10

Implementing Subprograms

The purpose of this chapter is to explore the implementation of subprograms. The discussion will provide the reader with some knowledge of how subprogram linkage works, and also why ALGOL 60 was a challenge to the unsuspecting compiler writers of the early 1960s. We begin with the simplest situation, nonnestable subprograms with static local variables, advance to more complicated subprograms with stack-dynamic local variables, and conclude with nested subprograms with stack-dynamic local variables and static scoping. The increased difficulty of implementing subprograms in languages with nested subprograms is caused by the need to include mechanisms to access nonlocal variables.

The static chain method of accessing nonlocals in static-scoped languages is discussed in detail. Then, techniques for implementing blocks are described. Finally, several methods of implementing nonlocal variable access in a dynamic-scoped language are discussed.

10.1 The General Semantics of Calls and Returns

The subprogram call and return operations are together called **subprogram linkage**. The implementation of subprograms must be based on the semantics of the subprogram linkage of the language being implemented.

A subprogram call in a typical language has numerous actions associated with it. The 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. It must save the execution status of the calling program unit. The execution status is everything needed to resume execution of the calling program unit. This includes register values, CPU status bits, and the environment pointer (EP). The EP, which is further discussed in Section 10.3, is used to access parameters and local variables during the execution of a subprogram. 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. Finally, 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.

The 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.

10.2 Implementing "Simple" Subprograms

We begin with the task of implementing simple subprograms. By "simple" we mean that subprograms cannot be nested and all local variables are static. Early versions of Fortran were examples of languages that had this kind of subprograms.

The semantics of a call to a "simple" subprogram requires the following actions:

1. Save the execution status of the current program unit.
2. Compute and pass the parameters.
3. Pass the return address to the called.
4. Transfer control to the called.

The semantics of a return from a simple subprogram requires the following actions:

1. 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.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. The execution status of the caller is restored.
4. Control is transferred back to the caller.

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.

The question now is the distribution of the call and return actions to the caller and the called. For simple subprograms, the answer is obvious for most of the parts of the process. The 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, third, and fourth actions must be done by the called. Once again, the 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, either at the beginning of its execution or at the end. These are sometimes called the **prologue** and **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.

A simple subprogram consists of two separate parts: the actual code of the subprogram, which is constant, and the local variables and data listed previously, which can change when the subprogram is executed. In the case of simple subprograms, both of these parts have fixed sizes.

The 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. The 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.

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. One possible layout for activation records is shown in Figure 10.1. The saved execution status of the caller is omitted here and in the remainder of this chapter because it is simple and not relevant to the discussion.

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.

Figure 10.2 shows a program consisting of a main program and three subprograms: `A`, `B`, and `C`. Although the figure shows all the code segments separated from all the activation record instances, in some cases, the activation record instances are attached to their associated code segments.

The construction of the complete program shown in Figure 10.2 is not done entirely by the compiler. In fact, if the language allows independent compilation, the four program units—`MAIN`, `A`, `B`, and `C`—may 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. The executable program shown in Figure 10.2 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

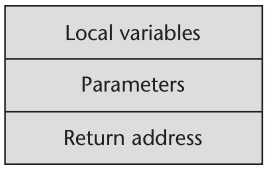


Figure 10.1

An activation record for simple subprogram

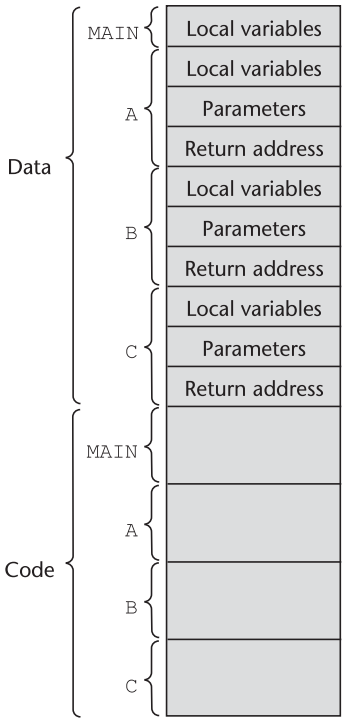


Figure 10.2

The code and activation records of a program with simple subprograms

patch in the target addresses for all calls to `A`, `B`, `C`, and any library subprograms called in `A`, `B`, `C`, and `MAIN`.

10.3 Implementing Subprograms with Stack-Dynamic Local Variables

We now examine the implementation of the subprogram linkage in languages in which locals are stack dynamic, again focusing on the call and return operations.

One of the most important advantages of stack-dynamic local variables is support for recursion. Therefore, languages that use stack-dynamic local variables also support recursion.

A discussion of the additional complexity required when subprograms can be nested is postponed until Section 10.4.

10.3.1 More Complex Activation Records

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

- The 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. The number of activations is limited only by the memory size of the machine. Each activation requires its activation record instance.

The 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. The typical activation record for such a language is shown in Figure 10.3.

Because the return address, dynamic link, and parameters are placed in the activation record instance by the caller, these entries must appear first.

The return address usually consists of a pointer to the instruction following the call in the code segment of the calling program unit. The **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. The 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.

Consider the following skeletal C function:

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

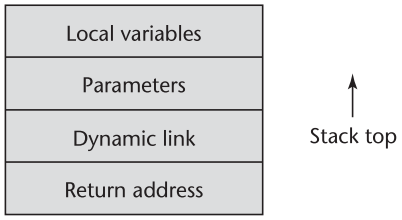


Figure 10.3

A typical activation record for a language with stack-dynamic local variables

The activation record for `sub` is shown in Figure 10.4.

Activating a subprogram requires the dynamic creation of an instance of the activation record for the subprogram. As stated earlier, the 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 run-time 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.

One more thing is required to control the execution of a subprogram—the EP. Initially, the EP points at the base, or first address of the activation record instance of the main program. 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.

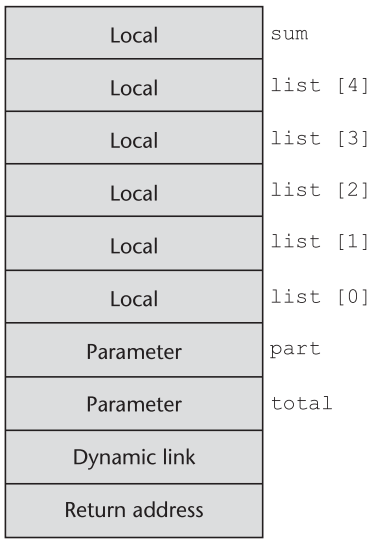


Figure 10.4

The activation record for function `sub`

The 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.

We have now discussed several new actions in the linkage process. The lists given in Section 10.2 must be revised to take these into account. Using the activation record form given in this section, the new actions are as follows:

The caller actions are as follows:

1. Create an activation record instance.
2. Save the execution status of the current program unit.
3. Compute and pass the parameters.
4. Pass the return address to the called.
5. Transfer control to the called.

The prologue actions of the called are as follows:

1. Save the old EP in the stack as the dynamic link and create the new value.
2. Allocate local variables.

The epilogue actions of the called are as follows:

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. 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.
4. Restore the execution status of the caller.
5. Transfer control back to the caller.

Recall from Chapter 9, that 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.

Parameters are not always transferred in the stack. In many compilers for RISC machines, parameters are passed in registers. This is because RISC machines normally have many more registers than CISC machines. In the remainder of this chapter, however, we assume that parameters are passed in

the stack. It is straightforward to modify this approach for parameters being passed in registers.

10.3.2 An Example Without Recursion

Consider the following skeletal C program:

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

The sequence of function calls in this program is

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

The stack contents for the points labeled 1, 2, and 3 are shown in Figure 10.5.

At point 1, only the activation record instances for function `main` and function `fun1` are on the stack. When `fun1` calls `fun2`, an instance of `fun2`'s activation record is created on the stack. When `fun2` calls `fun3`, an instance of `fun3`'s activation record is created on the stack. When `fun3`'s execution ends, the instance of its activation record is removed from the stack, and the EP is used to reset the stack top pointer. Similar processes take place when functions `fun2` and `fun1` terminate. After the return from the call to `fun1` from `main`, the stack has only the instance of the activation record of `main`. Note that some implementations do not actually use an activation record instance on the stack

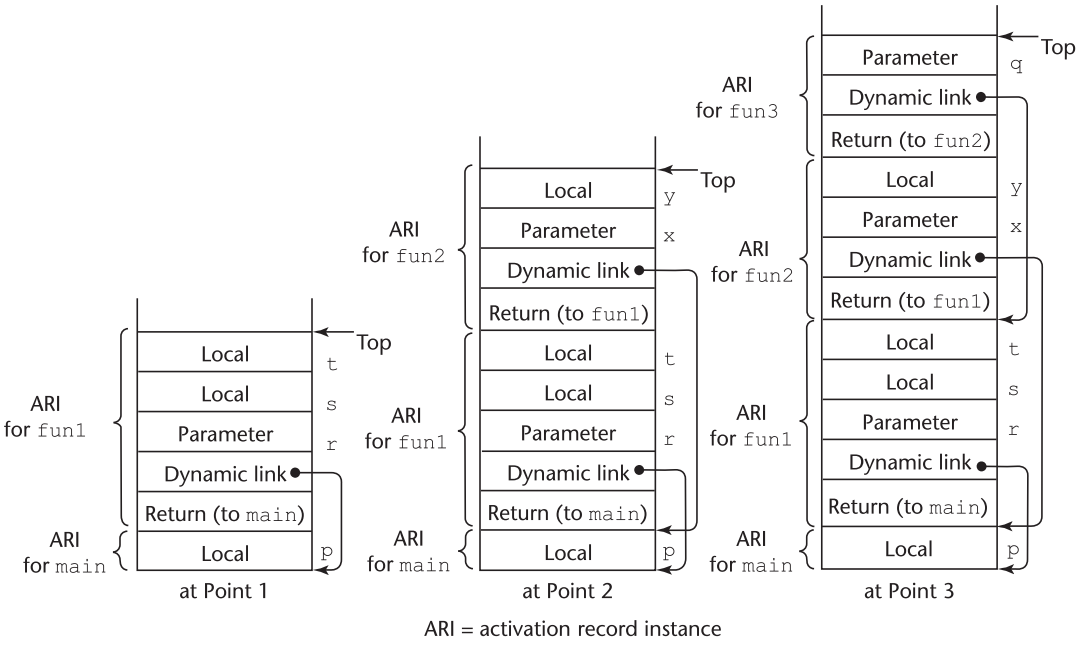


Figure 10.5

Stack contents for three points in a program

for main functions, such as the one shown in the figure. However, it can be done this way, and it simplifies both the implementation and our discussion. In this example and in all others in this chapter, we assume that the stack grows from lower addresses to higher addresses, although in a particular implementation, the stack may grow in the opposite direction.

The 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**.

The 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. To simplify the discussion, we assume that all variables take one position in the activation record. The 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. For example, consider the preceding example program. 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.

10.3.3 Recursion

Consider the following example C program, which uses recursion to compute the factorial function:

```
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 format for the function `factorial` is shown in Figure 10.6. Notice that it has an additional entry for the return value of the function.

Figure 10.7 shows the contents of the stack for the three times execution reaches position 1 in the function `factorial`. Each shows one more activation of the function, with its functional value undefined. The first activation record instance has the return address to the calling function, `main`. The others have a return address to the function itself; these are for the recursive calls.

Figure 10.8 shows the stack contents for the three times that execution reaches position 2 in the function `factorial`. Position 2 is meant to be the time after the **`return`** is executed but before the activation record has been removed from the stack. Recall that the code for the function multiplies the current value of the parameter `n` by the value returned by the recursive call to the function.

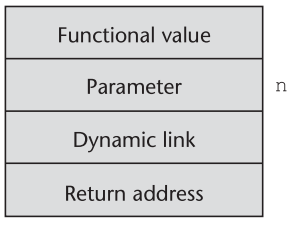


Figure 10.6

The activation record for `factorial`

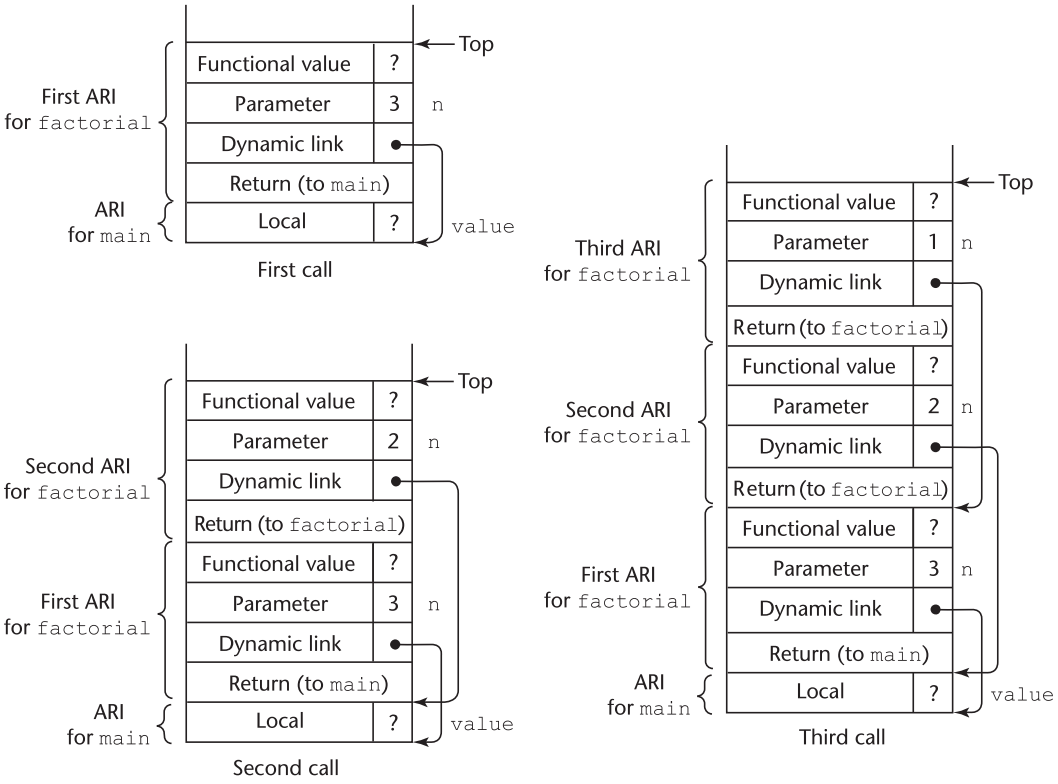


Figure 10.7

Stack contents at position 1 in `factorial`

The first return from `factorial` returns the value 1. The activation record instance for that activation has a value of 1 for its version of the parameter `n`. The result from that multiplication, 1, is returned to the second activation of `factorial` to be multiplied by its parameter value for `n`, which is 2. This step returns the value 2 to the first activation of `factorial` to be multiplied by its parameter value for `n`, which is 3, yielding the final functional value of 6, which is then returned to the first call to `factorial` in `main`.

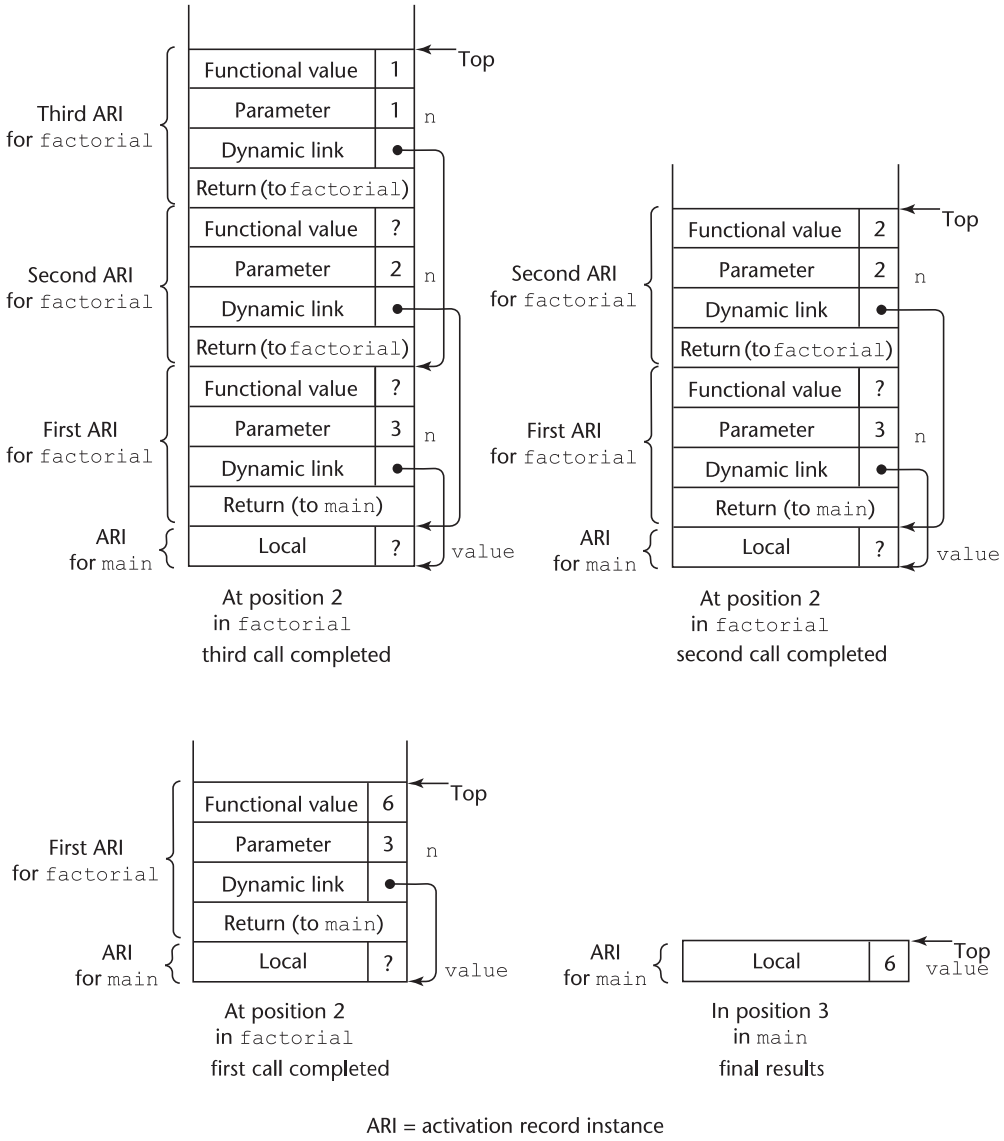


Figure 10.8

Stack contents during execution of `main` and `factorial`

10.4 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. In this section, we examine the most commonly used approach to

implementing subprograms that may be nested. Until the very end of this section, we ignore closures.

10.4.1 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. The first step of the access process is to find the instance of the activation record in the stack in which the variable was allocated. The second part is to use the local_offset of the variable (within the activation record instance) to access it.

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.[1] 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. This observation leads to the implementation approach described in the following subsection.

We do not address the issue of blocks until Section 10.5, so in the remainder of this section, all scopes are assumed to be defined by subprograms. Because functions cannot be nested in the C-based languages (the only static scope in those languages are those created with blocks), the discussions of this section do not apply to those languages directly.

10.4.2 Static Chains

The most common way to implement static scoping in languages that allow nested subprograms is static chaining. In this approach, a new pointer, called a static link, is added to the activation record. The **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. Typically, 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: the return address, the static link, and the dynamic link.

---

1. Closures, of course, violate this rule.

A **static chain** is a chain of static links that connect certain activation record instances in the stack. During the execution of a subprogram `P`, the static link of its activation record instance points to an activation record instance of `P`'s static parent program unit. That instance's static link points in turn to `P`'s static grandparent program unit's activation record instance, if there is one. So, the static chain connects all the static ancestors of an executing subprogram, in order of static parent first. This chain can obviously be used to implement the accesses to nonlocal variables in static-scoped languages.

Finding the correct activation record instance of a nonlocal variable using static links is relatively straightforward. When a reference is made to a nonlocal variable, the activation record instance containing the variable can be found by searching the static chain until a static ancestor activation record instance is found that contains the variable. However, it can be much easier than that. 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.

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 (local_offset is described in Section 10.3.2). For example, consider the following skeletal Python program:

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

The static_depths of the global scope, `f1`, `f2`, and `f3` are 0, 1, 2, and 3, respectively. 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.

To illustrate the complete process of nonlocal accesses, consider the following skeletal JavaScript program:

```
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
```

The sequence of procedure calls is

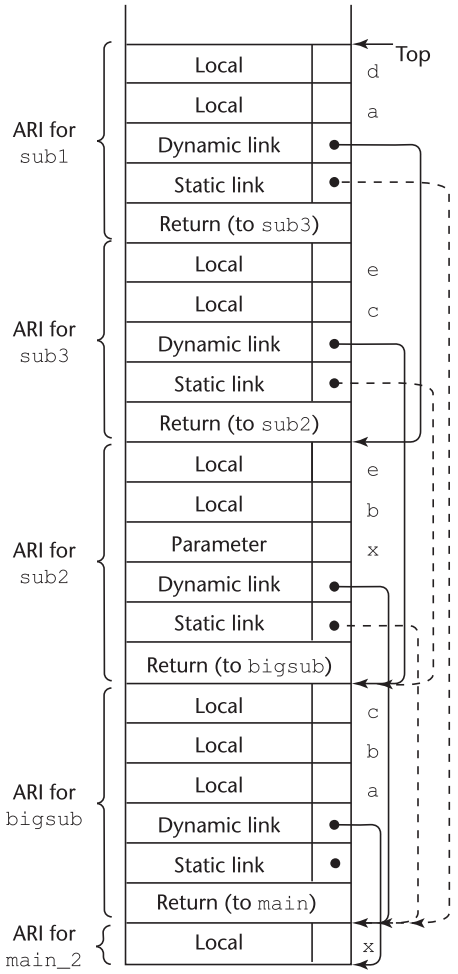
`main` calls `bigsub`
`bigsub` calls `sub2`
`sub2` calls `sub3`
`sub3` calls `sub1`

The stack situation when execution first arrives at point 1 in this program is shown in Figure 10.9.

At position 1 in procedure `sub1`, the reference is to the local variable, `a`, not to the nonlocal variable `a` from `bigsub`. This reference to `a` has the chain_offset/local_offset pair (0, 3). The reference to `b` is to the nonlocal `b` from `bigsub`. It can be represented by the pair (1, 4). The local_offset is 4, because a 3 offset

Figure 10.9

Stack contents at position 1 in the program `main`



ARI = activation record instance

would be the first local variable (`bigsub` has no parameters). Notice that if the dynamic link were used to do a simple search for an activation record instance with a declaration for the variable `b`, it would find the variable `b` declared in `sub2`, which would be incorrect. If the (1, 4) pair were used with the dynamic chain, the variable `e` from `sub3` would be used. The static link, however, points to the activation record for `bigsub`, which has the correct version of `b`. The variable `b` in `sub2` is not in the referencing environment at this point and is (correctly) not accessible. The reference to `c` at point 1 is to the `c` defined in `bigsub`, which is represented by the pair (1, 5).

After `sub1` completes its execution, the activation record instance for `sub1` is removed from the stack, and control returns to `sub3`. The reference to the variable `e` at position 2 in `sub3` is local and uses the pair (0, 4) for access. The

reference to the variable b is to the one declared in sub2, because that is the nearest static ancestor that contains such a declaration. It is accessed with the pair (1, 4). The local_offset is 4 because b is the first variable declared in sub1, and sub2 has one parameter. The reference to the variable a is to the a declared in bigsub, because neither sub3 nor its static parent sub2 has a declaration for a variable named a. It is referenced with the pair (2, 3).

After sub3 completes its execution, the activation record instance for sub3 is removed from the stack, leaving only the activation record instances for main, bigsub, and sub2. At position 3 in sub2, the reference to the variable a is to the a in bigsub, which has the only declaration of a among the active routines. This access is made with the pair (1, 3). At this position, there is no visible scope containing a declaration for the variable d, so this reference to d is a static semantics error. The error would be detected when the compiler attempted to compute the chain_offset/local_offset pair. The reference to e is to the local e in sub2, which can be accessed with the pair (0, 5).

In summary, the 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)

It is reasonable at this point to ask how the static chain is maintained during program execution. If its maintenance is too complex, the fact that it is simple and effective would be unimportant. We assume here that parameters that are subprograms are not implemented.

The static chain must be modified for each subprogram call and return. The return part is trivial: When the subprogram terminates, its activation record instance is removed from the stack. After this removal, the new top activation record instance is that of the unit that called the subprogram whose execution just terminated. Because the static chain from this activation record instance was never changed, it works correctly just as it did before the call to the other subprogram. Therefore, no other action is required.

The action required at a subprogram call is more complex. Although the correct parent scope is easily determined at compile time, the most recent activation record instance of the parent scope must be found at the time of the call. This can be done by looking at activation record instances on the dynamic chain until the first one of the parent scope is found. However, this search can be avoided by treating subprogram declarations and references exactly like variable declarations and references. When the compiler encounters a subprogram call, among other things, it determines the subprogram that declared the called subprogram, which must be a static ancestor of the calling routine. It then computes the nesting_depth, or number of enclosing scopes between the caller and the subprogram that declared the called subprogram. This information is stored and can be accessed by the subprogram call during execution. At the time of the call, the static link of the called subprogram's activation record instance is found by moving down the static chain of the caller. The number

of links in this move is equal to the nesting_depth, which was computed at compile time.

Consider again the program `main` and the stack situation shown in Figure 10.9. At the call to `sub1` in `sub3`, the compiler determines the nesting_depth of `sub3` (the caller) to be two levels inside the procedure that declared the called procedure `sub1`, which is `bigsub`. When the call to `sub1` in `sub3` is executed, this information is used to set the static link of the activation record instance for `sub1`. This static link is set to point to the activation record instance that is pointed to by the second static link in the static chain from the caller's activation record instance. In this case, the caller is `sub3`, whose static link points to its parent's activation record instance (that of `sub2`). The static link of the activation record instance for `sub2` points to the activation record instance for `bigsub`. So, the static link for the new activation record instance for `sub1` is set to point to the activation record instance for `bigsub`.

This method works for all subprogram linkage, except when parameters that are subprograms are involved.

One criticism of using the static chain approach to access nonlocal variables is that references to variables in scopes beyond the static parent cost more than references to locals. The static chain must be followed, one link per enclosing scope from the reference to the declaration. Fortunately, in practice, references to distant nonlocal variables are rare, so this is not a serious problem. Another criticism of the static-chain approach is that it is difficult for a programmer working on a time-critical program to estimate the costs of nonlocal references, because the cost of each reference depends on the depth of nesting between the reference and the scope of declaration. Further complicating this problem is that subsequent code modifications may change nesting depths, thereby changing the timing of some references, both in the changed code and possibly in code far from the changes.

Some alternatives to static chains have been developed, most notably an approach that uses an auxiliary data structure called a **display**. However, none of the alternatives has been found to be superior to the static-chain method, which is still the most widely used approach. Therefore, none of the alternatives are discussed here.

The processes and data structures described in this section correctly implement closures in languages that do not permit functions to return functions and do not allow functions to be assigned to variables. However, they are inadequate for languages that do allow one or both of those operations. Several new mechanisms are needed to implement access to nonlocals in such languages. First, if a subprogram accesses a variable from a nesting but not global scope, that variable cannot be stored only in the activation record of its home scope. That activation record could be deallocated before the subprogram that needs it is activated. Such variables could also be stored in the heap and given unlimited extend (their lifetimes are the lifetime of the whole program). Second, subprograms must have mechanisms to access the nonlocals that are stored in the heap. Third, the heap-allocated variables that are nonlocally accessed must be updated every time their stack versions are updated. Clearly, these are nontrivial extensions to the implementation static scoping using static chains.

10.5 Blocks

Recall from Chapter 5, that a number of languages, including the C-based languages, provide for user-specified local scopes for variables called **blocks**. As an example of a block, consider the following code segment:

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

A block is specified in the C-based languages as a compound statement that begins with one or more data definitions. The lifetime of the variable `temp` in the preceding block begins when control enters the block and ends when control exits the block. The 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 described in Section 10.4 for implementing nested subprograms. 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.

Blocks can also be implemented in a different and somewhat simpler and more efficient way. The 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.

For example, consider the following skeletal program:

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

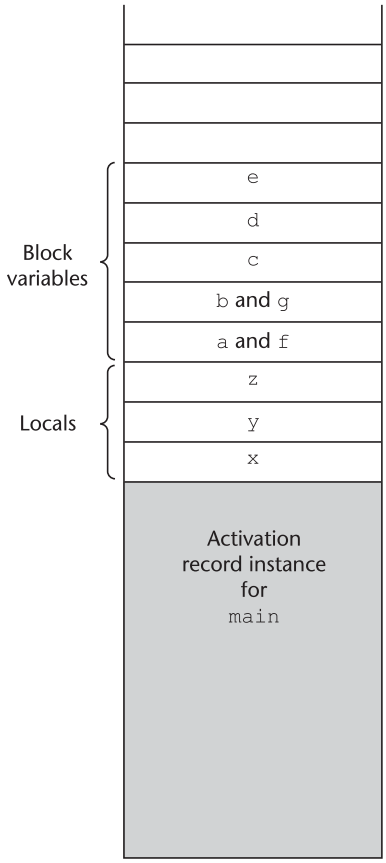


Figure 10.10

Block variable storage when blocks are not treated as parameterless procedures

For this program, the static-memory layout shown in Figure 10.10 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).

10.6 Implementing Dynamic Scoping

There are at least two distinct ways in which local variables and nonlocal references to them can be implemented in a dynamic-scoped language: deep access and shallow access. Note that 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.

10.6.1 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. The 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 following example skeletal program:

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

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

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

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

This program is written in a syntax that gives it the appearance of a program in a C-based language, but it is not meant to be in any particular language. Suppose the following sequence of function calls occurs:

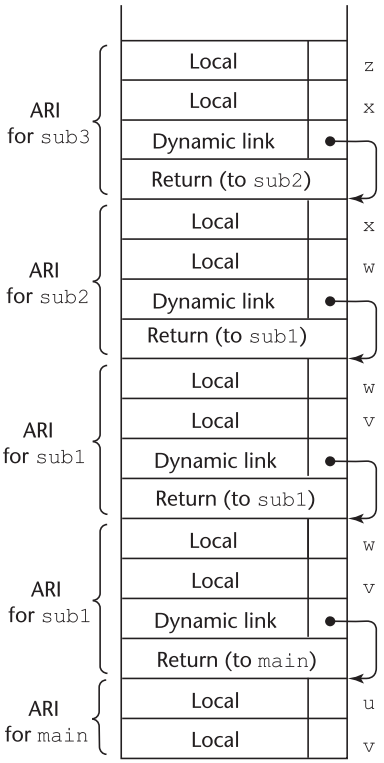
`main` calls `sub1`
`sub1` calls `sub1`
`sub1` calls `sub2`
`sub2` calls `sub3`

Figure 10.11 shows the stack during the execution of function `sub3` after this calling sequence. Notice that the activation record instances do not have static links, which would serve no purpose in a dynamic-scoped language.

Consider the references to the variables `x`, `u`, and `v` in function `sub3`. The reference to `x` is found in the activation record instance for `sub3`. The 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 10 variable names. The reference to `v` is found in the most recent (nearest on the dynamic chain) activation record instance for the subprogram `sub1`.

Figure 10.11

Stack contents for a dynamic-scoped program



There are 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.)

10.6.2 Shallow Access

Shallow access is an alternative implementation method, not an alternative semantics. As stated previously, the semantics of deep access and shallow access are identical. In the shallow-access 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. One variation of

shallow access is to 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.

Figure 10.12 shows the variable stacks for the earlier example program in the same situation as shown with the stack in Figure 10.11.

Another option for implementing shallow access is to 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. The offset is static, so the access can be fast. SNOBOL implementations use the central table implementation technique.

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.

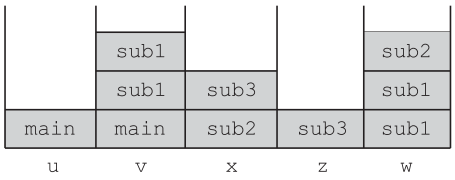
There have been several variations in the design of the central table and in the way values are stored when they are temporarily replaced. One variation is to have a "hidden" stack on which all saved objects are stored. Because subprogram calls and returns, and thus the lifetimes of local variables, are nested, this works well.

The second variation is perhaps the cleanest and least expensive to implement. 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.

The choice between shallow and deep access to nonlocal variables depends on the relative frequencies of subprogram calls and nonlocal references. The deep-access method provides fast subprogram linkage, but references to nonlocals,

Figure 10.12

One method of using shallow access to implement dynamic scoping



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

especially references to distant nonlocals (in terms of the call chain), are costly. The shallow-access method provides much faster references to nonlocals, especially distant nonlocals, but is more costly in terms of subprogram linkage.

SUMMARY

Subprogram linkage semantics requires many actions by the implementation. In the case of "simple" subprograms, these actions are relatively uncomplicated. At the call, the status of execution must be saved, parameters and the return address must be passed to the called subprogram, and control must be transferred. At the return, the values of pass-by-result and pass-by-value-result parameters must be transferred back, as well as the return value if it is a function, execution status must be restored, and control transferred back to the caller. In languages with stack-dynamic local variables and nested subprograms, subprogram linkage is more complex. There may be more than one activation record instance, those instances must be stored on the run-time stack, and static and dynamic links must be maintained in the activation record instances. The static link is to allow references to nonlocal variables in static-scoped languages.

Subprograms in languages with stack-dynamic local variables and nested subprograms have two components: the actual code, which is static, and the activation record, which is stack dynamic. Activation record instances contain the formal parameters and local variables, among other things.

Access to nonlocal variables in a dynamic-scoped language can be implemented by use of the dynamic chain or through some central variable table method. Dynamic chains provide slow accesses but fast calls and returns. The central table methods provide fast accesses but slow calls and returns.

REVIEW QUESTIONS

1. What is the definition used in this chapter for "simple" subprograms?
2. Which of the caller or callee saves execution status information?
3. What must be stored for the linkage to a subprogram?
4. What is the task of a linker?
5. What are the two reasons why implementing subprograms with stack-dynamic local variables is more difficult than implementing simple subprograms?
6. What is the difference between an activation record and an activation record instance?
7. Why are the return address, dynamic link, and parameters placed in the bottom of the activation record?
8. What kind of machines often use registers to pass parameters?

9. What are the two steps in locating a nonlocal variable in a static-scoped language with stack-dynamic local variables and nested subprograms?
10. Define *static chain*, *static_depth*, *nesting_depth*, and *chain_offset*.
11. What is an EP, and what is its purpose?
12. How are references to variables represented in the static-chain method?
13. Name three widely used programming languages that do not allow nested subprograms.
14. What are the two potential problems with the static-chain method?
15. Explain the two methods of implementing blocks.
16. Describe the deep-access method of implementing dynamic scoping.
17. Describe the shallow-access method of implementing dynamic scoping.
18. What are the two differences between the deep-access method for nonlocal access in dynamic-scoped languages and the static-chain method for static-scoped languages?
19. Compare the efficiency of the deep-access method to that of the shallow-access method, in terms of both calls and nonlocal accesses.

PROBLEM SET

1. Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position 1 in the following skeletal program. Assume `bigsub` is at level 1.

```
function bigsub() {
  function a() {
     function b() {
        ... <---------------------------1
     } // end of b
     function  c() {
       ...
       b();
       ...
     } // end of c
     ...
     c();
     ...
  } // end of a
  ...
  a();
  ...
} // end of bigsub
```

2. Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position 1 in the following skeletal program. Assume `bigsub` is at level 1.

```
function bigsub() {
var mysum;
function a() {
 var x;
  function b(sum) {
   var y, z;
   ...
   c(z);
   ...
 } // end of b
 ...
 b(x);
 ...
} // end of a
function c(plums) {
 ... <-----------------------------1
} // end of c
var  l;
...
a();
...
} // end of bigsub
```

3. Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position 1 in the following skeletal program. Assume `bigsub` is at level 1.

```
function bigsub() {
function a(flag) {
  function b() {
    ... a(false);
   ...
 } // end of b
 ...
  if (flag)
   b();
  else c();
 ...
} // end of a
function c() {
  function d() {
   ... <------------------------1
 } // end of d
 ...
 d();
```

```
    ...
  } // end of c
  ...
  a(true);
  ...
} // end of bigsub
```

The calling sequence for this program for execution to reach `d` is

```
bigsub  calls  a
a  calls  b
b  calls  a
a  calls  c
c  calls  d
```

4. Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position 1 in the following skeletal program. This program uses the deep-access method to implement dynamic scoping.

```
void fun1() {
  float a;
  . . .
}

void fun2() {
  int b, c;
  . . . <--------- 1
}

void fun3() {
  float d;
  . . .
}

void main() {
  char e, f, g;
   . . .
}
```

The calling sequence for this program for execution to reach `fun3` is

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

5. Assume that the program of Problem 4 is implemented using the shallow-access method using a stack for each variable name. Show the stacks for the time of the execution of `fun3`, assuming execution found its way to that point through the sequence of calls shown in Problem 4.