
  - For a "simple" subprogram, AR has fixed size, and can be statically allocated (not in stack)
  - Can it support recursion?



Example Layout

- Code and activation records of a program with three "simple" subprograms: A, B, C
- These parts may be separately compiled and put together by linker



Outline

- Semantics of Calls and Returns (Sec. 10.1)
- Implementing “Simple” Subprograms (Sec. 10.2)
- Implementing Subprograms with Stack-Dynamic Local Variables (Sec. 10.3)
- Nested Subprograms (Sec. 10.4)
- Blocks (Sec. 10.5)
- Implementing Dynamic Scoping (Sec. 10.6)

Stack-Dynamic Local Variables

- Allocate local variables on the run-time stack
  - Main advantage: support recursion
  - Why?
- More complex storage management:
  - Compiler must generate code for implicit allocation and deallocation of local variables on the stack
  - Recursion adds possibility of multiple simultaneous activations of a subprogram
    → multiple instances of activation records

Contents of Activation Record

- What data are needed for the function to run?

```
int AddTwo(int x, y)
{
    int sum;
    sum = x + y;
    return sum;
}
```

Parameters: x, y
Local variable: sum
Return address
Saved registers

"state"

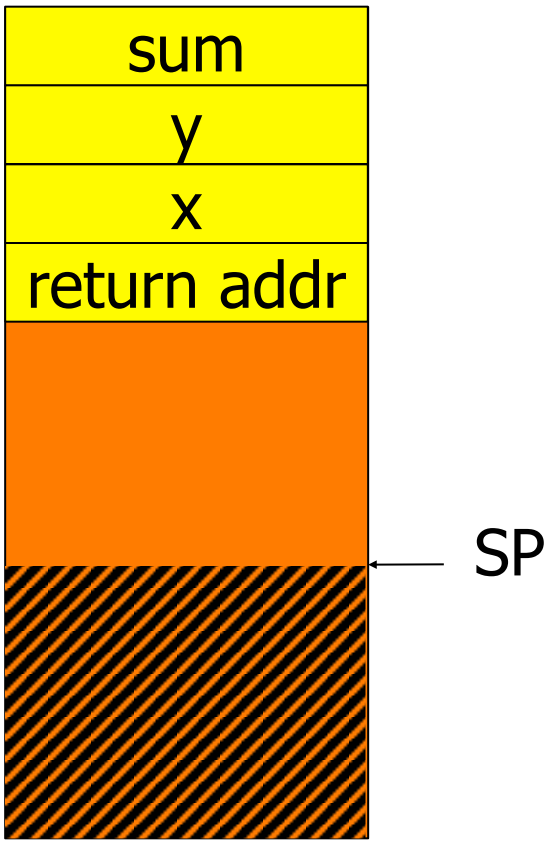
sum
y
x
return addr

Size can be determined at compile time

Accessing Activation Record

- When AddTwo is called, its AR is dynamically created and pushed onto the run-time stack
- How to reference the variables in stack, i.e., x, y, sum?

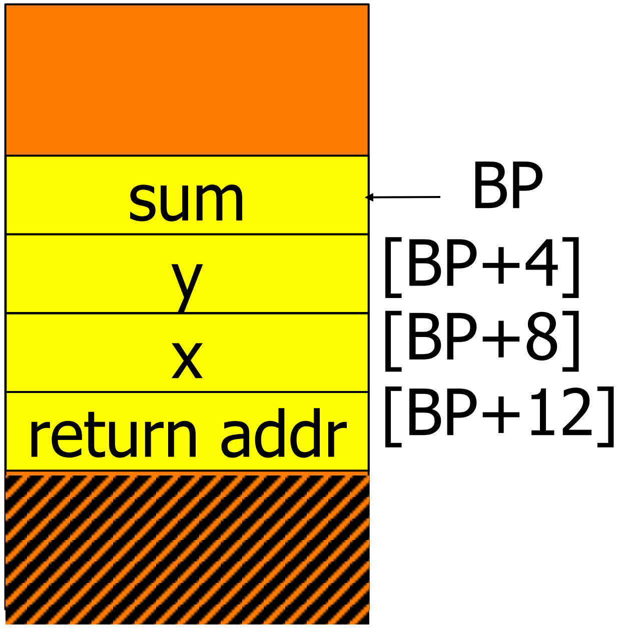
```
AddTwo  PROC
    mov   eax,x
    add   eax,y
    mov   sum,eax
    ret
AddTwo  ENDP
```



- How about SP of caller?

Accessing Activation Record

- Idea: use addresses relative to a base address of AR, which does not change during subprog. → base pointer, frame pointer, or dynamic link
- Dedicate a register to hold this pointer → BP
- A subprog. can explicitly access stack parameters using constant offsets from BP, e.g. [BP + 8]
- BP is restored to its original value when subprog. returns



Activation Record for Stack-Dyna

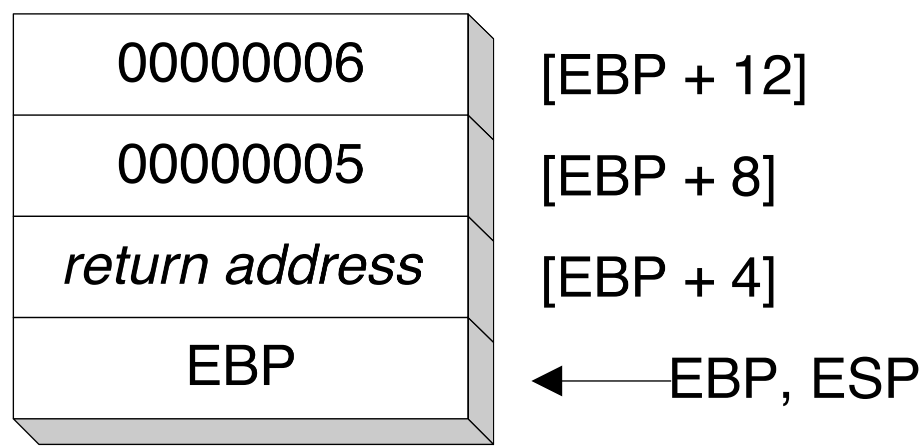
- Base pointer (BP):
  - Always points at the base of the activation record instance of the currently executing program unit
  - When a subprogram is called, the current BP is saved in the new AR instance and the BP is set to point at the base of the new AR instance
  - Upon return from the subprogram, BP is restored from the AR instance of the callee

Activation Record Example (x86)

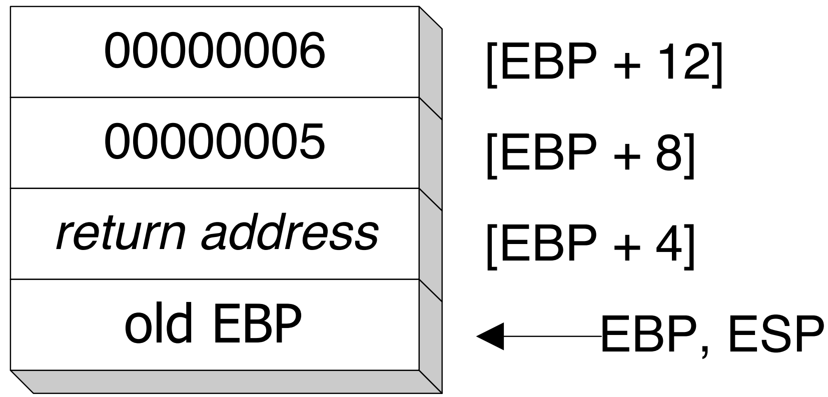
```
.data
sum DWORD ?
.code
   push 6           ; second argument
   push 5           ; first argument
   call AddTwo      ; EAX = sum
   mov  sum,eax     ; save the sum
```

Return value in eax

```
AddTwo PROC
   push ebp
   mov  ebp,esp
   .
   .
```



Activation Record Example (x86)

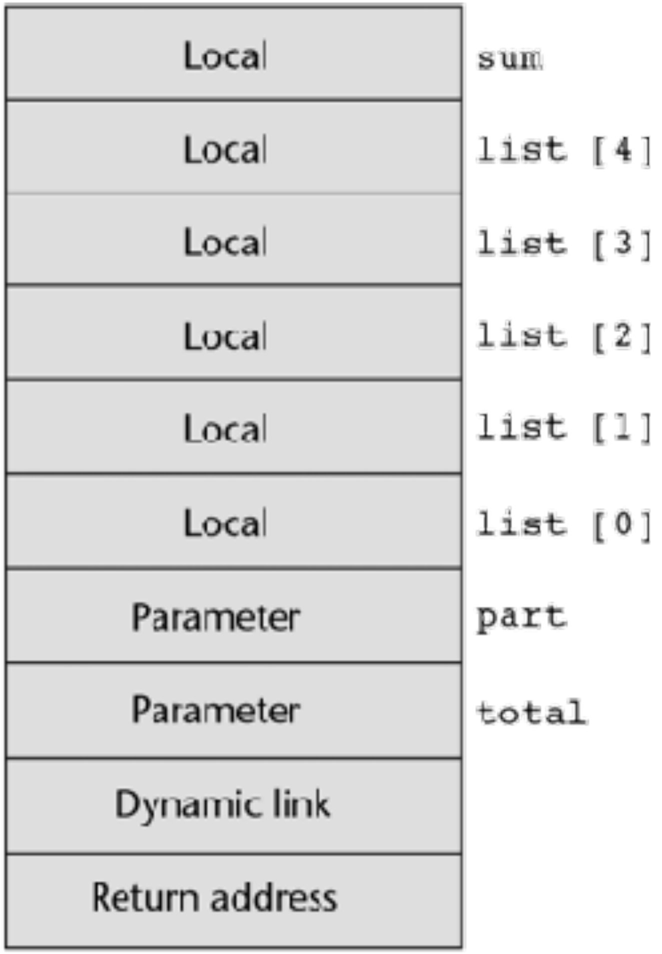



```
AddTwo PROC
   push ebp
   mov ebp,esp              ; base of stack frame
   mov eax,[ebp + 12]; second argument (6)
   add eax,[ebp + 8] ; first argument (5)
   pop ebp
   ret 8                    ; clean up the stack
AddTwo ENDP                 ; EAX contains the sum
```


14

Activation Record: Local Array

```
void sub(float total, int part)
{
  int list[5];
  float sum;
  …
}
```



An Example without Recursion

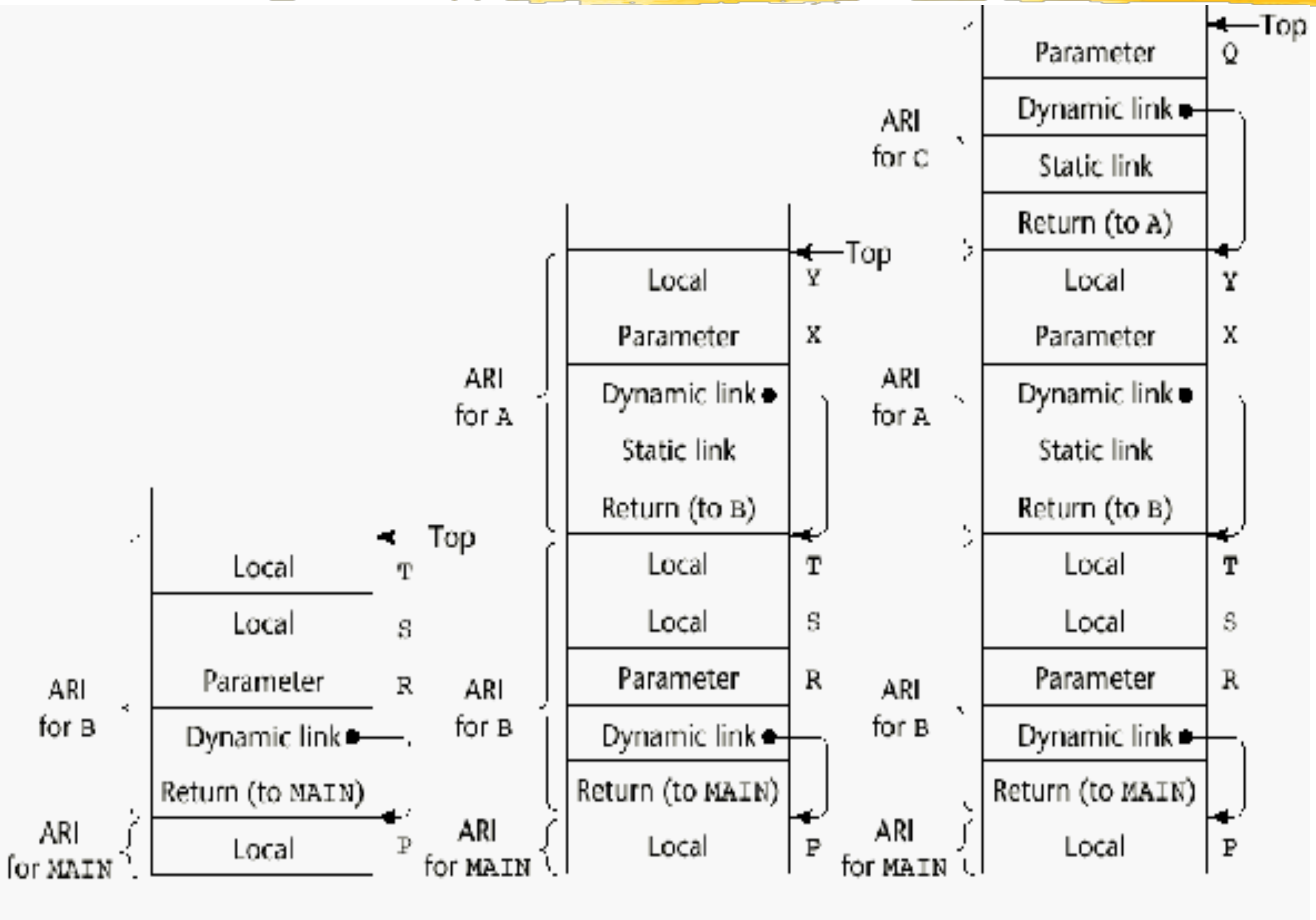
```
void A(int x) {
  int y;
  ...
  C(y);
  ...
}
void B(float r) {
  int s, t;
  ...
  A(s);
  ...
}
```

```
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

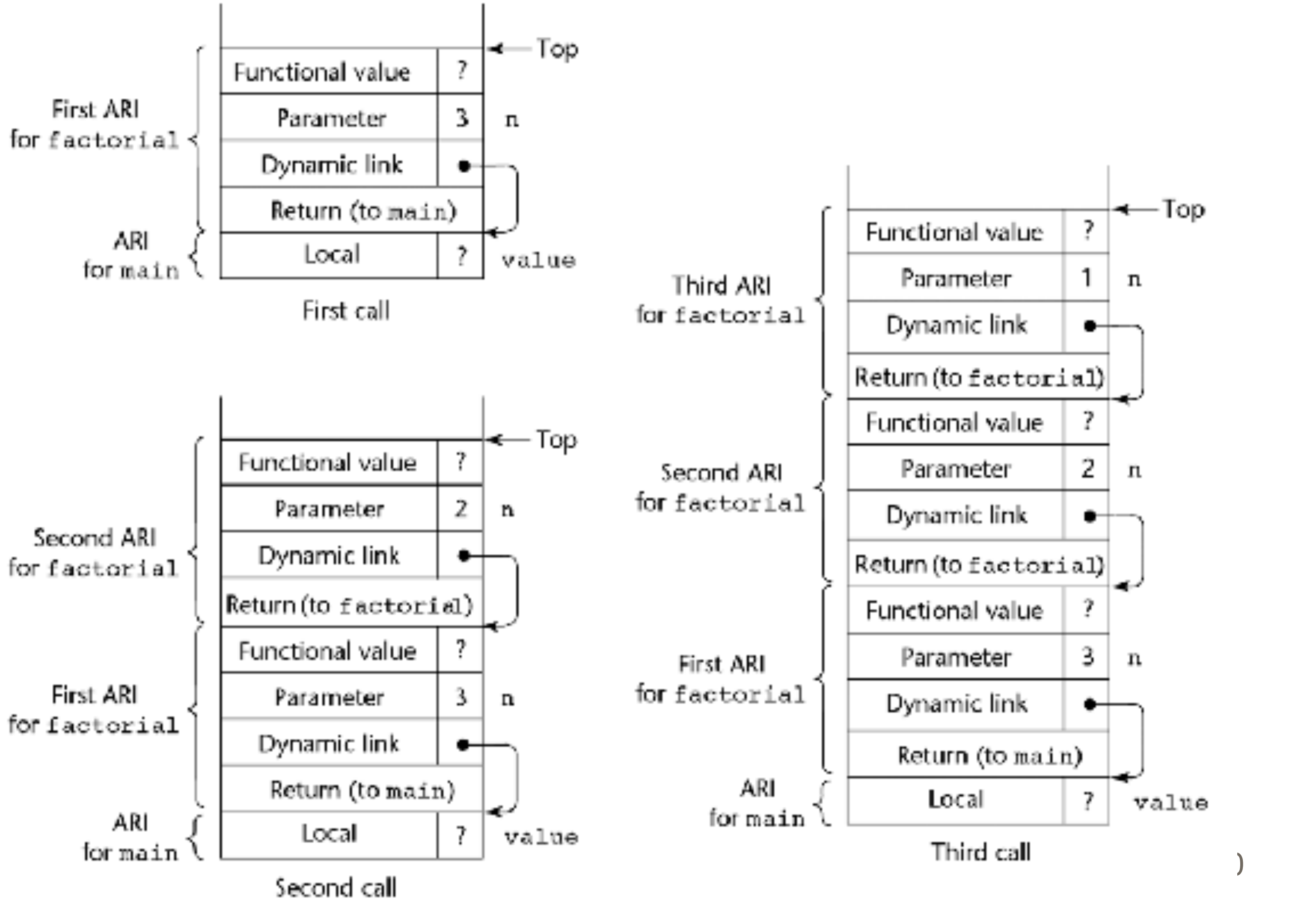
- The collection of dynamic links in the stack at a given time is called the dynamic chain, or call chain
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the BP. This offset is called the local_offset
- The local offset of a local variable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

An Example with Recursion

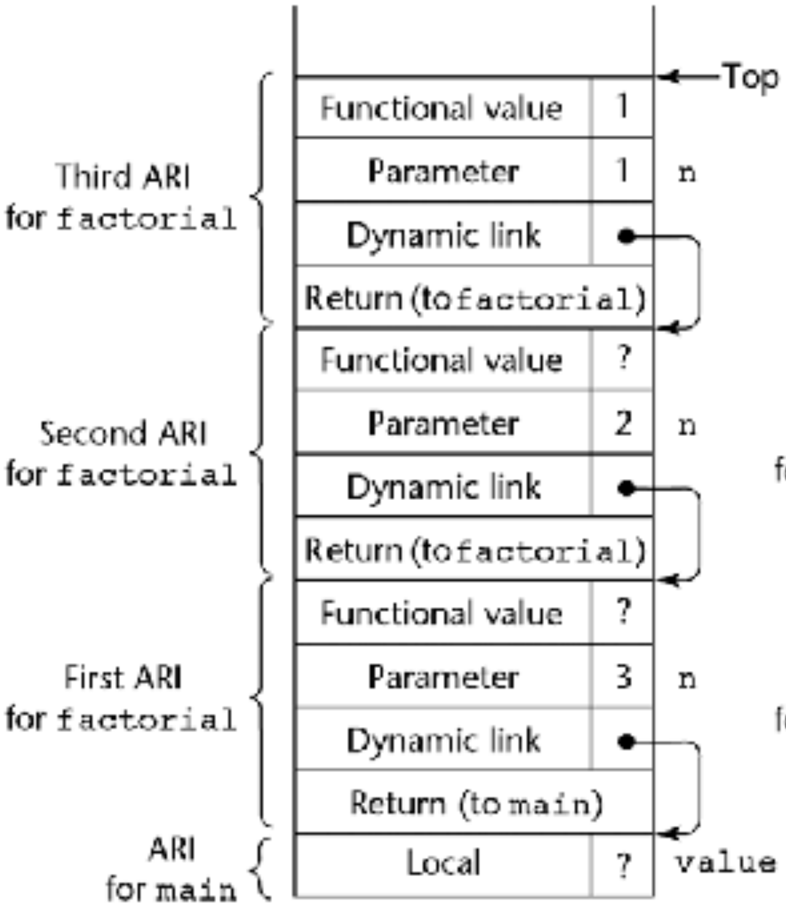
```
int factorial (int n) {
    <-----------------------------1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
    <-----------------------------2
}
void main() {
  int value;
  value = factorial(3);
    <-------------------3
}
```

Functional value	
Parameter	n
Dynamic link	
Return address	

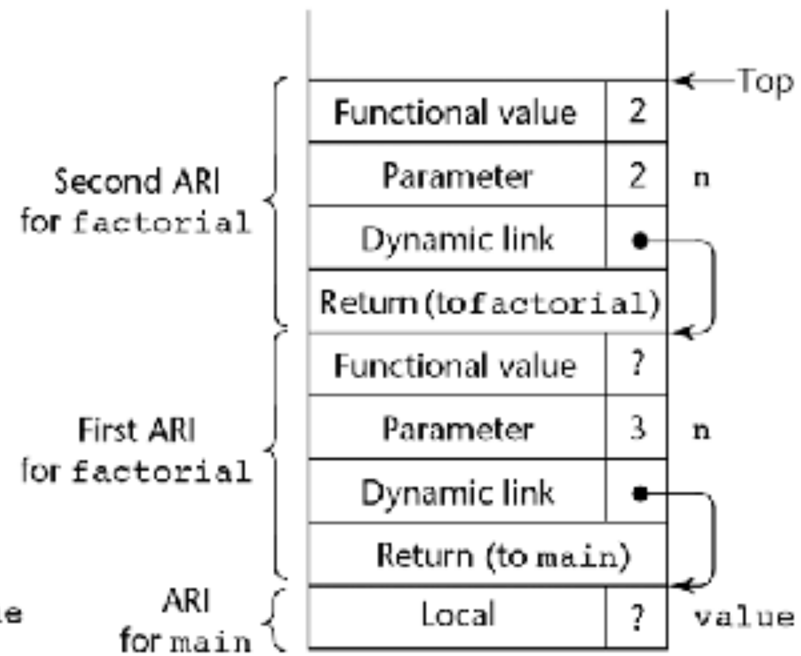
Stack at Position 1 in 3 Executions



Stack at Position 2



At position 2
in factorial
third call completed



At position 2
in factorial
second call completed

Outline

- Semantics of Calls and Returns (Sec. 10.1)
- Implementing “Simple” Subprograms (Sec. 10.2)
- Implementing Subprograms with Stack-Dynamic Local Variables (Sec. 10.3)
- Nested Subprograms (Sec. 10.4)
- Blocks (Sec. 10.5)
- Implementing Dynamic Scoping (Sec. 10.6)

Nested Subprograms

- Some languages (e.g., Fortran 95, Ada, Python, JavaScript, Ruby) use stack-dynamic local variables and allow subprograms to be nested

```
procedure A is
  procedure B is
    procedure C is
    end;  -- of C
  end; -- of B
end; -- of A
```

Nested Subprograms

- How to access variables that are non-local but are defined in outer subprograms?
  - These variables must reside in some AR instances deep in the stack
- The process of locating a non-local reference:
  - Find the correct activation record instance down in the stack: hard
  - Determine the correct offset within that activation record instance: easy

Finding Correct AR Instance

- Static scope semantics:
  - Only variables that are declared in static ancestor scope are visible and can be accessed
  - All non-local variables that can be referenced have been allocated in some AR instance on the stack when the reference is made
- Idea: chain AR instances of static ancestors

Static Chain

- Static link in an AR instance points to bottom of AR instance of the static parent
- Static chain connects all static ancestors of an executing subprogram, static parent first
- Can find correct AR instance following the chain
  - But, can be even easier, because nesting of scopes is known at compile time and thus the length of static chain to follow

Following Static Chain

- Static_depth: depth of nesting of a static scope
- Chain_offset or nesting_depth of a nonlocal reference is the difference between static_depth of the reference and that of the declare scope
- A reference to a variable can be represented by:

  (chain_offset, local_offset),

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

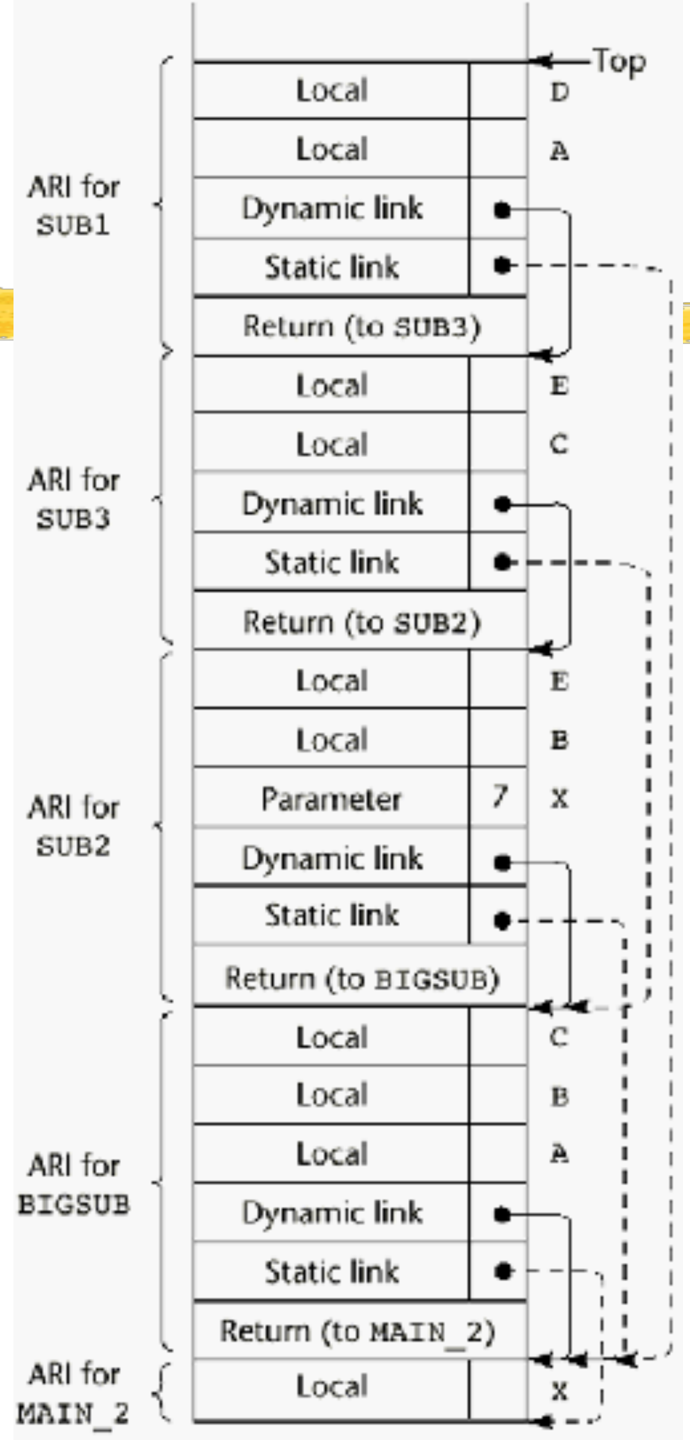
Example Ada Program

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
      A := B + C;  <--------------1
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
        Sub1;
        E := B + A:   <---------2
        end; -- of Sub3
      begin -- of Sub2
      Sub3;
      A := D + E;  <----------3
      end; -- of Sub2 }
    begin -- of Bigsub
    Sub2(7);
    end; -- of Bigsub
  begin
  Bigsub;
end; of Main_2 }
```

Stack Contents at Position 1

Main_2 calls Bigsub
Bigsub calls Sub2
Sub2 calls Sub3
Sub3 calls Sub1

Reference to variable A:
Position 1: (0,3)
Position 2: (2,3)
Position 3: (1,3)



Static Chain Maintenance

- At the call, AR instance must be built
  - The dynamic link is just the old stack top pointer
  - The static link must point to the most recent AR instance of the static parent
  - Two methods:
    1. Search the dynamic chain to find the parent scope
    2. When compiler encounter a subprogram call, it finds its static parent and records the nesting_depth from that parent to itself. When that subprogram is called, its static link can be found starting from the caller's static link and the number of nesting_depth

Evaluation of Static Chains

Problems:

- A nonlocal reference is slow if the nesting depth is large
- Time-critical code is difficult:
  - Costs of nonlocal references are difficult to determine
  - Code changes can change the nesting depth, and therefore the cost

Displays

- An alternative to static chains that solves the problems with that approach
- 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

Outline

- Semantics of Calls and Returns (Sec. 10.1)
- Implementing "Simple" Subprograms (Sec. 10.2)
- Implementing Subprograms with Stack-Dynamic Local Variables (Sec. 10.3)
- Nested Subprograms (Sec. 10.4)
- Blocks (Sec. 10.5)
- Implementing Dynamic Scoping (Sec. 10.6)

Blocks

- User-specified local scopes for variables

```
{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

Two Methods Implementing Blocks

- 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
- Put locals of a block in the same AR of the containing subprogram
  - 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

Summary

- Subprogram linkage semantics requires many action by the implementation
- Stack-dynamic languages are more complex and often have two components
  - Actual code
  - Activation record: AR instances contain formal parameters and local variables among other things
- Static chains are main method of implementing accesses to non-local variables in static-scoped languages with nested subprograms