Sudan University of Science and Technology

Concepts of programing Languages

Implementing Subprograms

Dr. Aghabi Nabil Abosaif

Topics

- The General Semantics **of Calls** and **Returns**
- Implementing "**Simple**" Subprograms
- Implementing **Subprograms with Stack-Dynamic Local Variables**
- Implementing **Dynamic Scoping**
- **Blocks**

Copyright © 2009 Addison-Wesley. All rights reserved.


The General Semantics of Calls and Returns

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

1. Parameter passing methods.
2. Stack-dynamic allocation of local variables.
3. Save the execution status of calling program.
4. Transfer of control and arrange for the return.
5. If subprogram nesting is supported, access to nonlocal variables must be arranged

Copyright © 2009 Addison-Wesley. All rights reserved.

The General Semantics of Calls and Returns

- General semantics of subprogram **returns:**

1. In mode and inout mode parameters must have their values returned.
2. Deallocation of stack-dynamic locals.
3. Restore the execution status.
4. Return control to the caller.

Copyright © 2009 Addison-Wesley. All rights reserved.

Implementing "Simple" Subprograms: Call Semantics

- "simple" means that subprograms **cannot be nested** and **all local variables are static**.

- **Call Semantics:**

1. Save the execution status of the caller(current program unit).
2. Compute and Pass the parameters
3. Pass the return address to the callee
4. Transfer control to the callee

Copyright © 2009 Addison-Wesley. All rights reserved.

Implementing "Simple" Subprograms: Return Semantics

- **Return Semantics:**

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

Copyright © 2009 Addison-Wesley. All rights reserved.

Implementing "Simple" Subprograms: Return Semantics

- The call and return actions **require storage** for the following:

1. Status information about the caller.
2. Parameters.
3. Return address.
4. Return value for functions

Copyright © 2009 Addison-Wesley. All rights reserved.

Distribution of the call and return actions Simple Subprograms

- The question now, is the distribution of the call and return actions **to the caller** and **the called**?

- In the case of the Call
  - The last three actions 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, third, and fourth actions must be done by the called/callee.
  - The restoration of the execution status of the caller could be done by either the caller or the called.

linkage actions- times

- In general, the linkage actions of the called can occur at two different times, either
  - at the beginning of its execution (**prologue)**
  - Or at the end (**epilogue)**.

- 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.

Implementing "Simple" Subprograms: Parts

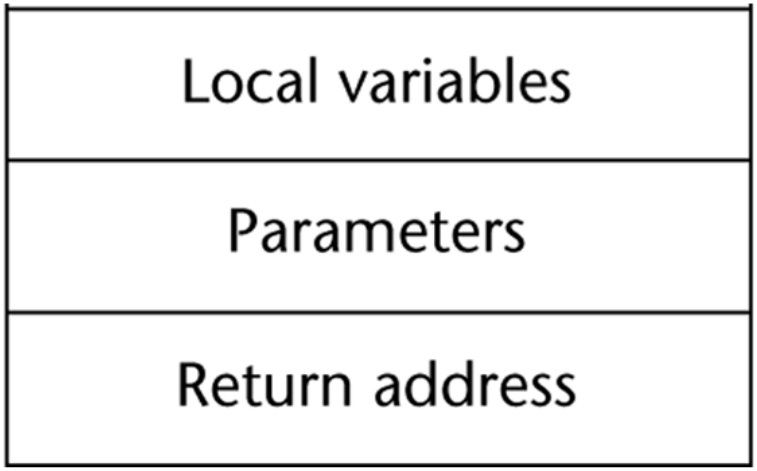
- **Two separate parts:**
  I. The actual code of the subprogram, which is constant.
  II. The non-code part (local variables and data that can change when the subprogram is executed).

- The format, or layout, of the non-code part of an executing subprogram is called **an *activation record,*** *because the data it describes are relevant only during them* ***activation*** *or* ***execution*** *of the subprogram.*

- **An *activation record instance* is** a *concrete example of an activation record (the collection of data for a particular subprogram activation)*
  - *Because languages with simple subprograms do not support recursion, there* ***can be only one active version of a given subprogram at a time.***

Copyright © 2009 Addison-Wesley. All rights reserved.

An Activation Record for "Simple" Subprograms

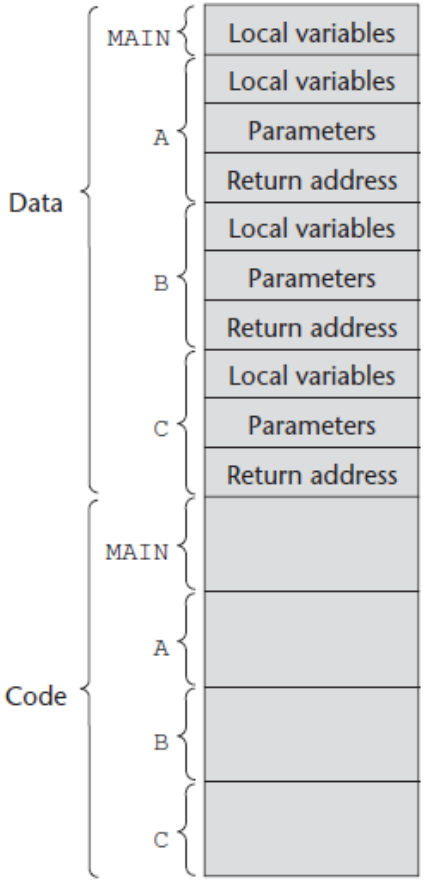


An activation record instance for a "simple" subprogram has **fixed** size, it can be **statically** allocated.

Copyright © 2009 Addison-Wesley. All rights reserved.

The code and activation records of a program with simple subprograms

- A program consisting of a main program and three subprograms: `A`, `B`, and `C`.



Implementing Subprograms with Stack-Dynamic Local Variables

- One of the most important advantages of stack-Dynamic local variables is support for recursion.

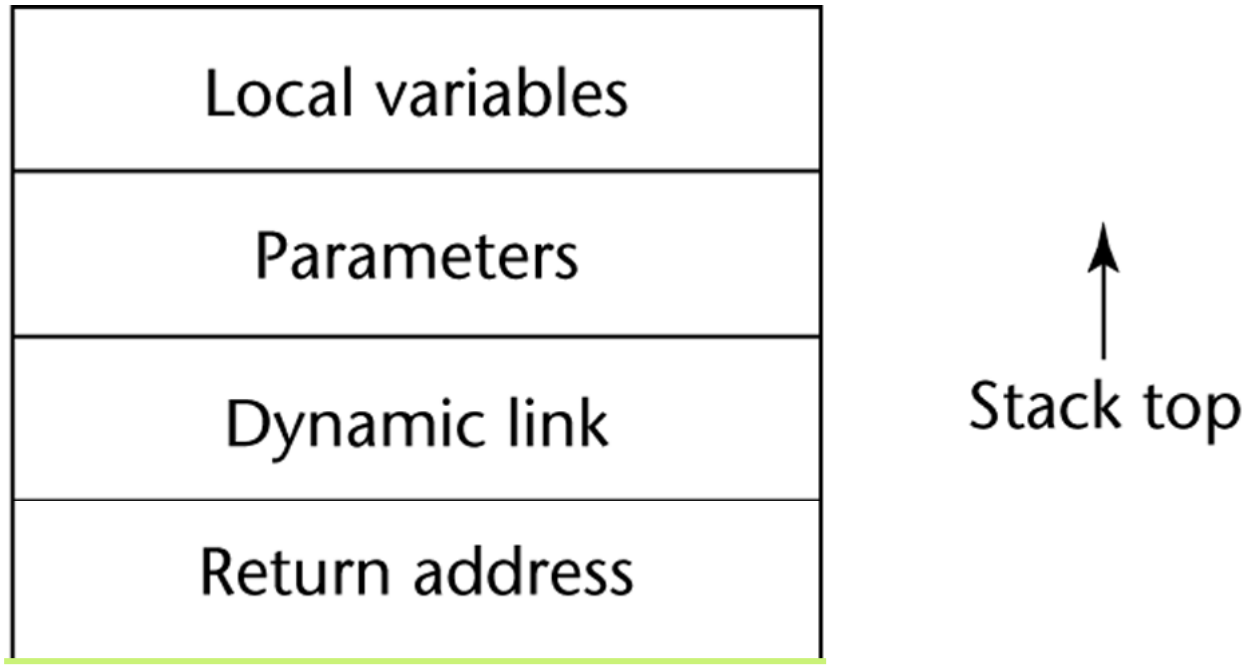
- Subprogram linkage in languages that use stack-Dynamic local variables are more complex than the linkage of simple subprograms for the following reasons:

  1. The compiler must generate code to cause implicit allocation and de-allocation of local variables.

  2. Recursion must be supported (*adds the possibility of multiple simultaneous activations of a subprogram, which means that there can be more than one instance)*

Copyright © 2009 Addison-Wesley. All rights reserved.

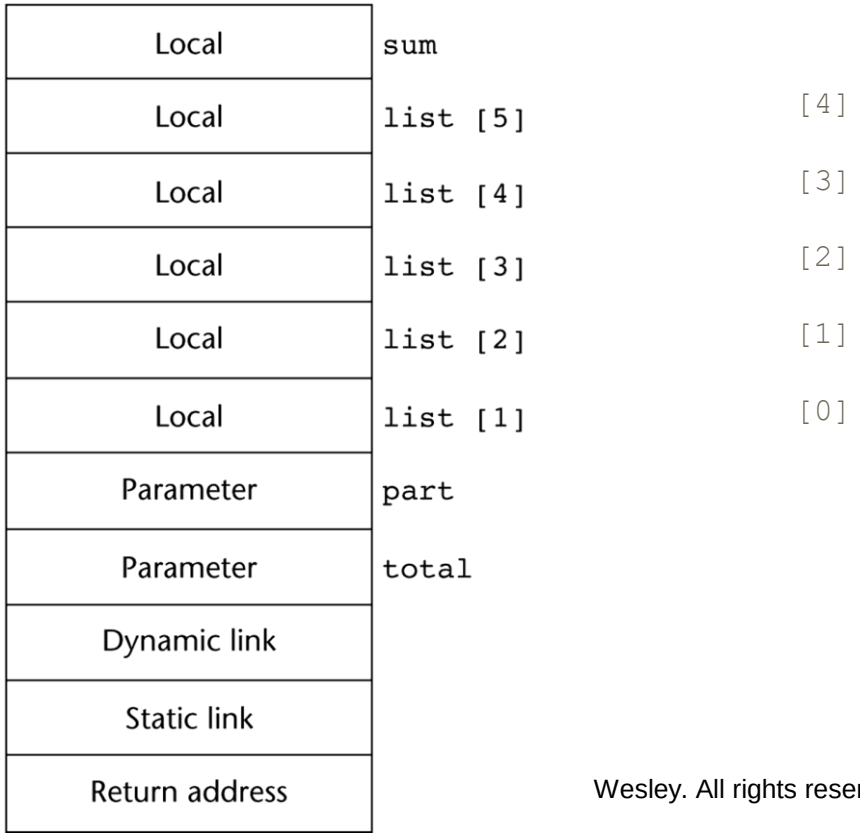
Typical Activation Record for a Language with Stack-Dynamic Local Variables



Copyright © 2009 Addison-Wesley. All rights reserved.

An Example: C Function

```
void sub(float total, int part)
{
  int list[3];
  float sum;
  …
}
```



Wesley. All rights reserved.

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record format is **static(**known at compile time**)**, but *its size may be* **dynamic(**the size of a local array can depend on the value of an actual parameter**).**
- 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.*
- Activation record instances reside on the **run-time stack.**

Copyright © 2009 Addison-Wesley. All rights reserved.

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- Every subprogram activation, whether recursive or non-recursive, 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.
- The execution status needed to resume execution of the calling program unit.
- This includes register values, CPU status bits, and the environment pointer (EP).

Copyright © 2009 Addison-Wesley. All rights reserved.

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- *The 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.*
- The *Environment Pointer* (EP) must be maintained by the **run-time system**.
- It always points at the base of the activation record instance of the currently executing program unit.

Copyright © 2009 Addison-Wesley. All rights reserved.

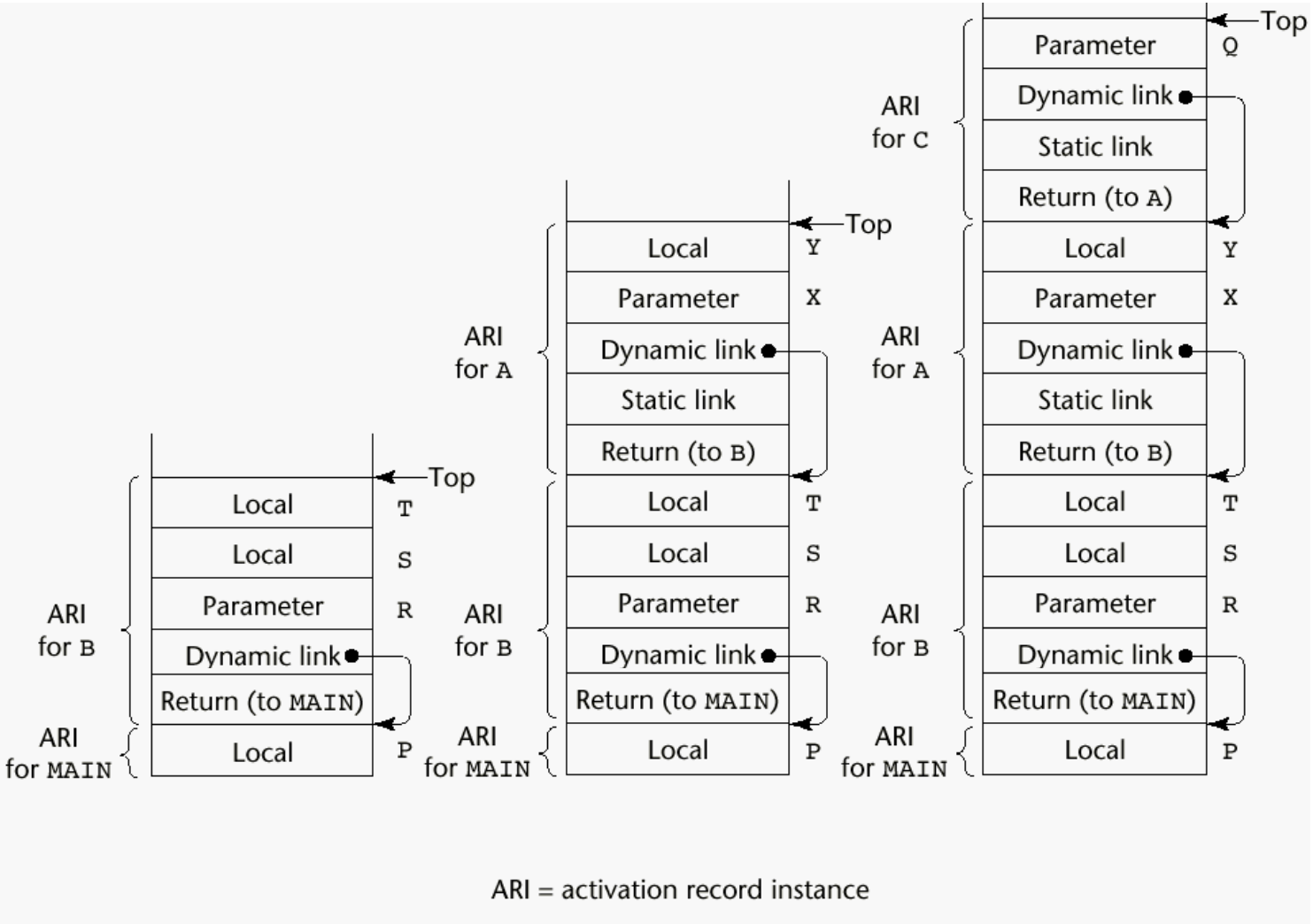
An Example Without Recursion

```
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

Copyright © 2009 Addison-Wesley. All rights reserved.

An Example Without Recursion



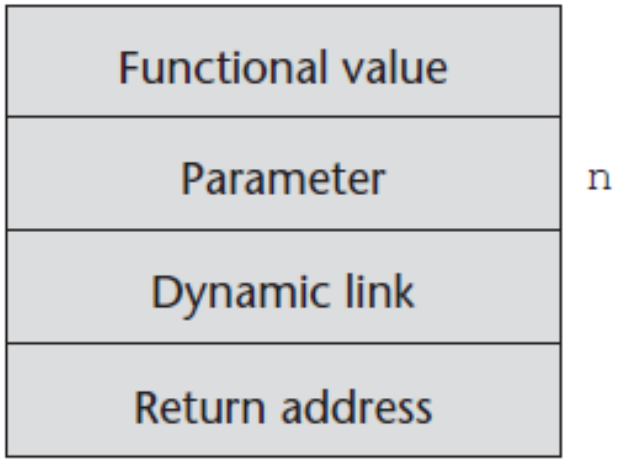
Dynamic Chain and Local Offset

- The collection of dynamic links in the stack at a given time is called the *dynamic chain*, or *call chain*.
- It represents the dynamic history of how execution got to its current position, *which is always 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*.
- This offset is called *a local_offset*.
- The **local_offset** of a local variable can be determined by the compiler at compile time/Run time.

Copyright © 2009 Addison-Wesley. All rights reserved.

An Example With Recursion

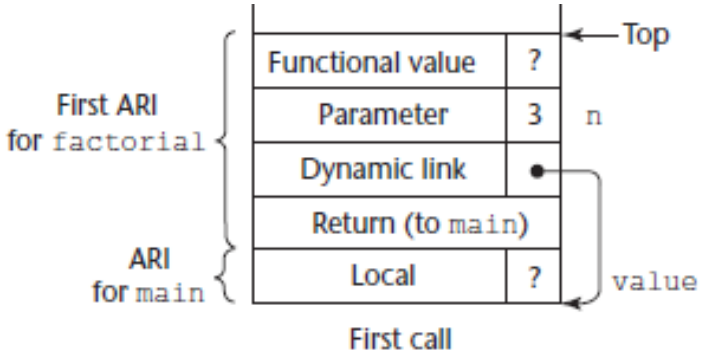
```
int factorial(int n) {
    <---------- 1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <---------- 2
 }
void main() {
  int value;
  value = factorial(3);
    <---------- 3
 }
```



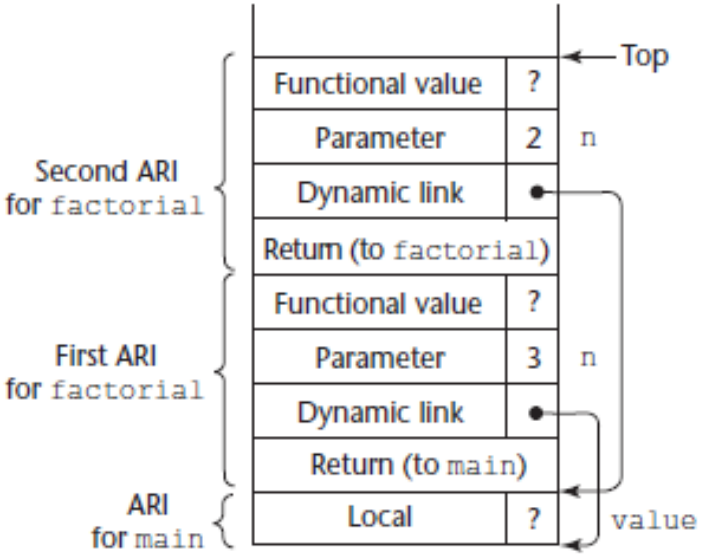
The activation record for `factorial`

An Example With Recursion

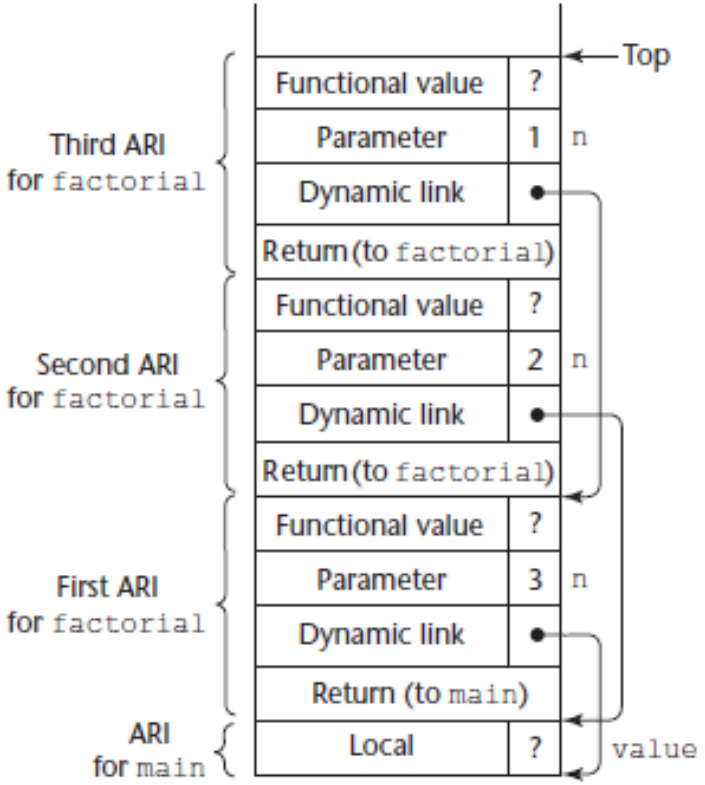
Stack contents at position 1 in `factorial`



First call



Second call



Third call

Nested Subprograms

- Some of programming languages use stack-dynamic local variables allow subprograms to be nested.

- Among these are Fortran 95+ Ada, Python, JavaScript, Ruby, and Lua, as well as the functional languages.

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

Copyright © 2009 Addison-Wesley. All rights reserved.

Static Scoping (continued)

- A reference to a variable can be represented by the pair:

  (**chain_offset**(static/Perent),**local_offset(**dynamic/caller)),

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

Copyright © 2009 Addison-Wesley. All rights reserved.

```
function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
      ...
      a = b + c; <-----------------1
      ...
    }  // end of sub1
    function sub2(x) {
      var b, e;
      function sub3() {
        var c, e;
        ...
        sub1();
        ...
        e = b + a; <---------------2
      } // end of sub3
      ...
      sub3();
      ...
      a = d + e; <-----------------3
    } // end of sub2
    ...
    sub2(7);
    ...
  } // end of bigsub
  ...
  bigsub();
  ...
} // end of main
```

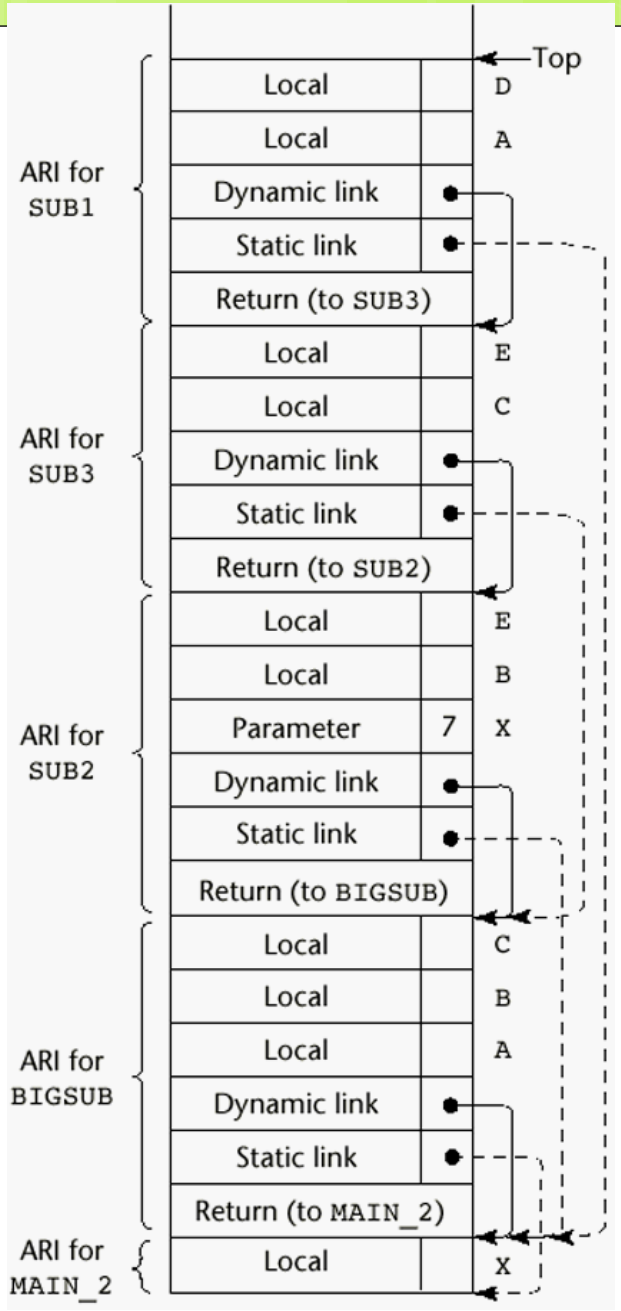
main calls bigsub

bigsub calls sub2

sub2 calls sub3

sub3 calls sub1

Stack Contents at Position 1



Copyright © 2009 Addison-Wesley. All rights reserved.

Static Chain Maintenance

- At the call,
  - 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.

Copyright © 2009 Addison-Wesley. All rights reserved.

Evaluation of Static Chains

- Problems:
  1. A nonlocal reference is slow if the nesting depth is large.

  2. Time-critical code is difficult:
     a. Costs of nonlocal references are difficult to determine.
     b. Code changes can change the nesting depth, and therefore the cost

Copyright © 2009 Addison-Wesley. All rights reserved.

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.

Copyright © 2009 Addison-Wesley. All rights reserved.

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

Copyright © 2009 Addison-Wesley. All rights reserved.

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

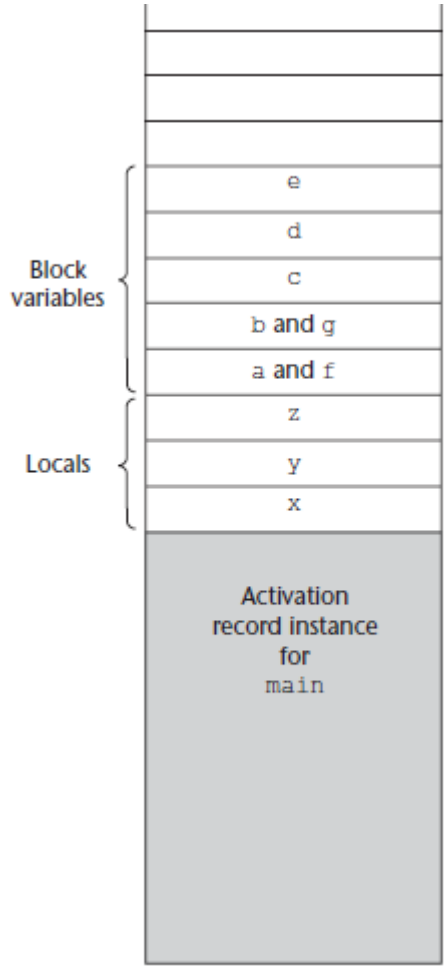
  2. Since the maximum storage required for a block can be statically determined, this amount of space can be **allocated after the local variables in the activation record.**

Copyright © 2009 Addison-Wesley. All rights reserved.

Block variable storage when blocks are not treated as parameter-less procedures

- For this program, the static memory layout could be used.
- 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;
    . . .
  }
  . .
}
```



Implementing Dynamic Scoping

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

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

Copyright © 2009 Addison-Wesley. All rights reserved.

Using Deep Access to Implement Dynamic Scoping

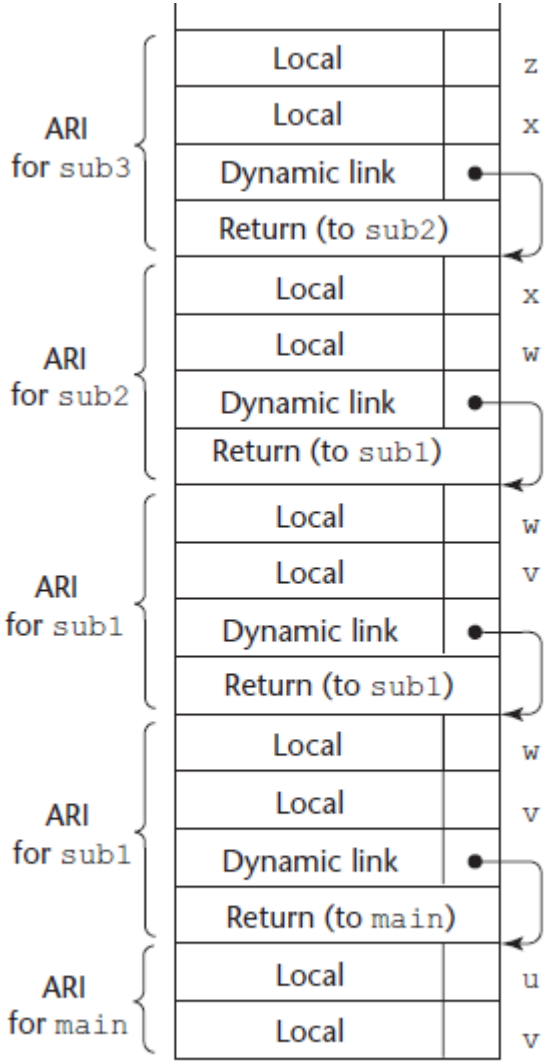
```
void sub3() {
  int x, z;
  x = u + v;
  . . .
}

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

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

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

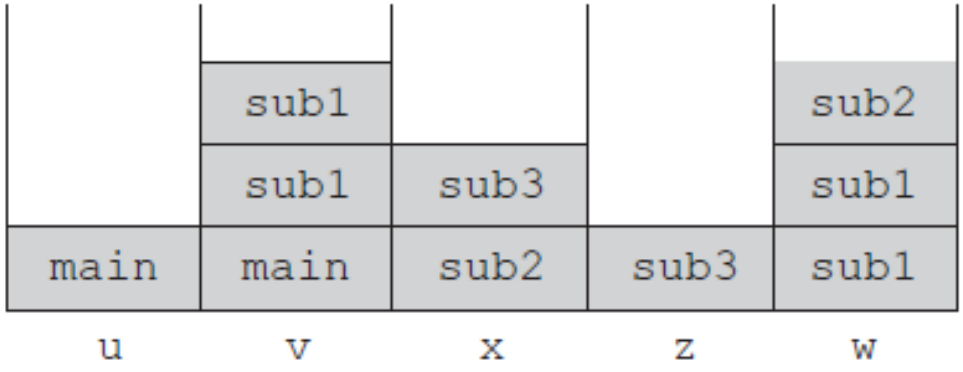
main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3



ARI = activation record instance

Using Shallow Access to Implement 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;
  …
}
```



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

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

Copyright © 2009 Addison-Wesley. All rights reserved.

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 metho.

Copyright © 2009 Addison-Wesley. All rights reserved.

Thank You Your Attention!