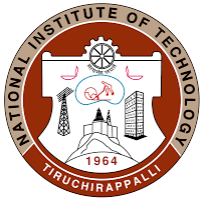


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
~~11.~~	~~Abstract Data Types and Encapsulation Constructs~~
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 10 – Implementing Subprograms

Objectives

- Explore the implementation of subprograms
- How subprogram linkage works
- Nonnestable subprograms with static local variables
- Subprograms with stack-dynamic local variables
- Nested subprograms with stack-dynamic local variables and static scoping
- Static chain method of accessing nonlocals in static-scoped languages
- Techniques for implementing blocks
- Several methods of implementing nonlocal variable access in a dynamic-scoped language

Static vs Dynamic Scoping

```
1  x <- 1
2  f <- function(a) x + a
3  g <- function() {
4    x <- 2
5    f(0)
6  }
7  g() # what does this return?
```

- Under *lexical scoping* (also known as *static scoping*), the scope of a variable is determined by the lexical (*i.e.*, textual) structure of a program
- Under *dynamic scoping*, a variable is bound to the most recent value assigned to that variable

- In Static Scoping, the final result will be 1
- In Dynamic Scoping, the final result will be 2

Static Scoping vs Dynamic Scoping

- Lexical scoping (sometimes known as *static scoping*) is a convention used with many programming languages that sets the *scope* (range of functionality) of a variable so that it may only be *called* (referenced) from within the block of code in which it is defined. The scope is determined when the code is compiled. A variable declared in this fashion is sometimes called a *private* variable
- The opposite approach is known as *dynamic scoping*. Dynamic scoping creates variables that can be called from outside the block of code in which they are defined. A variable declared in this fashion is sometimes called a *public* variable

Introduction

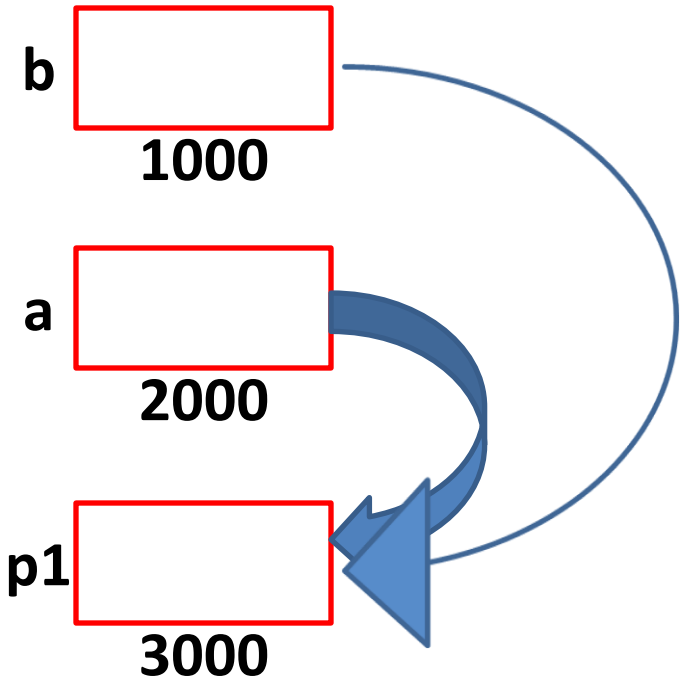
- Subprogram call and return operations are together called subprogram linkage
- Call process must include the implementation of whatever parameter-passing method is used
- If local variables are not static, the call process must allocate storage for the locals declared in the called subprogram and bind those variables to that storage
- Must save the execution status of the calling program unit
- Execution status is everything needed to resume execution of the calling program unit -> Register Values, CPU Status Bits and the Environment Pointer (EP)
- EP is used to access parameters and local variables during the execution of a subprogram
- Calling process also must arrange to transfer control to the code of the subprogram and ensure that control can return to the proper place when the subprogram execution is completed

Introduction

- If the language supports nested subprograms, the call process must create some mechanism to provide access to nonlocal variables that are visible to the called subprogram
- Required actions of a subprogram return are less complicated than those of a call
- If the subprogram has parameters that are out mode or inout mode and are implemented by copy, the first action of the return process is to move the local values of the associated formal parameters to the actual parameters
- Next, it must deallocate the storage used for local variables and restore the execution status of the calling program unit
- Finally, control must be returned to the calling program unit

Pass-by-Result (Miscellaneous)

```
void sub(out int a, out int b)
{
    a = 17;
    b = 35;
}
int p1;
f.sub(out p1, out p1);
```



Implementing “Simple” Subprograms

- By “simple” we mean that subprograms cannot be nested and all local variables are static
- Semantics of a call to a “simple” subprogram requires the following actions
  - Save the execution status of the current program unit
  - Compute and pass the parameters
  - Pass the return address to the called
  - Transfer control to the called
- Semantics of a return from a simple subprogram requires the following actions
  - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters
  - If the subprogram is a function, the functional value is moved to a place accessible to the caller
  - Execution status of the caller is restored
  - Control is transferred back to the caller

Implementing “Simple” Subprograms

- The call and return actions require storage for the following
  - Status information about the caller
  - Parameters
  - Return address
  - Return value for functions
  - Temporaries used by the code of the subprograms
- These, along with the local variables and the subprogram code, form the complete collection of information a subprogram needs to execute and then return control to the caller
- Question now is the distribution of the call and return actions to the caller and the called

Implementing “Simple” Subprograms

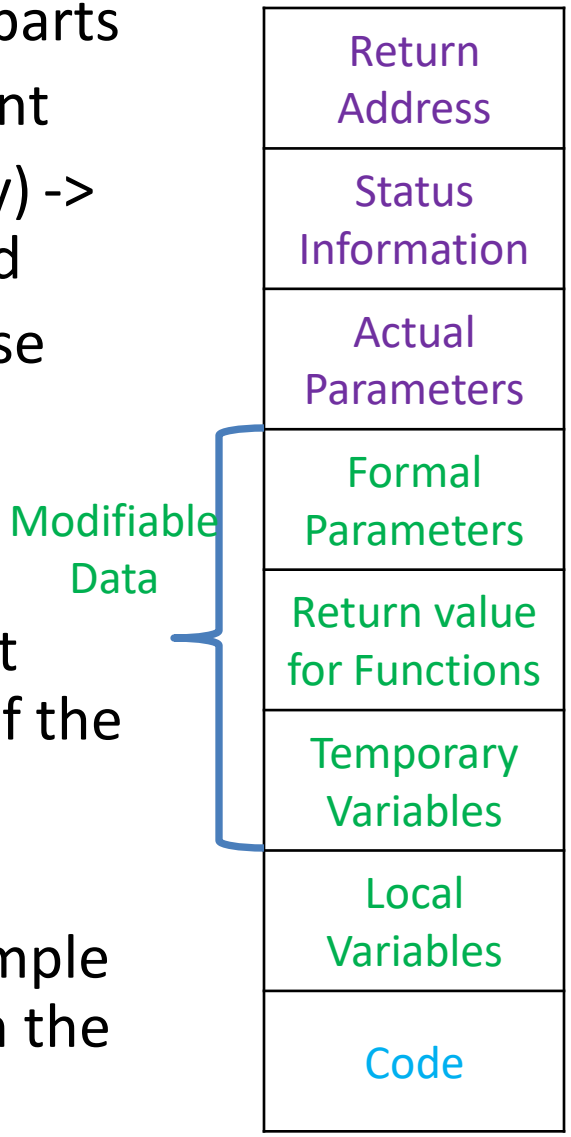
- Last three actions of a call clearly must be done by the caller
- Saving the execution status of the caller could be done by either
- In the case of the return, the first, second, ~~third~~ and fourth actions must be done by the called
- Restoration of the execution status of the caller could be done by either the caller or the called
- In general, the linkage actions of the called can occur at two different times
    - At the beginning of execution -> Prologue of the subprogram linkage
    - At the end of execution -> Epilogue of the subprogram linkage
- In the case of a simple subprogram, all of the linkage actions of the callee occur at the end of its execution, so there is no need for a prologue

Miscellaneous

- Semantics of a call to a “simple” subprogram requires the following actions
  - Save the execution status of the current program unit
  - Compute and pass the parameters
  - Pass the return address to the called
  - Transfer control to the called
- Semantics of a return from a simple subprogram requires the following actions
  - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters
  - If the subprogram is a function, the functional value is moved to a place accessible to the caller
  - Execution status of the caller is restored
  - Control is transferred back to the caller

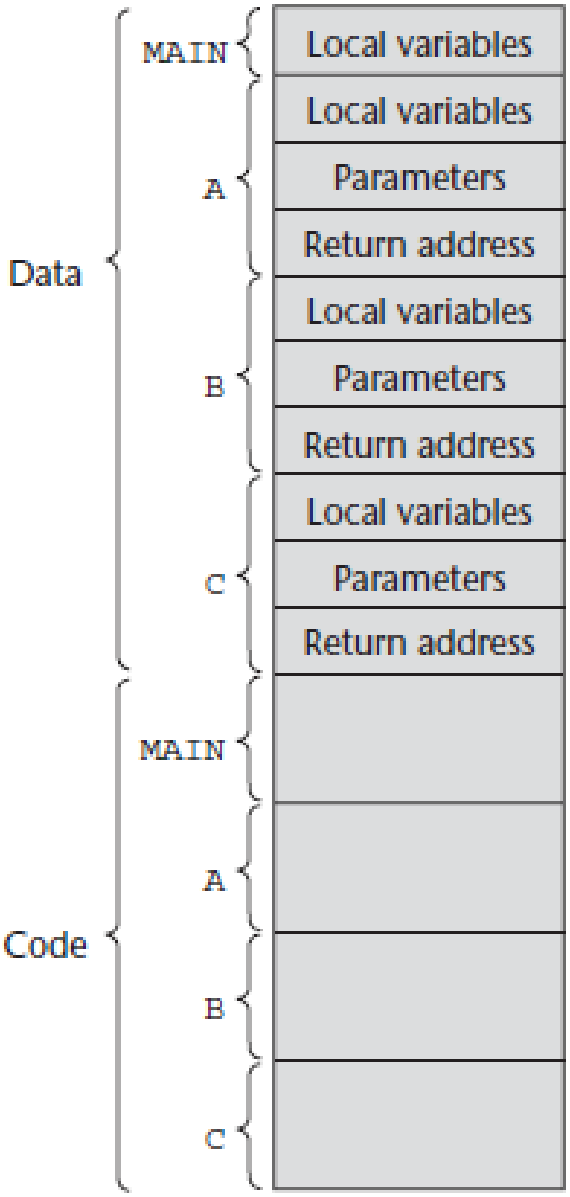
Implementing “Simple” Subprograms

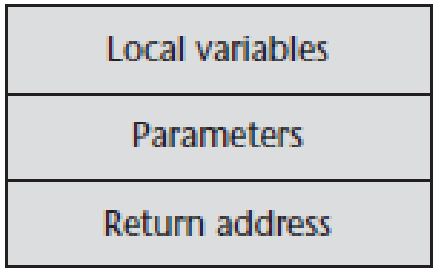
- A simple subprogram consists of two separate parts
  - Actual code of the subprogram -> Constant
  - Local variables and data (listed previously) -> Change when the subprogram is executed
- In the case of simple subprograms, both of these parts have fixed sizes
- Format or layout of the noncode part of a subprogram is called an activation record
  - Because the data it describes are relevant only during the activation, or execution of the subprogram
- Form of an activation record is static
- An activation record instance is a concrete example of an activation record -> A collection of data in the form of an activation record



Implementing “simple” Subprograms

- Because languages with simple subprograms do not support recursion, there can be only one active version of a given subprogram at a time
- Therefore, there can be only a single instance of the activation record for a subprogram
- Because an activation record instance for a “simple” subprogram has fixed size, it can be statically allocated
- In fact, it could be attached to the code part of the subprogram





Implementing “Simple” Subprograms

- Construction of the complete program is not done entirely by the compiler
- In fact, if the language allows independent compilation, the four program units have been compiled on different days, or even in different years
- At the time each unit is compiled, the machine code for it, along with a list of references to external subprograms, is written to a file
- Executable program is put together by the linker, which is part of the operating system (Sometimes linkers are called *loaders, linker/loaders or link editors)*
- When the linker is called for a main program, its first task is to find the files that contain the translated subprograms referenced in that program and load them into memory
- Then, the linker must set the target addresses of all calls to those subprograms in the main program to the entry addresses of those subprograms
- The same must be done for all calls to subprograms in the loaded subprograms and all calls to library subprograms
- In the previous example, the linker was called for MAIN
- The linker had to find the machine code programs for A, B, and C, along with their activation record instances, and load them into memory with the code for MAIN
- Then, it had to patch in the target addresses for all calls to A, B, C, and any library subprograms in A, B, C and MAIN

Implementing Subprograms with Stack-Dynamic Local Variables

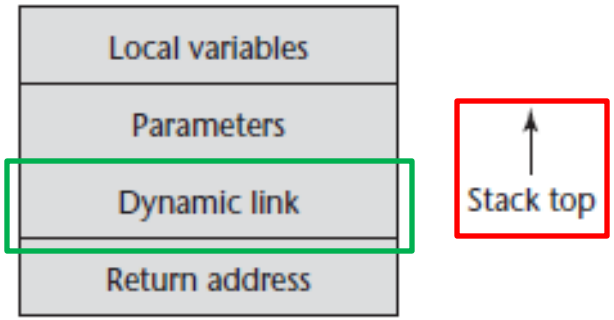
- Examine the implementation of the subprogram linkage in languages in which locals are stack dynamic
- One of the most important advantages of stack-dynamic local variables is support for recursion

More Complex Activation Record

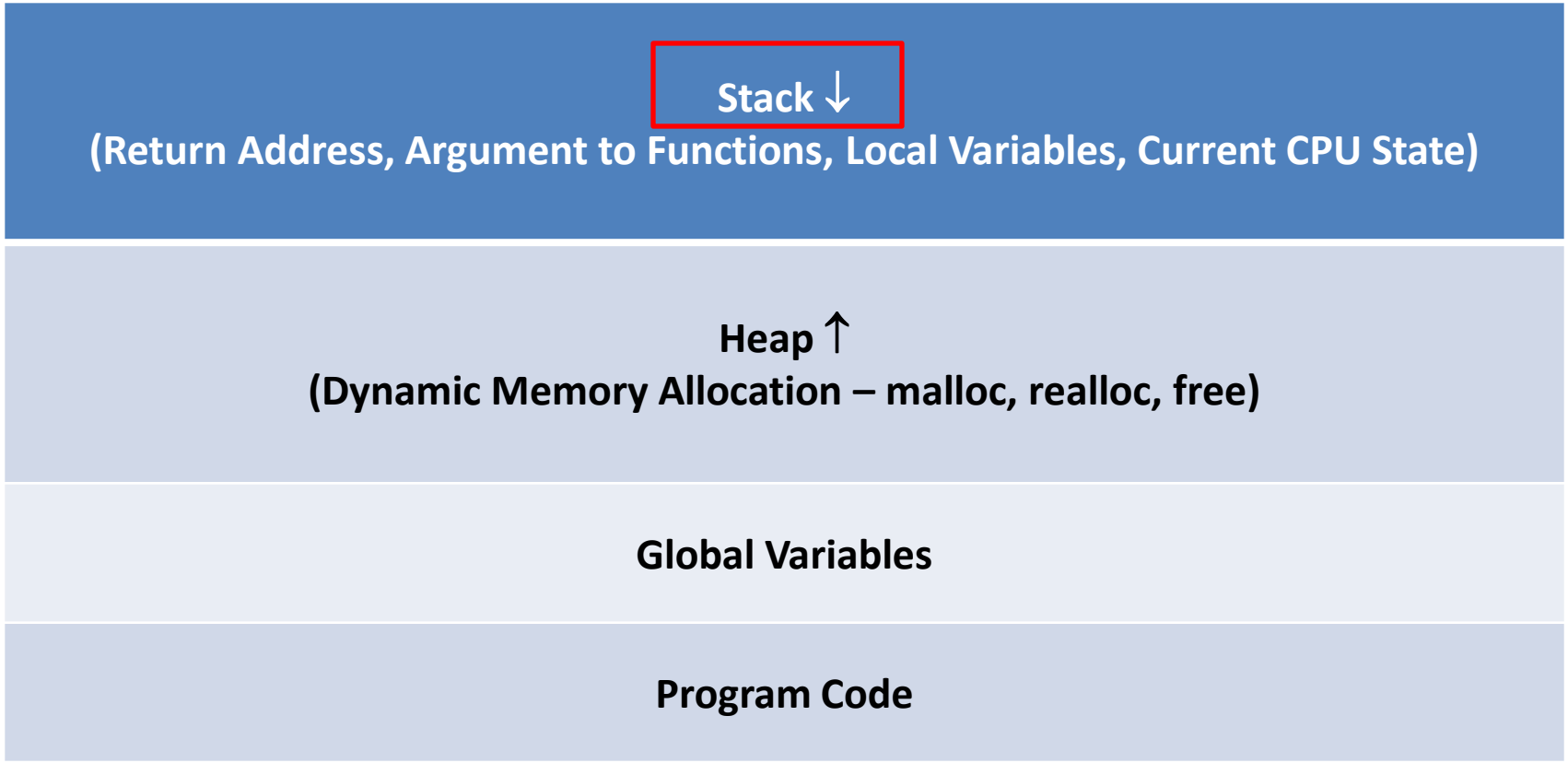
- Compiler must generate code to cause the implicit allocation and deallocation of local variables
- Recursion adds the possibility of multiple simultaneous activations of a subprogram, which means that there can be more than one instance (incomplete execution) of a subprogram at a given time, with at least one call from outside the subprogram and one or more recursive calls
    - Number of activations is limited only by the memory size of the machine
- Each activation requires its activation record instance

Implementing Subprograms with Stack-Dynamic Local Variables

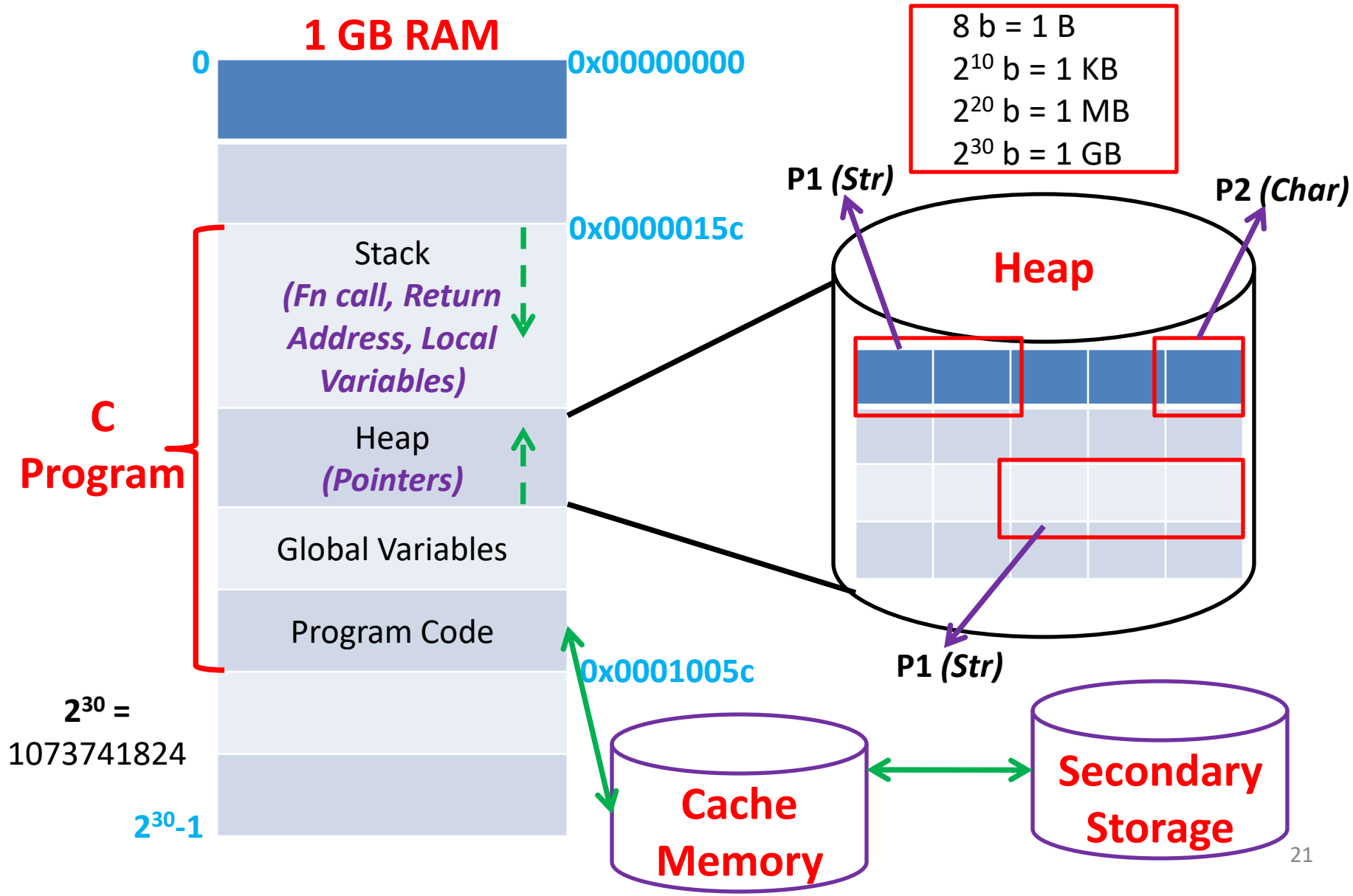
- Format of an activation record for a given subprogram in most languages is known at compile time
- In many cases, the size is also known for activation records because all local data are of a fixed size
- That is not the case in some other languages, such as Ada, in which the size of a local array can depend on the value of an actual parameter
- In those cases, the format is static, but the size can be dynamic
- In languages with stack-dynamic local variables, activation record instances must be created dynamically



Miscellaneous



Miscellaneous



Implementing Subprograms with Stack-Dynamic Local Variables

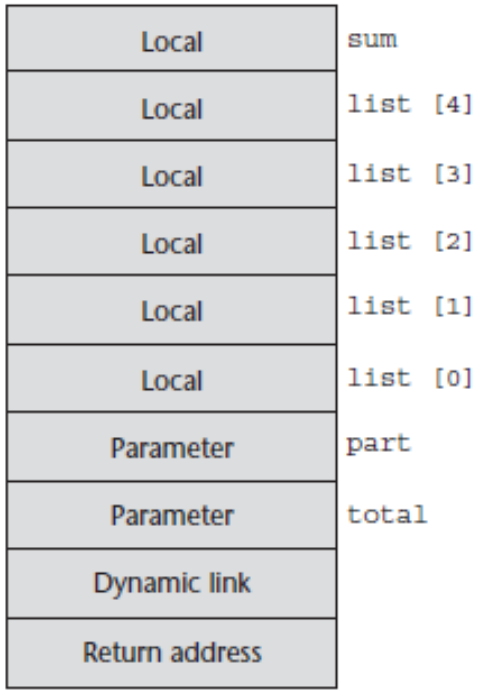
- Dynamic link is a pointer to the base of the activation record instance of the caller
  - In static-scoped languages, this link is used to provide traceback information when a run-time error occurs
  - In dynamic-scoped languages, the dynamic link is used to access nonlocal variables
- Actual parameters in the activation record are the values or addresses provided by the caller
- Local scalar variables are bound to storage within an activation record instance
- Local variables that are structures are sometimes allocated elsewhere, and only their descriptors and a pointer to that storage are part of the activation record
- Local variables are allocated and possibly initialized in the called subprogram, so they appear last

Scalar variable is a variable that holds only one value at a time

Implementing Subprograms with Stack-Dynamic Local Variables

- Format of the activation record is fixed at compile time, although its size may depend on the call in some languages
- Because the call and return semantics specify that the subprogram last called is the first to complete, it is reasonable to create instances of these activation records on a stack
- This stack is part of the runtime system and therefore is called the run-time stack, although we will usually just refer to it as the stack
- Every subprogram activation, whether recursive or nonrecursive, creates a new instance of an activation record on the stack
- This provides the required separate copies of the parameters, local variables and return address

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



Implementing Subprograms with Stack-Dynamic Local Variables

- Initially, the EP points at the base, or first address of the activation record instance of the main program
- Therefore, the run-time system must ensure that it always points at the base of the activation record instance of the currently executing program unit
- When a subprogram is called, the current EP is saved in the new activation record instance as the dynamic link
- The EP is then set to point at the base of the new activation record instance
- Upon return from the subprogram, the stack top is set to the value of the current EP ~~minus one~~ and the EP is set to the dynamic link from the activation record instance of the subprogram that has completed its execution
- Resetting the stack top effectively removes the top activation record instance

Miscellaneous

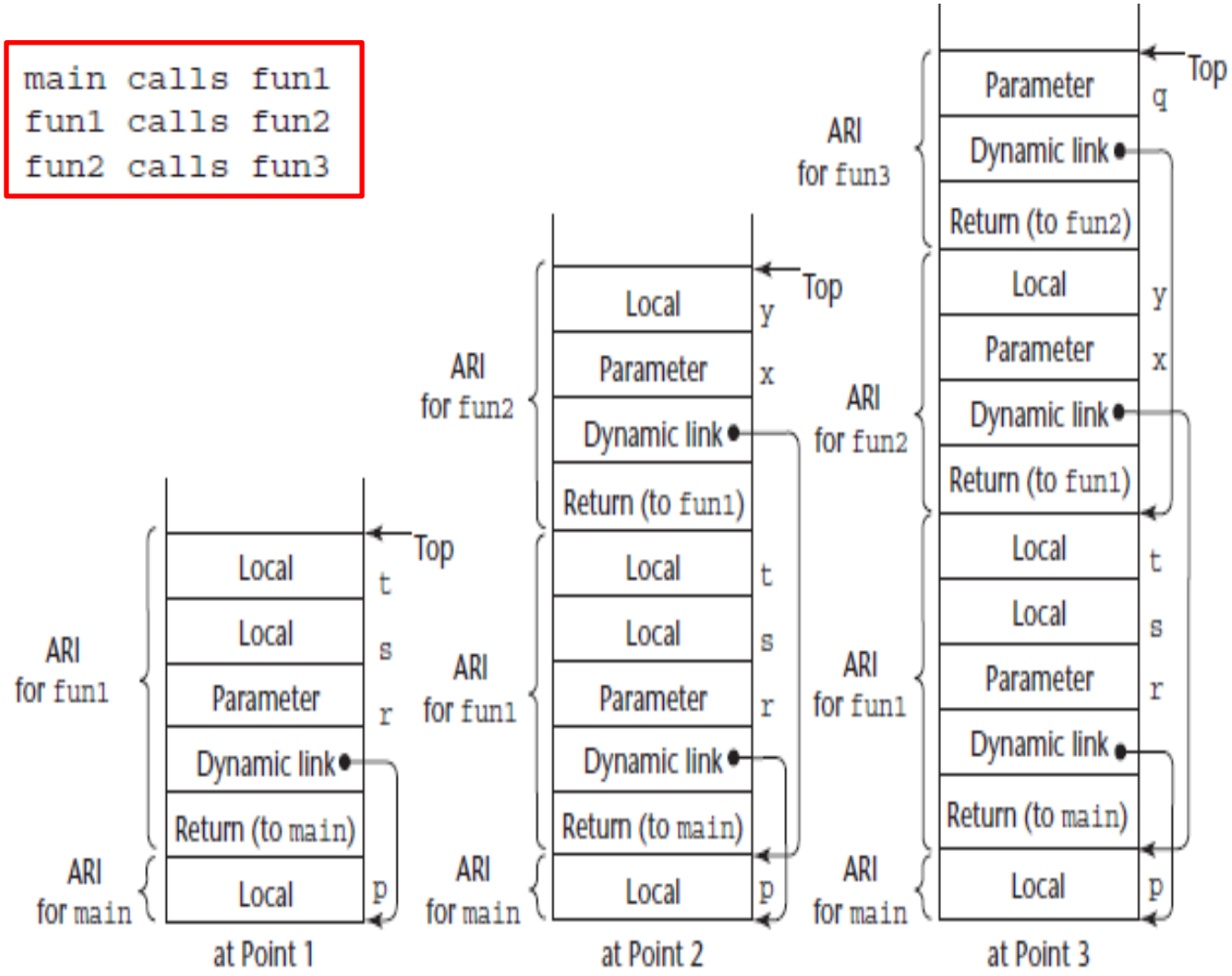
```
void fun1(float r) {
  int s, t;
  ...        <-- 1
  fun2(s);
  ...
}

void fun2(int x) {
  int y;
  ...        <-- 2
  fun3(y);
  ...
}

void fun3(int q) {
  ...        <-- 3
}

void main() {
  float p;
  ...
  fun1(p);
  ...
}
```

```
main calls fun1
fun1 calls fun2
fun2 calls fun3
```



Implementing Subprograms with Stack-Dynamic Local Variables

- EP is used as the base of the offset addressing of the data contents of the activation record instance -> Parameters and Local variables
- Note that the EP currently being used is not stored in the run-time stack
- Only saved versions are stored in the activation record instances as the dynamic links
- Using the activation record form given in this section, the new actions are as follows
- Caller actions are as follows
    - Create an activation record instance
    - Save the execution status of the current program unit
    - Compute and pass the parameters
    - Pass the return address to the called
    - Transfer control to the called

Implementing Subprograms with Stack-Dynamic Local Variables

- The prologue actions of the called are as follows
  - Save the old EP in the stack as the dynamic link and create the new value
  - Allocate local variables
- The epilogue actions of the called are as follows
  - If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
  - If the subprogram is a function, the functional value is moved to a place accessible to the caller
  - Restore the stack pointer by setting it to the value of the current EP ~~minus one~~=and set the EP to the old dynamic link
  - Restore the execution status of the caller
- A subprogram is active from the time it is called until the time that execution is completed
- At the time it becomes inactive, its local scope ceases to exist and its referencing environment is no longer meaningful
- Therefore, at that time, its activation record instance can be destroyed

Implementing Subprograms with Stack-Dynamic Local Variables

- Parameters are not always transferred in the stack
  - In many compilers for RISC (Reduced Instruction Set Computer) machines, parameters are passed in registers
  - This is because RISC machines normally have many more registers than CISC (Complex Instruction Set Computer) machines
- Assume that parameters are passed in the stack
- It is straightforward to modify this approach for parameters being passed in registers

Implementing Subprograms with Stack-Dynamic Local Variables

An Example Without Recursion

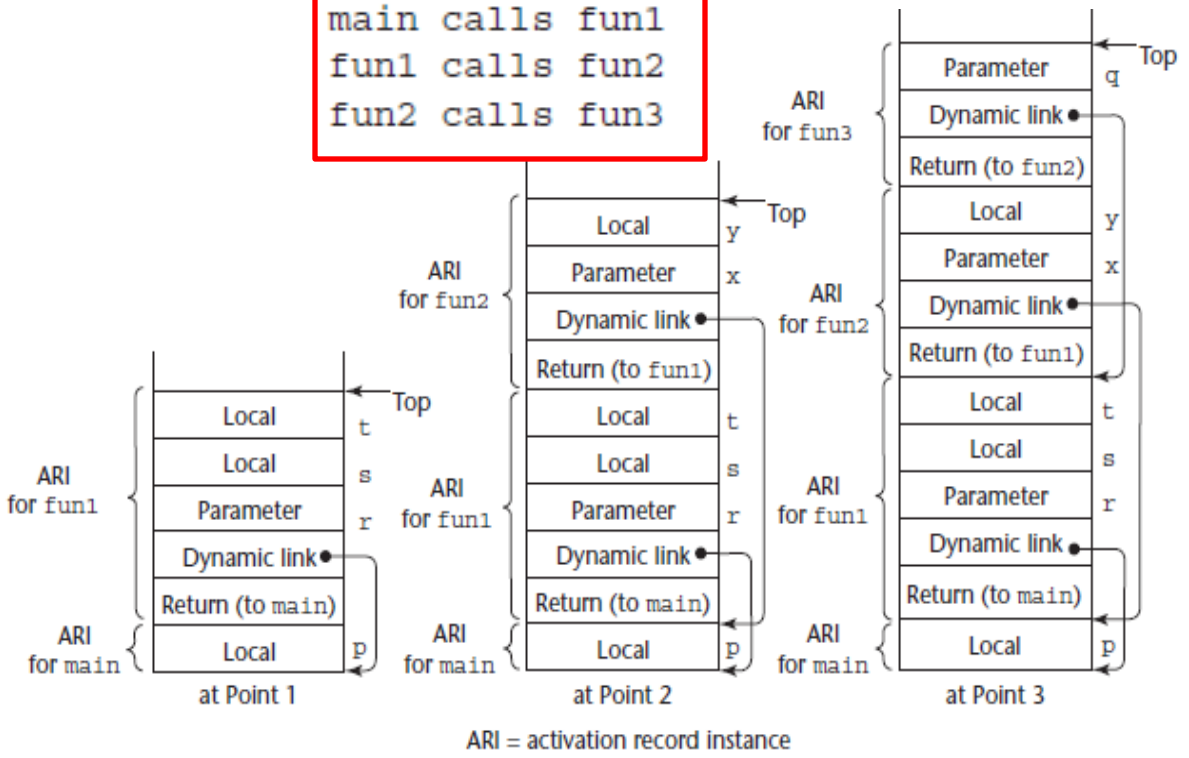
```
void fun1(float r) {
  int s, t;
  ...      <-- 1
  fun2(s);
  ...
}

void fun2(int x) {
  int y;
  ...      <-- 2
  fun3(y);
  ...
}

void fun3(int q) {
  ...      <-- 3
}

void main() {
  float p;
  ...
  fun1(p);
  ...
}
```

```
main calls fun1
fun1 calls fun2
fun2 calls fun3
```



- Some implementations do not actually use an activation record instance on the stack for main functions
- Assume that the stack grows from lower addresses to higher addresses, although in a particular implementation, the stack may grow in the opposite direction

Implementing Subprograms with Stack-Dynamic Local Variables

- Collection of dynamic links present in the stack at a given time is called the dynamic chain or call chain
- It represents the dynamic history of how execution got to its current position, which is always in the subprogram code whose activation record instance is on top of the stack
- References to local variables can be represented in the code as offsets from the beginning of the activation record of the local scope, whose address is stored in the EP
- Such an offset is called a local_offset
- Local_offset of a variable in an activation record can be determined at compile time, using the order, types, and sizes of variables declared in the subprogram associated with the activation record

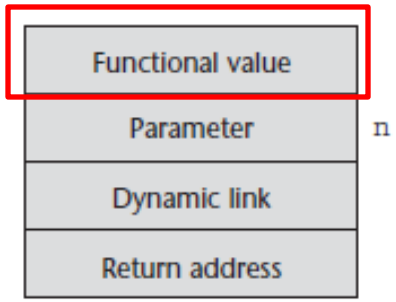
Implementing Subprograms with Stack-Dynamic Local Variables

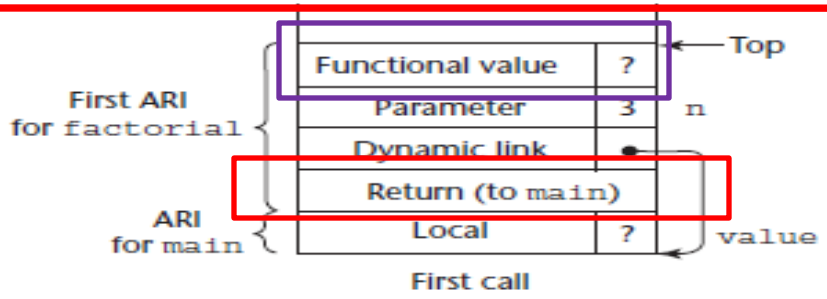
- We assume that all variables take one position in the activation record
  - First local variable declared in a subprogram would be allocated in the activation record two positions plus the number of parameters from the bottom (the first two positions are for the return address and the dynamic link)
  - The second local variable declared would be one position nearer the stack top and so forth
    - **Eg:** In fun1, the local_offset of s is 3; for t it is 4
    - Likewise, in fun2, the local_offset of y is 3
- To get the address of any local variable, the local_offset of the variable is added to the EP

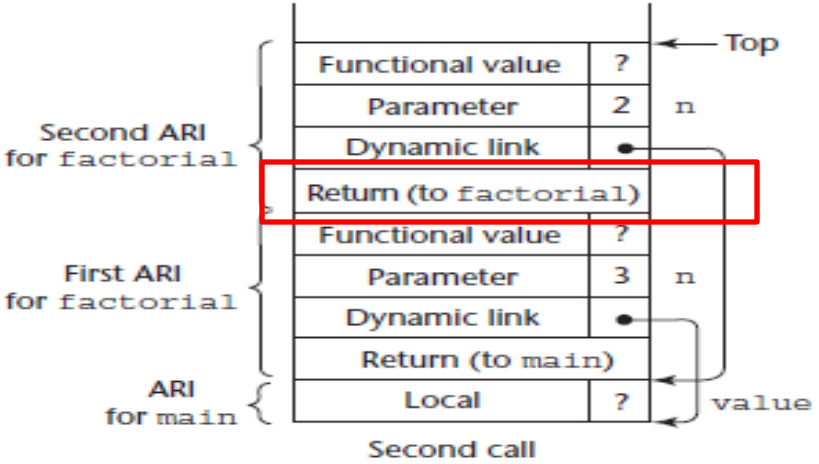
Implementing Subprograms with Stack-Dynamic Local Variables

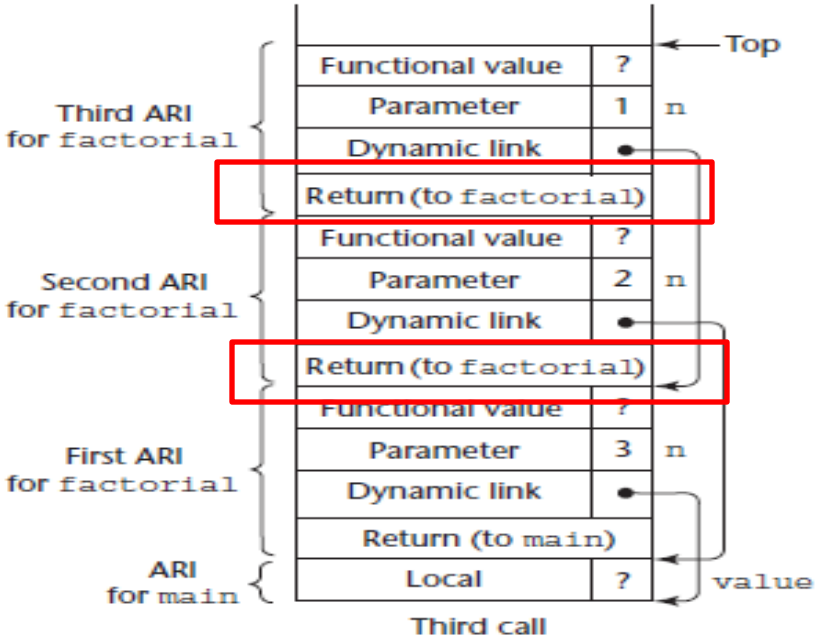
With Recursion

```
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
 }
```

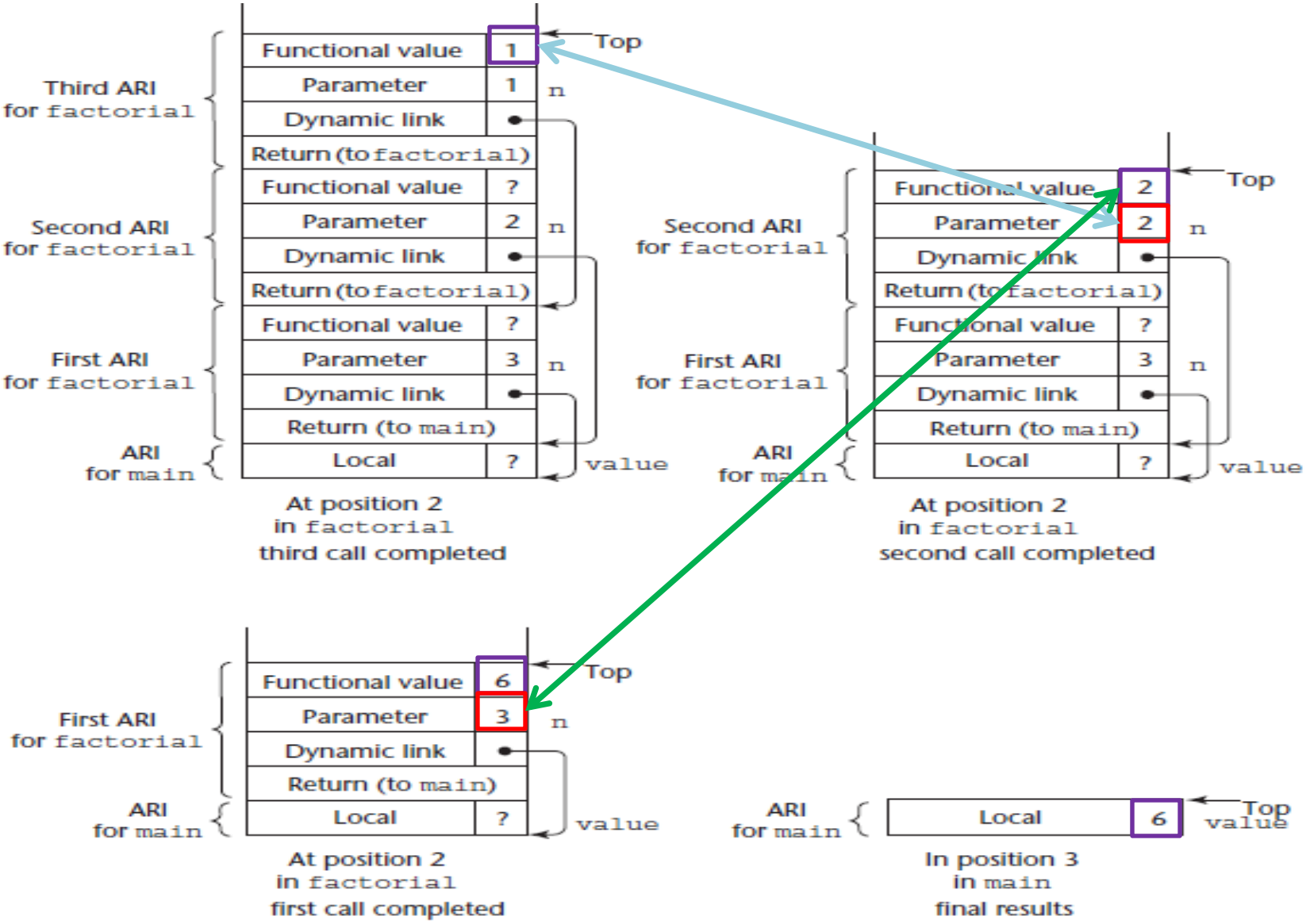








Implementing Subprograms with Stack-Dynamic Local Variables



Nested Subprograms

- Some of the non–C-based static-scoped programming languages use stack-dynamic local variables and allow subprograms to be nested
- Among these are Fortran 95+, Ada, Python, JavaScript, Ruby and Lua, as well as the functional languages

The Basics

- A reference to a nonlocal variable in a static-scoped language with nested subprograms requires a two-step access process
- All nonstatic variables that can be nonlocally accessed are in existing activation record instances and therefore are somewhere in the stack
- First step of the access process is to find the instance of the activation record in the stack in which the variable was allocated
- Second part is to use the local_offset of the variable (within the activation record instance) to access it

Miscellaneous

```
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  -- of Main_2
  ...
  Bigsub;
  ...
end;  -- of Main_2
```

Nested Subprograms

- Finding the correct activation record instance is the more interesting and more difficult of the two steps
- First, note that in a given subprogram, only variables that are declared in static ancestor scopes are visible and can be accessed
- Also, activation record instances of all of the static ancestors are always on the stack when variables in them are referenced by a nested subprogram
- This is guaranteed by the static semantic rules of the static-scoped languages:
  - A subprogram is callable only when all of its static ancestor subprograms are active
  - If a particular static ancestor were not active, its local variables would not be bound to storage, so it would be nonsense to allow access to them
- The semantics of nonlocal references dictates that the correct declaration is the first one found when looking through the enclosing scopes, most closely nested first
- So, to support nonlocal references, it must be possible to find all of the instances of activation records in the stack that correspond to those static ancestors

Nested Subprograms

Static Chains

- Most common way to implement static scoping in languages that allow nested subprograms is static chaining
- A new pointer, called a static link, is added to the activation record
- Static link, which is sometimes called a *static scope pointer,* points to the bottom of the activation record instance of an activation of the static parent
- It is used for accesses to nonlocal variables
- The static link appears in the activation record below the parameters
- The addition of the static link to the activation record requires that local offsets differ from when the static link is not included
- Instead of having two activation record elements before the parameters, there are now three -> Return address, Static link and dynamic link
- A static chain is a chain of static links that connect certain activation record instances in the stack

Nested Subprograms

- Finding the correct activation record instance of a nonlocal variable using static links is relatively straightforward
- Because the nesting of scopes is known at compile time, the compiler can determine not only that a reference is nonlocal but also the length of the static chain that must be followed to reach the activation record instance that contains the nonlocal object
- Let static_depth be an integer associated with a static scope that indicates how deeply it is nested in the outermost scope
- A program unit that is not nested inside any other unit has a static_depth of 0
- If subprogram A is defined in a nonnested program unit, its static_depth is 1
- If subprogram A contains the definition of a nested subprogram B, then B's static_depth is 2

```
# Global scope
...
def f1():
  def f2():
    def f3():
      ...
    # end of f3
    ...
  # end of f2
  ...
# end of f1
```

Nested Subprograms

- The length of the static chain needed to reach the correct activation record instance for a nonlocal reference to a variable X is exactly the difference between the static_depth of the subprogram containing the reference to X and the static_depth of the subprogram containing the declaration for X
  - This difference is called the nesting_depth or chain_offset, of the reference
- The actual reference can be represented by an ordered pair of integers (chain_offset, local_offset), where chain_offset is the number of links to the correct activation record instance
- The static_depths of the **global scope**, **f1**, **f2**, and **f3** are **0**, **1**, **2**, and **3**
  - If procedure f3 references a variable declared in f1, the chain_offset of that reference would be 2 (static_depth of f3 minus the static_depth of f1)
  - If procedure f3 references a variable declared in f2, the chain_offset of that reference would be 1
- References to locals can be handled using the same mechanism, with a chain_offset of 0, but instead of using the static pointer to the activation record instance of the subprogram where the variable was declared as the base address, the EP is used

Nested Subprograms

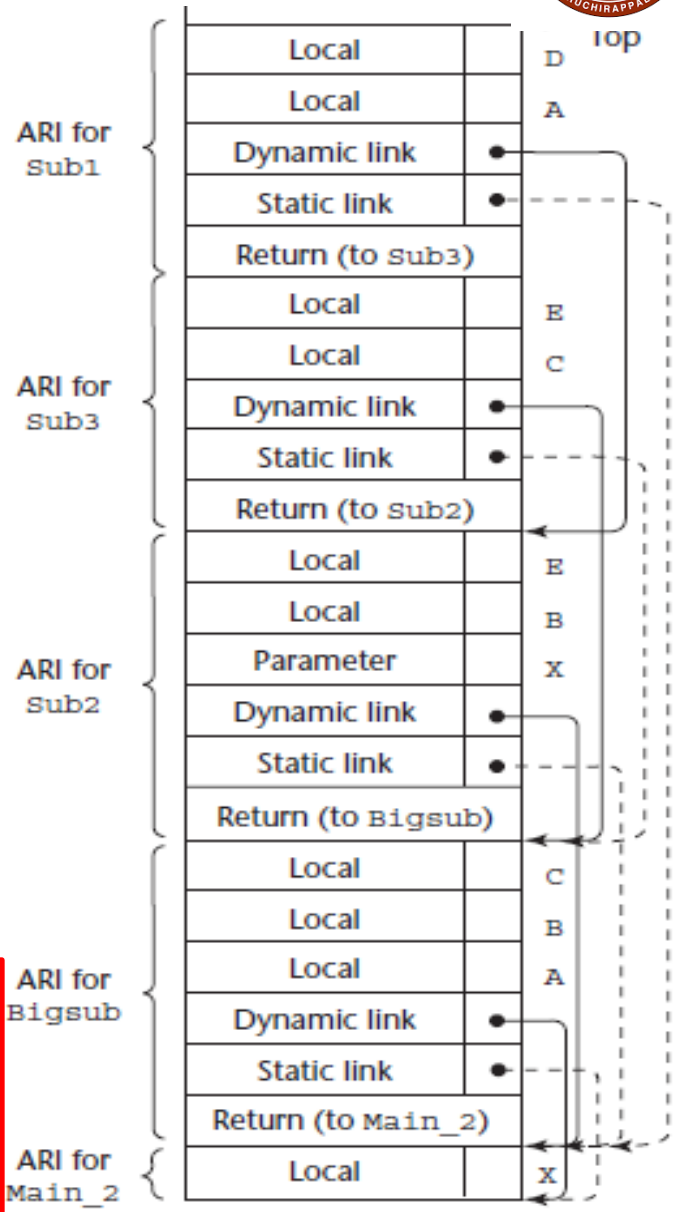
```
procedure Main_2 is                                   0
  X : Integer;
  procedure Bigsub is                                 1
    A, B, C : Integer;
    procedure Sub1 is                                 2
      A, D : Integer;
      begin  -- of Sub1
      A := B + C;  <-------------------------- 1
      . . .
    end;  -- of Sub1
    procedure Sub2(X : Integer) is                    2
      B, E : Integer;
      procedure Sub3 is                               3
        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  -- of Main_2
  . . .
  Bigsub;
  . . .
end;  -- of Main_2
```

Read Page No. from 457 to 460

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

References to the variable A at points 1, 2, and 3 would be represented by the following points:

(0, 3) (local)
(2, 3) (two levels away)
(1, 3) (one level away)



Blocks

- C-based languages, provide for user-specified local scopes for variables called blocks

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

- Lifetime of the variable temp in the preceding block begins when control enters the block and ends when control exits the block
- Advantage of using such a local is that it cannot interfere with any other variable with the same name that is declared elsewhere in the program, or more specifically, in the referencing environment of the block
- Blocks can be implemented by using the static-chain process

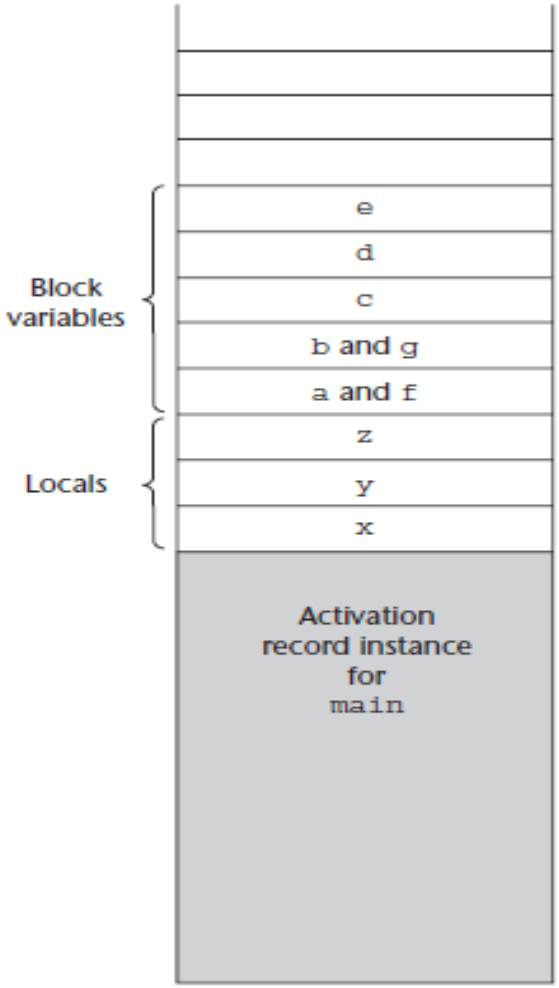
Blocks

- Blocks are treated as parameterless subprograms that are always called from the same place in the program
  - Therefore, every block has an activation record
  - An instance of its activation record is created every time the block is executed
- Maximum amount of storage required for block variables at any time during the execution of a program can be statically determined, because blocks are entered and exited in strictly textual order
- This amount of space can be allocated after the local variables in the activation record
- Offsets for all block variables can be statically computed, so block variables can be addressed exactly as if they were local variables

Blocks

- Note that f and g occupy the same memory locations as a and b, because a and b are popped off the stack when their block is exited (before f and g are allocated)

```
void main() {
  int x, y, z;
  while ( ... ) {
    int a, b, c;
    ...
    while ( ... ) {
      int d, e;
      ...
    }
  }
  while ( ... ) {
    int f, g;
    ...
  }
  ...
}
```



Static vs Dynamic Scoping

```
1  x <- 1
2  f <- function(a) x + a
3  g <- function() {
4    x <- 2
5    f(0)
6  }
7  g() # what does this return?
```

- Under *lexical scoping* (also known as *static scoping*), the scope of a variable is determined by the lexical (*i.e.*, textual) structure of a program
- Under *dynamic scoping*, a variable is bound to the most recent value assigned to that variable

- In Static Scoping, the final result will be 1
- In Dynamic Scoping, the final result will be 2

Implementing Dynamic Scoping

- Two distinct ways in which local variables and nonlocal references to them can be implemented in a dynamic-scoped language -> Deep access; Shallow access
- Deep access and shallow access are not concepts related to deep and shallow binding
- An important difference between binding and access is that deep and shallow bindings result in different semantics
- Deep and shallow accesses do not

Deep Access

- If local variables are stack dynamic and are part of the activation records in a dynamic-scoped language, references to nonlocal variables can be resolved by searching through the activation record instances of the other subprograms that are currently active, beginning with the one most recently activated
- This concept is similar to that of accessing nonlocal variables in a static-scoped language with nested subprograms, except that the dynamic—rather than the static—chain is followed

Implementing Dynamic Scoping

- Dynamic chain, links together all subprogram activation record instances in the reverse of the order in which they were activated
- Therefore, the dynamic chain is exactly what is needed to reference nonlocal variables in a dynamic-scoped language
- This method is called deep access, because access may require searches deep into the stack
- Consider the references to the variables x, u, and v in function sub3
    - Reference to x is found in the activation record instance for sub3
    - Reference to u is found by searching *all of the activation record instances on* the stack, because the only existing variable with that name is in main
    - This search involves following four dynamic links and examining 9 ~~10~~ variable names
    - Reference to v is found in the most recent (nearest on the dynamic chain) activation record instance for the subprogram sub1

Implementing Dynamic Scoping

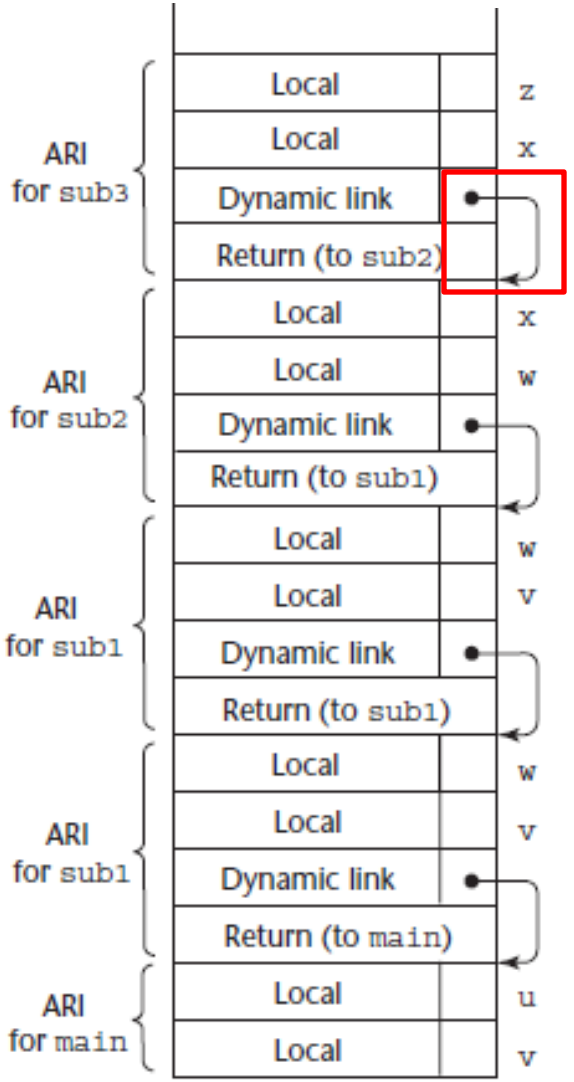
```
void sub3() {
  int x, z;
  x = u + v;
  ...
}

void sub2() {
  int w, x;
  ...
}

void sub1() {
  int v, w;
  ...
}

void main() {
  int v, u;
  ...
}
```

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3



ARI = activation record instance

Implementing Dynamic Scoping

- Two important differences between the deep-access method for nonlocal access in a dynamic-scoped language and the static-chain method for static-scoped languages
  - First, in a dynamic-scoped language, there is no way to determine at compile time the length of the chain that must be searched
    - ✓ Every activation record instance in the chain must be searched until the first instance of the variable is found
    - ✓ This is one reason why dynamic-scoped languages typically have slower execution speeds than static-scoped languages
  - Second, activation records must store the names of variables for the search process, whereas in static-scoped language implementations only the values are required (Names are not required for static scoping, because all variables are represented by the chain_offset, local_offset pairs)

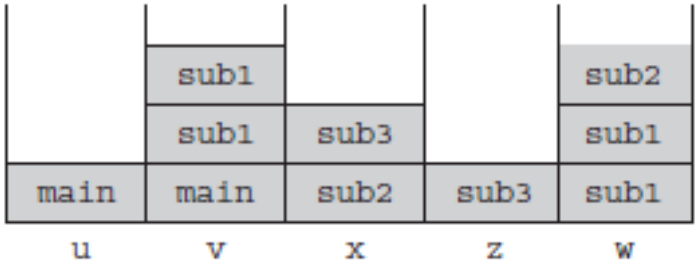
Implementing Dynamic Scoping

- Shallow access is an alternative implementation method
- Variables declared in subprograms are not stored in the activation records of those subprograms
- Because with dynamic scoping there is at most one visible version of a variable of any specific name at a given time, a very different approach can be taken

Method 1

- Have a separate stack for each variable name in a complete program
- Every time a new variable with a particular name is created by a declaration at the beginning of a subprogram that has been called, the variable is given a cell at the top of the stack for its name
- Every reference to the name is to the variable on top of the stack associated with that name, because the top one is the most recently created
- When a subprogram terminates, the lifetimes of its local variables end, and the stacks for those variable names are popped
- This method allows fast references to variables, but maintaining the stacks at the entrances and exits of subprograms is costly

Implementing Dynamic Scoping



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

Central Table

Variable	Value	Active
u	- 5	1
v	15	1
x	20	1
Z	-25	1
w	12	1

Method 2

- Use a central table that has a location for each different variable name in a program
- Along with each entry, a bit called active is maintained that indicates whether the name has a current binding or variable association
- Any access to any variable can then be to an offset into the central table
- Offset is static, so the access can be fast

Miscellaneous

```
void sub3() {
  int x, z;
  x = u + v;
  ...
}

void sub2() {
  int w, x;
  ...
}

void sub1() {
  int v, w;
  ...
}

void main() {
  int v, u;
  ...
}
```

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3

Implementing Dynamic Scoping

- Maintenance of a central table is straightforward
- A subprogram call requires that all of its local variables be logically placed in the central table
- If the position of the new variable in the central table is already active—that is, if it contains a variable whose lifetime has not yet ended (which is indicated by the active bit)—that value must be saved somewhere during the lifetime of the new variable
- Whenever a variable begins its lifetime, the active bit in its central table position must be set
- There have been several variations in the design of the central table and in the way values are stored when they are temporarily replace

Method 2 - Technique 1

- One variation is to have a “hidden” stack on which all saved objects are stored
- Because subprogram calls and returns, and thus the lifetimes of local variables, are nested, this works well

Implement Dynamic Scoping

Central Table

Program

Program + Subprogram

x = 2

Variable	Value	Active
u	- 5	1
v	15	1
x	20	1
Z	-25	1
w	12	1

Hidden Stack

Central Table

Program

Program + Subprogram

Variable	Value	Active
u	- 5	1
v	15	1
x	2	1
Z	-25	1
w	12	1

Hidden Stack

x = 20

Implementing Dynamic Scoping

Method 2 - Technique 2

- A central table of single cells is used, storing only the current version of each variable with a unique name
- Replaced variables are stored in the activation record of the subprogram that created the replacement variable
- This is a stack mechanism, but it uses the stack that already exists, so the new overhead is minimal

Discussion

- Choice between shallow and deep access to nonlocal variables depends on the relative frequencies of subprogram calls and nonlocal references
  - Deep-access method provides fast subprogram linkage, but references to nonlocals, especially references to distant nonlocals (in terms of the call chain), are costly
  - Shallow-access method provides much faster references to nonlocals, especially distant nonlocals, but is more costly in terms of subprogram linkage