6 IMPLEMENTATION OF BLOCK-STRUCTURED LANGUAGES

6.1 ACTIVATION RECORDS AND CONTEXT

Block-Structured Languages Required the Development of New Run-Time Techniques

We have seen in previous chapters that Algol and Pascal contain many features that prevent the use of the run-time structures that are used with FORTRAN. For example, since Algol and Pascal procedures can be recursive, there may be several instances of a procedure active at one time; hence, there must be some provision for the dynamic creation of activation records to hold the state of these instances. Therefore, the static "one activation record per procedure" techniques that we learned in Chapter 2 will not work. Also, we have seen that Pascal provides dynamic memory management by allocating space for the locals of a procedure on a stack. This storage is allocated on procedure entry and deallocated on procedure exit. This means that variables cannot be statically bound to memory locations as is common in FORTRAN and assembly languages. In this chapter we study in depth the implementation techniques required for Pascal since they are applicable to almost all modern languages.

An Activation Record Represents the State of a Procedure

Since the FORTRAN notion of an activation record is not adequate for block-structured languages like Algol and Pascal, it is worthwhile to reanalyze activation records by considering their purposes. Activation records record the state of an activation of a procedure. To know the state of a procedure activation, we need to know the following information:

1. *The code, or algorithm, that makes up the body of the procedure*
2. *The place in that code where this activation of the procedure is now executing*
3. The values of all of the variables visible to this activation

Since item 1, the code, does not vary between instances of execution of the procedure, it does not have to be part of the activation record; the other two items may vary between instances and so must be part of the activation record. The result is that we divide the representation of the state of a procedure into two parts:

1. A fixed *program* part
2. A variable *activation record* part

We will now analyze the parts of an activation record.

The Instruction Part Represents the Site of Control

To know the state of execution of a procedure, we must know both the current statement or expression and the context in which it is to be executed. The first of these is represented by the *ip* or *instruction part* (also, instruction pointer) of the activation record. Typically, this is just a pointer to the next instruction of the procedure to be executed; this is analogous to the IP of our interpreter and is usually contained in the IP register of the computer when the instance is executing.

The Environment Part Represents the Context

It is a familiar fact that the meanings of natural language sentences depend on their *contexts.* For example, the sentence "John shot a buck" means that John spent a dollar when interpreted in the context of John's trip to the store. Conversely, it means that he shot a male deer, when interpreted in the context of John's hunting trip.

The same is the case in programming languages. The FORTRAN statement

```
X = A(I)
```

denotes a subscripting operation in a context in which `A` has been declared to be an array and a function invocation in a context in which `A` has been declared to be a function. It is thus crucial that the interpreter (whether human or computer) interpret programming language constructs in the correct context.

What does this have to do with activation records? In Pascal, procedures are *scope defining* constructs; that is, the statements and expressions inside a procedure are interpreted in a different context from those outside. Therefore, to know the state of a procedure activation, it is not sufficient to know the statement it is currently executing; it is also necessary to know the context in which this statement must execute.

The context is defined by the *ep* or *environment part* (also, environment pointer) of an activation record. The result is that an activation record has these two parts:

1. The *ep* (environment part), which defines the context to be used for this activation of the procedure
2. The *ip* (instruction part), which designates the current instruction being (or to be) executed in this activation of the procedure

Activation Records Contain the Locals

The context of the statements contained in a procedure is simple: It is just the names declared in that procedure together with the names declared in the surrounding procedures. We must add the condition that if any name is declared in more than one of these procedures, then it is the innermost binding that is seen; This is the rule of the contour diagrams. We state this in a more procedural way: If we wish to look up a name, then we look to see if it is in the local environment. If it is, we take this binding; otherwise we look in the surrounding environment. This process continues, looking in outer and more outer surrounding environments until a binding for the name is found; if no binding is found, then the name is undeclared and the program is in error.

To implement recursive procedures and the dynamic storage allocation features of Algol and Pascal, we have seen (Section 3.3) that an activation record is used to hold the local variables and formal parameters of each procedure. These activation records are created and deleted when the corresponding procedures are entered and exited, thereby allocating and deallocating storage for the local variables. This provides immediate access to part of the context—the local variables—since they are stored directly in the activation record. The environment part of the activation record must also make some provision for gaining access to the nonlocal parts of the context. Thus, there are two components to the environment part:

1. The local context
2. The nonlocal context

Our analysis has yielded the following representation for procedure activations:

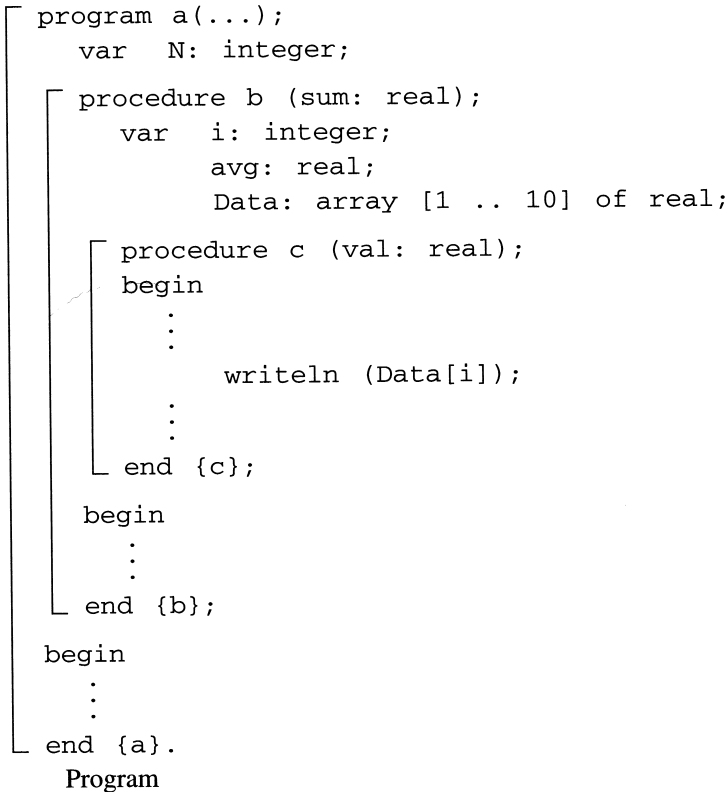
- **I.** Fixed program part
- **II.** Variable activation-record part
  - **A.** instruction part
  - **B.** environment part
    - **1.** local context
    - **2.** nonlocal context

We now develop one means for providing the nonlocal access.

A Static Link Points to the Outer Activation

Every variable is local to some procedure. (The global variables are local to the "main procedure"; in effect, the main program is called by the operating system.) Hence, every variable can be found in the activation record for some procedure. Therefore, to provide access to the nonlocal variables of a procedure, it is necessary to provide access to the activation records of the procedures to which these variables are local.

How can we provide access to the activation record for the surrounding procedure? The simplest approach is to keep a pointer to it, as we can see by looking at a contour diagram. Consider the program in Figure 6.1; the contour diagram shows the context when the procedures (a), (b), and (c) have all been entered. To look up a variable, such as N, we start in



```
program a(...);
   var   N: integer;

   procedure b (sum: real);
      var   i: integer;
            avg: real;
            Data: array [1 .. 10] of real;

      procedure c (val: real);
      begin
         .
         .
            writeln (Data[i]);
         .
         .
      end {c};

   begin
      .
      .
   end {b};

begin
   .
   .
end {a}.
```

Program

(a) N (b) sum i avg Data (c) val EP

Dynamic structure

Figure 6.1 The State of a Program

the current local context, indicated by EP (representing the active local context), and begin looking outward through the surrounding contexts. Thus, we must provide a *link* from each contour to the surrounding one. Together these links form a *chain* leading from the currently active activation (containing EP) to the outermost (or global) activation. This link is called the *static* link (and the chain, the *static* chain), because it reflects the *static* structure of the program, that is, the way the procedures are nested. (Compare the *dynamic chain* discussed in Section 2.3.)

Figure 6.2 shows a typical implementation of the activation records of procedures. The state of execution is presumed to be inside of procedure (c). We can see that there is an activation record on the stack for each of procedures (a), (b), and (c) and that the static link points from each record to the one below it on the stack. Each activation record contains both an `SL` field, containing the static link, and space for the local variables (and other information not shown). The `EP` (environment pointer) register points to the activation record for the currently active context; we will call this the *current* activation record. The `SP` (stack pointer) register always points to the top of the stack. We call the `IP-EP` register pair the *locus of control*, because these two registers together define the instruction and context controlling the computer.

The formats we use for activation records here (or anywhere else in this book) are not sacred; the format chosen for a particular machine will often depend on the instruction set

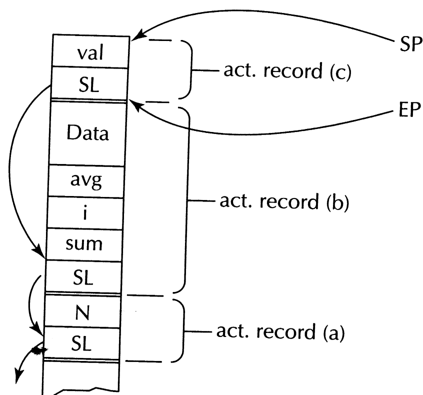


Figure 6.2 Activation Records for Procedures

and other characteristics of that machine. For example, we have chosen to make the static links (including the `EP` register) point at the base of the activation records (specifically, at their static link fields); this makes it easy to chain from one activation record to another when accessing a variable (a process discussed below).

- ***Exercise 6-1:*** Draw the stack and `EP` and `SP` registers when the assignment to `val` is being executed in the program in Figure 6.14 (p. 237).

- ***Exercise 6-2:*** Draw the stack and `EP` and `SP` registers just after the call `Q(P)` is executed [i.e., before `fp(5)` is executed] in the program skeleton in Figure 6.5 (p. 226).

Variables Are Addressed by Two Coordinates

It is probably apparent that it would be very inefficient to carry out this search process literally; it would require variable names to be looked up at run-time every time a variable is referenced. Thus, we must find some way to avoid this overhead. In Chapter 2 we saw that a FORTRAN compiler assigns fixed memory locations to each variable and then uses the addresses of these locations to access the variables at run-time. This binding of variables to locations is done by the compiler and is recorded in its *symbol table.* The symbol table is discarded at the end of the compilation process since all variable references have been replaced by absolute addresses. Here is an example of part of a FORTRAN symbol table.

Name	Type	Location
.	.	.
.	.	.
.	.	.
I	INTEGER	0245
J	INTEGER	0246
K	INTEGER	0247
.	.	.
.	.	.
.	.	.

Static binding will not work for block-structured languages since variables are allocated memory locations at run-time and since there may be several instances of the same variable in existence at the same time. To see how this can be implemented, look again at the contour diagram in Figure 6.1. The variable `val` is local to the procedure (c), so it is contained in the currently active activation record; we do not have to follow the chain to get to it. The variable `sum` is contained in procedure (b), which immediately surrounds (c); we have to follow the static chain a distance of one link to get to the activation record containing `sum`. Finally, `N` is declared in the procedure (a), so we have to follow the static chain a distance of two links to get to the activation record containing `N`. Therefore, if we know the "distance" from the *use* of a variable to its *declaration*, then we can traverse that many links of the static chain to get to the environment of definition of the variable. Hence, we must investigate the determination of this distance.

It is clear that this distance depends on how deeply nested the use of the variable is within the procedure in which the variable is declared. That is, if the procedure of use is two levels deeper than the procedure of declaration, then the distance is two. Some terminology will help to clarify these ideas. We will call the number of levels of **procedure-end** containing a use or declaration of a name the *static nesting level* of that use or declaration. To put it another way, the static nesting level of a use or declaration of a name is the number of contour lines surrounding that use or declaration. For example, the static nesting level of the declaration of `N` is one, of the declaration of `sum` is two, and of the declaration of `val` is three. The static nesting level of the use of `Data` in procedure (c) is three.

The *static distance* between two constructs is the difference between their static nesting levels. For example, since the static nesting level of the use of `Data` in procedure (c) is three and the static nesting level of the declaration of `Data` is two, the static distance between this use of `Data` and its declaration is one. We can see that the static distance between a use of a variable and its declaration tells us how many static links must be traversed to get to the activation record containing that variable.

It is quite easy for the compiler to keep track of this information: It must always know the static nesting level of the procedure it is compiling so it increments this number whenever it encounters a **procedure** and decrements it whenever it encounters a procedure **end**. Then, whenever the compiler processes a declaration, it must record in the symbol table entry for the declared variable the static nesting level (snl) of its declaration. For example, a symbol table for the program in Figure 6.1 might look like this (ignore the 'offset' field for now):

Name	Type	Snl	Offset
`N`	`integer`	1	1
`sum`	`real`	2	1
`i`	`integer`	2	2
`avg`	`real`	2	3
`Data`	`real array`	2	4
`val`	`real`	3	1

For any statement containing a variable, the difference between the static nesting level of that statement and the static nesting level given for that variable in the symbol table is the distance to the activation record containing the variable.

It is not sufficient to know the particular activation record in which a variable resides; it is also necessary to know its position within that activation record. This is the purpose of the *offset* field in the symbol table shown above; it gives the fixed offset, or distance, from the base of the activation record to the variable. Therefore, we can see that we are using a *two-coordinate* method of addressing variables: The first coordinate is the static nesting level of the variable's declaration, which allows us to get to the activation record in which the variable resides. The second coordinate is the `offset`, or position of the variable within that activation record. Notice that although the position of the activation record may vary at run-time, the position of the variable within that activation record is fixed. We will find that many objects in block-structured languages are addressed by these two coordinates:

1. An *environment pointer* (the static nesting level, in this case), which allows us to get to the activation record for the environment of definition of the object
2. A *relative offset* (the variable offset, in this case), which allows us to access the desired object within its activation record

■ ***Exercise 6-3:*** Compute the static nesting levels of all identifiers (i.e., variable and procedure names) in the programs in Figures 6.5 and 6.14. Compute the static nesting levels of each use of a name and the static distances between the uses and declarations of the names in these programs. Finally, compute the offsets of the variables in these programs.

Accessing a Variable Requires Two Steps

Since two coordinates are needed to locate a variable, it is natural that two steps are required to access a variable: (1) The activation record for the environment of definition must be located. (2) The variable must be located within this activation record. More specifically.

1. At run-time skip down the static chain the number of links given by the static distance to get to the activation record in which the variable resides. The static distance between the use and the declaration of the variable is a constant computed by the compiler.
2. The address of the variable is obtained by adding the fixed offset of the variable to the address obtained in step 1. This offset is also a constant computed by the compiler.

To see how this might actually be implemented on a computer, we will suppose the formats shown in Figure 6.2. Now, suppose that we wish to access a local variable, say `val`, from within procedure (c). Since it is local, the static distance to its activation record is zero so we don't have to follow the static chain at all. Therefore, the address of `val` is just `EP + 1`, where 1 is the fixed offset recorded in the symbol table entry for `val`. The general case can be symbolized:

```
fetch M[EP + offset(v)]
```

where v is any local variable and `offset(v)` is the constant offset recorded for v in the symbol table.

Next consider accessing `sum`, which is at a static distance of one. This means that one

link of the static chain must be traversed to get the address of sum's activation record. We will hold this address in a temporary register AP (activation record pointer):

```
AP := M[EP];       traverse static link
fetch M[AP + 1];   access the variable
```

since 1 is the offset of sum.

Finally, let's consider the code to access N, which is at a static distance of two. In this case, two links of the static chain must be traversed:

```
AP := M[EP];       traverse first static link
AP := M[AP];       traverse second static link
fetch M[AP + 1];   access the variable
```

since 1 is the offset of N.

Let's summarize the steps to access a variable. If the variable is at a static distance of zero, then it can be accessed directly by `M[EP + offset(v)]`. If it is at a static distance greater than zero ($sd > 0$), then the steps required are

```
           AP := M[EP]; traverse first static link
(sd - 1) × AP := M[AP]; traverse remaining static links
           fetch M[AP + offset(v)]; access the variable
```

where *sd* is the static distance. The meaning of the second line above is `(sd - 1)` duplicates of the instruction `AP := M[AP]`. If $sd = 1$, this instruction is omitted and if $sd = 0$, the first two instructions are omitted and EP is used instead of AP.

We pause to analyze the performance of this method of accessing variables. Notice that each traversal of a static link requires one memory reference (ignoring any memory references required to decode the instructions themselves), so *sd* memory references are required to get to the activation record. An additional memory reference is required to read or write the variable once it is located so the total memory references required to access a variable by this method is $sd + 1$. (The exact count may vary from machine to machine.) Therefore, it can be quite expensive to access variables at a long static distance, although access to local variables is quite inexpensive. It has been observed that programs most frequently reference local variables and global variables (i.e., variables declared in the innermost and outermost procedures), therefore, in a deeply nested program, the average time to access a variable could become excessive. This is important because the time required to access the variables often dominates the running time of a program. Later in this chapter, we will discuss another way of implementing variable accesses that is better in this regard.

6.2 PROCEDURE CALL AND RETURN

Activation Records Represent the State of an Activation

In Chapter 2 (Section 2.3), we discussed the implementation of FORTRAN subprograms and we saw that the *state* of each subprogram was represented in an *activation record*, which held all of the information necessary to characterize the state of the computation in progress.

This included (1) the storage for the procedure's parameters, local variables, and temporaries; (2) the resumption address of the subprogram; and (3) a *dynamic link*, or pointer to the caller's activation record.

These activation records can be easily adapted to accommodate block-structured, recursive languages. Combining the needs of environmental access under block structure and recursive procedure call and return implies that a procedure activation record must have these parts:

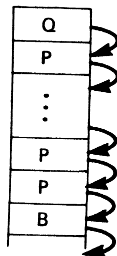
1. The *ep* (environment part), which defines the context to be used for this activation of the procedure and comprises the following:
   a. The *parameters* and *local variables*, which are the innermost (local) scope
   b. The *static link*, which provides access to the surrounding (nonlocal) scope
2. The *ip* (instruction part), which designates the current instruction being (or to be) executed in this activation of the procedure (essentially the *resumption address* of Chapter 2)
3. The *dynamic link*, which points to the activation record of the caller of this activation of this procedure (i.e., the *activator* of this activation) and allows us to restore the state of the caller upon procedure exit

Procedures Require Both Static and Dynamic Links

We have described above two links for a procedure—the static link and the dynamic link. Is it possible to combine these two links into one? Are they really the same thing? After all, they both point to activation records lower down on the stack. In fact, two separate links *are* required.

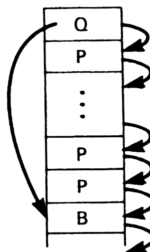
Recall that Algol and Pascal use *static scoping*; that is, a procedure executes in the environment of its definition rather than the environment of its caller (see Chapter 3, Section 3.3). The dynamic link field of a procedure's activation record, as we have described it in Chapter 2, points to the previous activation record on the stack, which is the activation record of the caller. Therefore, if we follow the dynamic chain, we will not get to the correct environment for the procedure; we will get the environment of the caller rather than the environment of definition.

Furthermore, the environment of definition is not at any fixed location down the dynamic chain. To see this, suppose that two procedures P and Q are defined in the same procedure B and that P calls itself recursively a number of times before it calls Q. When Q has been called, the stack looks like this (the arrows represent the dynamic chain):



Notice that Q's environment of definition is B. There is no way that the compiler can know how many of P's activation records are between Q's and B's since this number depends on the dynamic behavior of P. In other words, there is no simple way that Q can get to its context, B, via the dynamic chain.

One straightforward solution to this problem is to provide an explicit pointer from Q to its environment of definition as illustrated below:



This pointer is the static link.

Procedure Activation Has Three Steps

In Chapter 2 (Section 2.3) we saw that four steps are required to invoke a FORTRAN subprogram:

1. Transmit the parameters to the callee.
2. Save the caller's state in the caller's activation record.
3. Establish the dynamic link from the callee to the caller.
4. Enter the callee at its first instruction.

These can be rearranged into the following basic functions that must be accomplished to activate a procedure:

1. The state of the caller must be saved (step 2 above).
2. An activation record for the callee must be created (steps 1 and 3 above).
3. The callee must be entered in the context of the new activation record (step 4 above).

That is, to *deactivate* the caller and *activate* the callee, it is necessary to (1) suspend the caller into its activation record, (2) initialize the callee's activation record, and (3) transfer the locus of control from the caller to the callee.

Saving the Caller's State

Let's consider the first of these steps, saving the state of the caller. The state of the caller has two major components—the instruction part (*ip*) and the environment part (*ep*). The *ip*

is the address at which the caller must resume execution after the callee exits so we must store this address in the IP part of the caller's activation record,[1] that is,

```
M[EP].IP := resume;    save resume location
```

The EP register always points to the activation record of the currently active procedure (i.e., the caller).

The second component of the caller's state is the environment part, which is in turn composed of the locals and the nonlocals. The access to the nonlocals is already saved in the static link of the caller's activation record so no further work is required on the nonlocals' account. The locals are contained in the caller's activation record so they are also safe.

Have we accounted for all of the caller's context? It would seem so since we have ensured that both the locals and the nonlocals are saved. Unfortunately, we have taken care of only the *programmer visible* environment; there are other variables, such as temporary locations and the machine's registers, that must really be considered part of the local context. After all, their contents affect the meaning of the machine code instructions in the procedure. We saw in the FORTRAN call sequence that a necessary step was saving these temporaries. Since the details are, of necessity, very machine dependent, we ignore saving and restoring the temporary locations and machine registers in the rest of this chapter.

Creating the Callee's Activation Record

The second of the steps in activating a procedure is to create a properly initialized activation record for the callee and to install this activation record as the new active context. What is required to accomplish this? We have seen that a procedure activation record has these parts:

- PAR parameters
- SL static link
- IP resumption address
- DL dynamic link

Therefore, each of these parts must be properly initialized. Also, to install this activation record as the new active context, a pointer to the activation record must be placed in the EP register (which points to the beginning of the static chain, see Section 6.1).

Next, we will consider each of these steps in the order listed above, although they do not necessarily have to be performed in that order. In fact, the best order for the operations usually depends on the particular arrangement chosen for the activation record, which varies from machine to machine. There are a few *logical* dependencies among the above steps that we will note as they occur.

[1] Note that there are IP and EP *registers*, which are related to, but not the same as, the IP and EP *fields* in the activation record.

The Actuals Are Put in the Parameter Part

The parameter part of the activation record (PAR) contains the parameters to this activation: the parameter's value in the case of *value* parameters, its address in the case of *reference* parameters, and a pointer to a *thunk* (Chapter 3, Section 3.5) in the case of *name* parameters. Therefore, each actual parameter must be evaluated to yield either a value or an address, as appropriate, and this result must be stored in the appropriate place in PAR. Therefore, if *callee* represents the address of the new activation record for the callee, then the parameter transmission process can be written:

```
M[callee].PAR[1]  :=  evaluation of parameter 1;
M[callee].PAR[2]  :=  evaluation of parameter 2;
                   .
                   .
                   .
M[callee].PAR[n]  :=  evaluation of parameter n;
```

We will determine the value of *callee* later.

Whenever we see that something is to be evaluated, as in the above code sequence, we must ask ourselves: "In what environment should this thing be evaluated"? Clearly, since the actual parameters are *written* in the context of the caller, they should also be *evaluated* in the context of the caller. This is the context that the programmer was assuming when the call was written, and we do not want to violate these assumptions. Since the parameters must be evaluated in the context of the caller, the parameter-part initialization must be done before the new activation record (*callee*) is installed as the new active context (i.e., before the EP register is altered). This is one of those constraints on the ordering of the steps that we mentioned above.

The Static Link Is Set to the Environment of Definition

The static link (SL) is the next part of a procedure activation record that must be initialized. By definition, the static link points to the *environment of definition* of the procedure. How do we get to this environment? We faced a similar problem in accessing a variable (Section 6.1) since it was necessary first to get to the *environment of definition* of the variable before its contents could be accessed. This was done by following (at run-time) the static chain for the number of links given by the *static distance* (computed at compile-time) between the use of the variable and its definition. This same technique works for procedures. At compile-time we must record in the symbol table entry for each procedure the static nesting level of that procedure's definition. Then, when a procedure call is being compiled, the compiler subtracts the static nesting level of the definition from the static nesting level of the call; this gives the static distance between the call and the definition and, hence, the distance down the static chain to the environment of definition.

We summarize the operations to initialize the static link in the following instructions. If the procedure is defined in the current context (i.e., the static distance from call to definition is zero), then this code suffices:

```
M[callee].SL  :=  EP;     set the static link to this context
```

In this case the environment of definition is the environment of the caller.

If the distance from the call to the definition is $sd > 0$, then it is first necessary to traverse the static chain to get to the environment of definition:

```
                AP := M[EP];        traverse first static link
(sd - 1) × AP := M[AP];             traverse remaining static links
       M[callee].SL := AP;          set the static link
```

Notice the similarity between this and the code for accessing a nonlocal variable. This is so because the Pascal scope rules apply to both variables and procedures so the same process is required to get to the context in which either kind of name is defined.

The Final Steps Are Simple

Initializing the rest of the callee's activation record is very simple. First, consider the instruction part (`IP`): There is no reason to initialize this field now since it will be used only when (and if) the *callee* becomes caller by calling a procedure.

The dynamic link (`DL`) field is also simple to initialize; since it points to the activation record of the caller, which is contained in the `EP` register, the following code suffices:

```
M[callee].DL := EP;     set dynamic link
```

Clearly, this code must be executed before the `EP` register is altered to refer to the callee's activation record; this is another example of a constraint between the steps.

The next step is to install the callee's activation record as the new active context. Since the `EP` register always points to the activation record of the active context (the currently active procedure), this is accomplished by

```
EP := callee;     install new AR
```

The final steps are to allocate space for the activation record on the stack and to enter the callee at its first instruction:

```
SP := SP + size(callee's AR);     allocate callee's AR
goto entry(callee);               enter the callee
```

Both the size of the callee's activation record and the address of the caller's entry point are constants known to the compiler.[2]

The last instruction in the above code sequence is in effect a store into the `IP` register. That is, **`goto`** `entry(`*callee*`)` is equivalent to `IP := entry(`*callee*`)`. The transfer of the locus of control is effected by the assignments to `IP` and `EP` [i.e., `EP :=` *callee* and `IP := entry(`*callee*`)`].

The steps to call a Pascal procedure are summarized in Figure 6.3. We have improved the code sequence slightly: Since the `EP` register has already been saved, we can scan down the static chain with *sd* repetitions of `EP := M[EP]`. Also, since `SP` points to the next

[2] This is true for Pascal. Some languages, such as Algol, have dynamic arrays, which means that the size of the activation records can vary at run-time. Compilers for these languages must produce code to compute the activation record size.

```
         M[SP].PAR[1]  := eval. par. 1;
                  .
                  .                                transmit parameters
                  .
         M[SP].PAR[n]  := eval. par. n;
         M[EP].IP  := resume;                      save resume location
         M[SP].DL  := EP;                          set dynamic link
    sd × EP  := M[EP];                             scan down static chain
         M[SP].SL  := EP;                          set static link
         EP  := SP;                                install new AR
         SP  := SP + size(callee's AR);            allocate callee's AR
         goto entry(callee);                       enter the callee
    resume:                                        resume location
```

Figure 6.3 Procedure Call Sequence with Static Chain

available stack location, it can be used as the base of the callee's activation record, so *callee* = SP.

We can estimate the number of memory references in a procedure call by looking at the code sequence in Figure 6.3. There is one reference to save the caller's IP, *sd* references to traverse the static chain, and two references to set the static and dynamic links. (We do not count the time to store the *n* parameters.) Thus, a call costs $sd + 3$ memory references. Notice that the cost of a call (like the cost of a variable access) depends heavily on the static distance to the procedure's declaration. Thus, it will be relatively expensive to call global procedures from inner procedures (a common situation). We investigate solutions to this later (Section 6.3).

■ ***Exercise 6-4:*** Draw the state of the stack and registers after each of the steps in Figure 6.3.

■ ***Exercise 6-5:*** The instruction sequence of Figure 6.3 is duplicated for every procedure call. If some of these instructions were made part of the code of the callee, they would not have to be duplicated over and over. Rearrange the code of Figure 6.3 to minimize the number of instructions that must be duplicated for each call.

Procedure Exit Reverses Procedure Entry

The code for returning from a procedure must reverse the effects of the call. That is, the locus of control must be transferred from the callee back to the caller. In other words, the callee must be *deactivated* and the caller *reactivated.* A return is generally simpler than a call since things are being thrown away rather than created. Two tasks must be accomplished:

1. Delete the callee's activation record.
2. Restore the state of the caller.

In practice these two steps must be interleaved since the information required to restore the caller's state (namely, the dynamic link) is in the callee's activation record.

Deleting the callee's activation record is accomplished by subtracting from the stack pointer the size of the callee's activation record:

```
SP := SP - size(callee's AR);          delete callee's AR
```

Reinstalling the caller's context as the active context is accomplished by loading the `EP` register from the dynamic link of the callee:

```
EP := M[EP].DL;     reactive caller's AR
```

Since `EP` now points to the caller's activation record, we can use it to resume execution of the caller:

```
goto M[EP].IP;     resume execution
```

The **goto** is in effect `IP := M[EP].IP`, so these last two steps return the locus of control (`EP-IP` pair) to the caller. Figure 6.4 summarizes the code for returning from a procedure. This would be compiled either at the end of the procedure body of Pascal procedures, or for each **return**-statement for languages that have **return**-statements. Since memory must be referenced for the dynamic link and the `IP` field, a return requires two memory references.

■ ***Exercise 6-6:*** Draw the state of the stack and registers after each of the instructions in Figure 6.4.

Procedural Parameters Are Represented by Closures

Recall that Algol and Pascal (and many other languages) allow procedures and functions to be passed as parameters to other procedures. For an example of *procedural parameters*, see the program in Figure 6.5. In this case, we have a procedure `Q` that takes another procedure (corresponding to the formal parameter `fp`) as a parameter. We can see two calls on `Q`: one in which it is passed the procedure `P` and another in which it is passed `T`. There are two problems we must solve: (1) What exactly is it that is passed to `Q` to represent `P` or `T`? (2) How is the indirect call `fp(5)` in the body of `Q` implemented?

We will consider the second question first since this will help us to answer the first question. Let's assume for a moment that a call on a formal procedure, such as `fp(5)`, is implemented like any other call, that is, with the code sequence in Figure 6.3. Does this work? The first three steps (transmitting the parameters, saving the resume location, and setting the dynamic link) are all fine. The next step, scanning down the static chain for the environment of definition, cannot be done, however, because we need to know the static distance from the call to the environment of definition, which requires us to know the static nesting level of the environment of definition. For a normal procedure call this is simple to determine during compilation; it is recorded in the symbol table entry for the procedure. In a call on a formal pro-

```
SP := SP - size(callee's AR);          delete callee's AR
EP := M[EP].DL;                        reactivate caller's AR
goto M[EP].IP;                         resume execution
```

Figure 6.4 Procedure Return with Static Chain

```
program A;
  procedure P (x: integer);
  begin
     ⋮
  end {P};
  procedure Q (procedure fp (n: integer) );
  begin
     fp(5);
  end {Q};
  procedure R;
    procedure T (x: integer);
    begin
       ⋮
    end {T};
  begin
     Q(P);
     Q(T);
  end {R};
begin
   R;
end.
```

Figure 6.5 Example of Procedural Parameters

cedure, such as `fp(5)`, we need to know the static nesting level of the corresponding actual procedure (`P` or `T`, in this example). Unfortunately, this can vary at run-time from one call of `Q` to another. In fact, `T`'s environment of definition is not even in the active static chain when it is called from `Q` by `fp(5)`. Thus, we can see that part of the information that must be passed to `Q` is the environment of definition of the corresponding actual procedure.

■ ***Exercise 6-7:*** Explain why the activation record for `T`'s environment of definition is not in the active static chain when it is invoked from `Q` by `fp(5)`. Draw the stack and all the static and dynamic links at the time of call.

We can take a direct solution to this problem and represent a procedural actual parameter as a two-element record:

1. The *ip* (or *instruction pointer*) field contains the entry address of the actual procedure.
2. The *ep* (or *environment pointer*) field contains a pointer to the environment of definition of the actual procedure.

Such a record is called an *ep-ip pair*, or *closure.*

If *fp* represents the location in an activation record of the closure passed for a procedural parameter, then *fp*.EP is the *ep* part of this closure. (Note that *fp* is a parameter that must be accessed like any other parameter, that is, by using the variable accessing method described in Section 6.1.) Hence, the static link in the callee's activation record can be set by

```
M[SP].SL := fp.EP;      set static link
```

This solves the problem of accessing the callee's environment of definition.

Consider again the code sequence in Figure 6.3. There is another problem. The second to the last step allocates space for the callee's activation record. This requires knowing the size of this activation record, which depends on the number of local variables declared in the corresponding actual procedure. One solution is to pass this information along with the closure; a simpler solution is to make this allocation instruction the first instruction of each procedure. At this point the compiler knows exactly how much space is needed.

The final step, entering the procedure, also requires access to the closure since different procedural actuals have different entry points. Thus the closure specifies the entry address of the argument procedure (*ip*) and the context in which it must execute (*ep*).

The resulting code sequence for calling a formal procedure is shown in Figure 6.6. Five memory references are required (including two for accessing the *ip* and *ep* of the procedure). Procedure exit must be the same as for normal procedures, Figure 6.4, since a procedure may be called both directly and as a parameter.

How is the closure (the *ep*-*ip* pair) for a procedural actual parameter constructed? The *ip* part is just the entry point to the procedure, which is a constant determined by the compiler. The *ep* part is the environment of definition of the procedure, which is accessed along the static chain using the static nesting level in the procedure's symbol table entry. For example, to construct the *ep*-*ip* pair for P in the call Q(P), we must follow one link of the static chain. The code for constructing a closure in M[SP].PAR[1] is

```
M[SP].PAR[1].IP := entry(P);    build ip part
AP := M[EP].SL;                 get environment of definition
M[SP].PAR[1].EP := AP;          build ep part
```

In general, if *sd* is the static distance between the declaration of a procedure P and a call that uses P as the *i*th actual parameter, then the code to build the *ep*-*ip* pair for this parameter is

```
        M[SP].PAR[1]  := eval. par. 1;
                  .
                  .                       transmit parameters
                  .
        M[SP].PAR[n]  := eval. par. n;
        M[EP].IP := resume;              save resume location
        M[SP].DL := EP;                  set dynamic link
        M[SP].SL := fp.EP;               set static link
        EP := SP;                        install callee's AR
        goto fp.IP;                      enter the callee
resume:                                  resume location
```

Figure 6.6 Calling a Formal Procedure

```
               M[SP].PAR[i].IP := entry(P);   build ip part
               AP := M[EP].SL;                traverse first
                                                static link
(sd - 1) × AP := M[AP].SL;                    traverse remaining
                                                static links
               M[SP].PAR[i].EP := AP;         build ep part
```

Notice that $sd + 2$ memory references are required: *sd* to access the environment of definition and 2 to store the *ep* and *ip*.

Pascal, Algol, and many other languages allow procedures and functions to be passed as arguments to other procedures and functions. Considerations of regularity and symmetry may lead us to ask if procedures and functions can be returned as results from other procedures and functions. Allowing functions to accept and return other functions (that is, treating functions and procedures as *first-class citizens*) leads to a very powerful style of programming, called *functional programming*, which is discussed in Chapter 10. Unfortunately, most languages do not permit function-valued or procedure-valued functions. The reason is simple. Suppose a function F were to return a procedure P local to F. Observe that the environment of definition of P is that activation of F and that whenever P is called it must execute in that environment. Unfortunately, when F returns, its activation record is deleted from the stack, so the environment of definition of P is destroyed. To implement procedure-valued functions properly, it would be necessary to ensure that the activation record for F be retained so long as P (or any other procedure local to F) could be called. It is not possible to accomplish this with simple stacked activation records of the kind we have described. Functional programming languages must use a different discipline for the allocation and deallocation of activation records.

■ ***Exercise 6-8**:*** Describe a discipline for the allocation and deallocation of activation records that properly implements function-valued functions (and procedure-valued functions, etc.).

Nonlocal gotos Require Environment Restoration

One other construct found in Algol and Pascal (and most other block structured languages) must manipulate the run-time stack: the **goto**. Why is this? Local **goto**s (i.e., **goto**s to a label local to the block or procedure containing the **goto**) are easy; they are implemented as simple machine jumps. Nonlocal **goto**s (i.e., **goto**s to a label declared in a block or procedure surrounding the block or procedure containing the **goto**) must restore the environment to that of the label, otherwise the stack will not be in the state expected at the destination of the jump. That is, the EP and SP registers must be restored to the values appropriate to the environment of definition of the label. This is analogous to calling a procedure in its environment of definition. Consider the Pascal program in Figure 6.7; the stack and registers just prior to and after the '**goto** 1' are shown in Figure 6.8. We can summarize these observations by saying that a **goto** transfers the locus of control from the **goto** to the label. Both the site (IP) and context (EP) of execution must be altered.

Restoring the context at the destination of the **goto** requires restoring the EP and SP

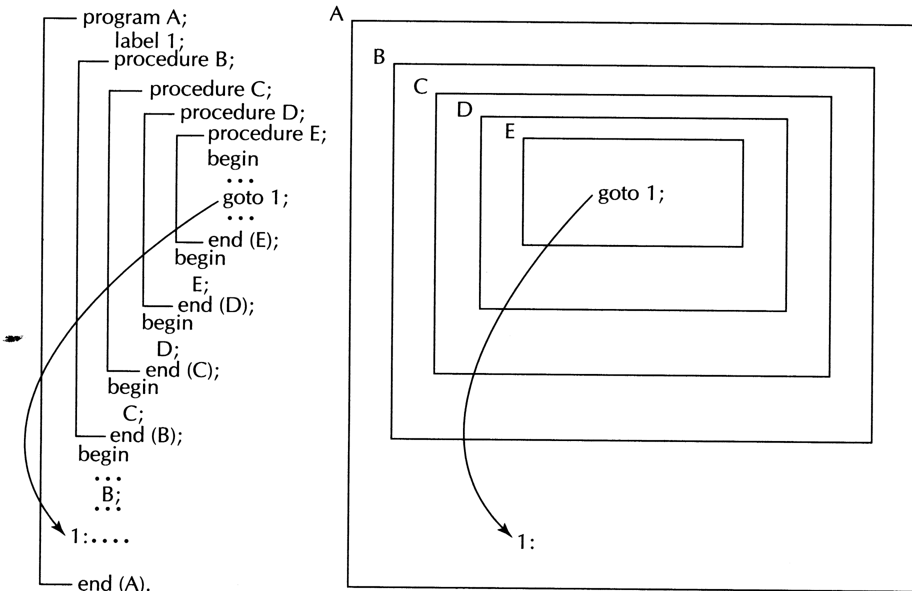


Figure 6.7 Example of a Nonlocal **goto**

registers. How is this accomplished? Getting to the environment of definition of the label is just like getting to the environment of definition of a procedure: The symbol table entry for the label contains the static nesting level of its definition so the static distance to the environment can be computed at compile-time as the difference of the static nesting levels of the use and definition of the label. Thus, `EP` can be set (at run-time) to the context of the label by *sd* traversals of the static chain, where *sd* is the static distance between the **goto** and the definition of the label:

```
sd × EP := M[EP];           scan down static chain
     where sd = snl(goto) - snl(label)
```

and `snl(x)` is the static nesting level of *x*.

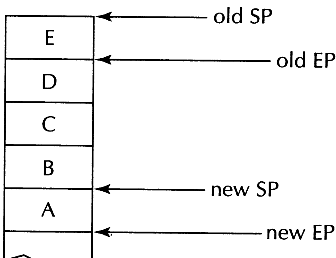


Figure 6.8 Stack and Registers before and after Nonlocal **goto**

The SP register points to the next available stack location above the current activation record, so it can be restored by

```
SP := SP + size(AR of the label);
```

The size of the activation record is known to the compiler or can be computed from information in the activation record.

The last step is to transfer to the address corresponding to the label, which is a constant available to the compiler. The steps for a nonlocal **goto** can be summarized as follows:

```
sd × EP := M[EP];
     SP := EP + size(AR of the label);
     goto address(label);
```

We can see that the number of memory references to do a nonlocal **goto** is *sd*, which is the static distance from the **goto** to the label.

- ***Exercise 6-9:*** What is the static distance between '**goto** 1' and the definition of label '1' in the program in Figure 6.7?

- ***Exercise 6-10:*** Some languages (e.g., Algol) allow labels to be passed as parameters to procedures; this is analogous to passing a procedure or function as a parameter. Describe in detail, including code sequences, the implementation of a label actual and a **goto** to a formal label.

Summary of Static Chain Implementation

The memory references required for *static chain* implementation for variables and procedures are summarized in Table 6.1. Do not take the actual numbers in this table too seriously; they may vary from machine to machine depending on the number of available registers, the instructions provided, and the exact format of the activation records. They are indicative of the costs, however, and will enable us to compare the static chain method to other implementation methods.

TABLE 6.1 Cost of Static Chain Implementation

Operation	Memory References
variable access	*sd* + 1
procedure call	*sd* + 3
procedure return	2
pass procedural actual	*sd* + 2
formal procedure call	5
goto	*sd*

6.3 DISPLAY METHOD

Displays Allow Random Access to Contexts

We saw in the previous section that accessing a variable requires *sd* + 1 memory references, where *sd* is the static distance from the use of the variable to its declaration. This is fine for local variables but can become quite expensive for global variables in a deeply nested program. The problem results from the sequential organization of the static chain; every access to a nonlocal variable requires scanning down the static chain until its environment of definition is found. Performance could be improved if there were some way of getting *directly* to the environment of definition. This kind of direct access can be achieved by having an array (a *random access* data structure) that contains pointers to all of the accessible contexts. That is, if `D` is the array, then `D[i]` is a pointer to the activation record for the environment at static nesting level *i*. Such an array is called a *display.* Figure 6.9 shows a stack and its display. Notice that no `EP` register is required since the compiler knows the level at which each statement will execute. Therefore, if a statement is to execute at level *i*, it can find its activation record through `D[i]`.

How does a display improve variable accesses? Recall that there are two steps in accessing a variable: (1) locating the activation record containing the variable and (2) locating the variable within its activation record. In the static chain implementation, the first step required skipping down the static chain. With the display the activation record is immediately accessible by `D[snl]`, where *snl* is the static nesting level of the variable's declaration (recall that this is a constant computed by the compiler and stored in the symbol table entry for the variable). The second step is accomplished by adding the variable's *offset* to the base address given in the first step. Accessing a variable with coordinates (*snl, offset*) is accomplished by

```
fetch M[D[snl] + offset]
```

This requires two memory references: one to get the display entry and one to get the variable itself. In some implementations the display is stored in high-speed registers, which means that only one memory reference is required. In either case, the time required by the display

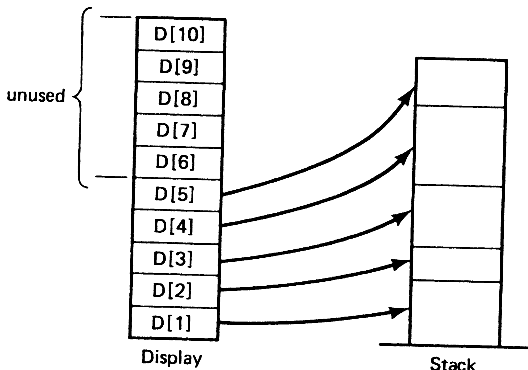


Figure 6.9 Example of a Display

method compares favorably with the $sd + 1$ memory references required by the static chain implementation.

■ ***Exercise 6-11:*** In the static chain implementation, access to a *local* variable requires one memory reference (because $sd = 0$). In the display implementation, access to any variable requires two memory references. Suggest a modification to the display implementation that accesses local variables in one memory reference.

Calls Require Saving the Display

Consider a call from static nesting level u to a procedure defined at static nesting level d. If the procedure's name is visible, then it must have been declared at the same or a lower static nesting level than that from which it is being called. In other words, $u \geq d$. The stack before and after the call is shown in Figure 6.10. Notice that if the call is from level u, then all of the display entries from `D[1]` to `D[u]` must be in use (i.e., contain pointers to activation records). Further, notice that if the environment of definition of the procedure is at level d, then the procedure itself is at level $d + 1$, and the pointer to its own activation record must go in `D[d + 1]`. This will destroy the previous contents of `D[d + 1]`, which was in use if $u > d$ (which is often the case). Therefore, the previous contents of `D[d + 1]` must be saved. (It is not necessary to save the contents of `D[d + 2]` through `D[u]` at this time; they will be saved if and when a call to the corresponding level takes place.)

To accomplish this we will set aside a field called `EP` in a procedure activation record to hold the saved element of the display. The parts of an activation record for the display method are

- `PAR` parameters
- `IP` resumption address
- `EP` saved display element
- `DL` dynamic link

There is no `SL` (static link) field because the display has taken over its function.

We can get the code sequence for a procedure call with a display by taking the static

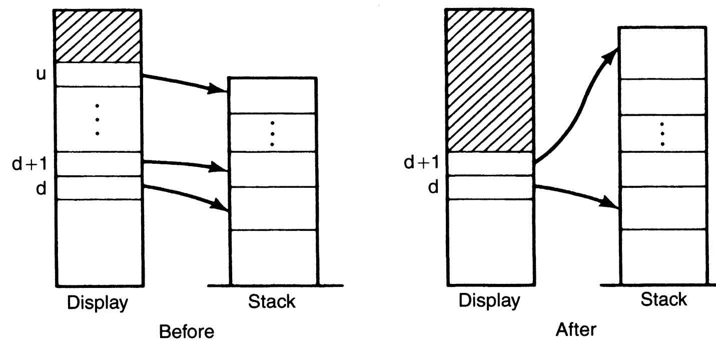


Figure 6.10 Example of Procedure Call with Display

```
        M[SP].PAR[1]   := eval. par. 1;
                          .
                          .                    transmit parameters
                          .
        M[SP].PAR[n]   := eval. par. n;
        AP := D[u];                            get caller's AR
        M[AP].IP := resume;                    save resume location
        M[AP].EP := D[d + 1];                  save display element
        M[SP].DL := AP;                        set dynamic link
        D[d + 1] := SP;                        install new AR
        SP := SP + size(callee's AR);          allocate callee's AR
        goto entry(callee);                    enter the callee
resume:
```

Figure 6.11 Procedure Call Sequence for Displays

chain version (Figure 6.3) and replacing the operation to set the static link with an instruction to save the display entry for the level of the callee. The resulting sequence is shown in Figure 6.11. Notice that the number of memory references required (omitting parameter initialization) is 6. This is often cheaper than the $sd + 3$ references required in the static chain implementation.

Returning from a procedure just reverses the operations of a procedure call. The `IP` and the display element must be restored from the caller's activation record, and the callee's activation record must be deleted. The code sequence is shown in Figure 6.12. Notice that restoring the display drives up the cost: It requires five memory references to return with a display but only two with the static chain.

- ***Exercise 6-12:*** Draw the state of the stack, display, and registers after each of the instructions in Figures 6.11 and 6.12.

- ***Exercise 6-13:*** Describe the implementation of procedural parameters with the display method. As with the static chain method, it is necessary to make sure that the callee executes in the correct environment. Develop code sequences for passing procedures as actual parameters and for calling formal procedures (analogous to those in Section 6.2). Compute the memory references required for each of these operations.

- ***Exercise 6-14**:*** (Difficult) Discuss the implementation of nonlocal **goto**s with the display method. How will you restore the display so that it correctly reflects the context of the label? Develop the code sequences and compute the number of memory references for a nonlocal **goto**. What conclusions can you draw?

```
SP := SP - size(callee's AR);          deallocate callee's AR
AP := M[D[d + 1]].DL;                  get caller's AR
D[d + 1] := M[AP].EP;                  restore display element
goto M[AP].IP;                         resume execution of caller
```

Figure 6.12 Procedure Return with Display

Comparison of Static Chain and Display

Table 6.2 compares the number of references required by various operations using the static chain and display implementations. What conclusions can we draw? Notice that the display implementation accesses nonlocal variables much more efficiently than the static chain implementation. This was the motivation for the display method. On the other hand, the display implementation of procedures is a little *less* efficient than the static chain implementation. Whether the static chain or display implementation is better depends on the relative frequency (at run-time) of procedure calls and variables accesses and on the average static distance to each of these. Since most programs do many more variable accesses than procedure invocations, the display is probably preferable.

Shallow Binding Method

The static chain and display methods are not the only possible implementation techniques for block-structured languages. In one other method, sometimes called *shallow binding*, each procedure is *statically* allocated one copy of its activation record (cf. FORTRAN activation records, Section 2.3). This static copy always holds the information and local variables for the most recent activation of that procedure. Since these activation records are stored at fixed locations in memory, variable access is always efficient. How is recursion implemented? Whenever a procedure is invoked, the contents of the static activation record area must be pushed onto a stack so that the activation record area can be used by the new activation. When the callee returns, the contents of the activation record will be restored from the stack. Hence, the major cost in the shallow binding method is the cost of saving and restoring these activation records on procedure call and return. Also, since the activation records are allocated statically, their size is fixed so shallow binding cannot be used for Algol and other languages with dynamic arrays.[3]

■ ***Exercise 6-15:*** Define and analyze the shallow binding method. Describe the code sequences for variable access, procedure call and return, and nonlocal **gotos**. Analyze the number of memory references required for all of these operations and compare them with the static chain and display methods. What conclusions do you draw?

TABLE 6.2 Comparison of Static Chain and Display

Operation	Static Chain	Display
local variable	1	2
nonlocal variable	$sd + 1$	2
procedure call	$sd + 3$	6
procedure return	2	5

[3] This is true so long as the arrays are allocated space in the activation record. If they are allocated space in a separate *heap*, then the activation records may be constant size.

■ ***Exercise 6-16*:*** Some computers have a number of high-speed registers capable of holding display elements. Usually there are not enough of these registers to hold the entire display. Discuss a strategy for making efficient use of the display registers. Show code sequences and estimate the number of memory references required for the various operations (call, return, etc.).

6.4 BLOCKS

Blocks Are Degenerate Procedures

In Pascal, since the procedure is the only scope-defining construct, it is the only construct that adds or deletes activation records to or from the stack. Other languages, including Algol and Ada (which is discussed in Chapters 7 and 8), have another scope-defining construct—the *block.*

Since activation records represent contexts and blocks define contexts, blocks will also have to have activation records. These activation records will have to be created when the block is *activated* (i.e., entered) and destroyed when the block is *deactivated* (i.e., exited). This automatic allocation and deallocation of activation records provide the automatic dynamic storage allocation discussed in Chapter 3 (Section 3.3).

How is block entry-exit implemented? We can solve the problem of implementation of blocks by reducing it to another problem that we have already solved—the implementation of procedures. There is a clear similarity: When a procedure is entered, an activation record is created, just as for a block; when a procedure is exited, its activation record is destroyed, just as for a block. Thus, we can think of entering and exiting a block as calling and returning from a procedure.

To see how this can be, consider the *Algol* program in Figure 6.13; it can be translated into the *Pascal* program in Figure 6.14. (We have translated an Algol program into a Pascal program because Algol has blocks but Pascal does not and Pascal procedures have local storage but Algol's don't.) Notice that each block has been turned into a procedure that is invoked in exactly one place—where the corresponding block was nested. A perfectly correct implementation of blocks would be to translate them into procedures in exactly this way. We will see next that this is a little inefficient and that a number of improvements can be made by considering the particular characteristics of blocks.

Block Entry-Exit Is a Degenerate Call-Return

We derive the steps for block entry by considering the steps for procedure call (Figure 6.3) one by one. The crucial differences result from the fact that a block corresponds to a procedure that is called from exactly one place in the program. For example, we do not have to perform the first step, saving the resume location, since for a block it is always the same—the instruction immediately following the **end**. Similarly, we do not have to evaluate the

Figure 6.13 Algol Program with Blocks

```
begin
  integer N;
  N := 0;
  begin
    real sum, avg;
    sum := 0.0;
    .
    .
    .
  end;
  N := 1;
  begin
    real val;
    val := 3.14159;
    .
    .
    .
  end;
  N := 2;
end.
```

parameters because a block is equivalent to a parameterless procedure. Hence, the first step that is relevant to block entry is setting the dynamic link[4]:

```
M[SP].DL  =  EP;
```

The next step in a procedure call is to find the environment of execution for the procedure and to use this to initialize the static link. The environment of execution for Algol is always the environment of definition, and the environment of definition for a block is always the immediately surrounding block. That is, the static link of a block always points to the immediately surrounding block, which is contained in the `EP` register. Therefore, the static link of the new block is set by

```
M[SP].SL  :=  EP;
```

Notice that the static and dynamic links are the same; more about this later.

The next steps in a procedure call install the new activation record and allocate its space on the stack. This is also required for a block:

```
EP  :=  SP;
SP  :=  SP  +  size(AR);
```

The final step of a call, jumping to the beginning of the procedure, is not required for a block since the first instruction of the block immediately follows the block entry code (i.e., the code for **begin**).

Block exit is patterned on procedure return (Figure 6.4). In this case, we omit jumping

[4] We begin with the static chain method since it is simpler than the display.

```
procedure B1;
   var N: integer;
   procedure B2;
      var sum, avg: real;
   begin
      sum := 0.0;
      .
      .
      .
   end {B2};
   procedure B3;
      var val: real;
   begin
      val := 3.14159;
      .
      .
      .
   end {B3};
begin
   N := 0;
   B2;
   N := 1;
   B3;
   N := 2;
end.
```

Figure 6.14 Pascal Program with Procedures Instead of Blocks

back to the resume location since this always immediately follows the **end** (i.e., is the next instruction following the exit code).

How can we improve these code sequences? Since the static and dynamic links are always the same for a block, there is no reason to have them both in a block activation record. Therefore, we will eliminate the dynamic link. The result is that a block activation record has a very simple structure:

- `LV` local variables
- `IP` resumption address
- `SL` static link

A resumption address (`IP`) is required because a block may call a procedure; hence we need a place to save its state of execution.

The **begin-end** sequence, the code for entering and exiting blocks, is summarized in Figure 6.15. Note that only two memory references are required for entry and exit combined.

Why is it that blocks require only one link but procedures require two? The *static link* leads to the *statically* prior context, and the *dynamic link* leads to the *dynamically* prior context. For a block, the statically and dynamically prior contexts are always the same; hence,

```
M[SP].SL := EP;                  set static link
EP := SP;                        install inner AR
SP := SP + size(block AR);       allocate inner AR
                    .
                    .            body of innerblock
                    .
SP := SP - size(block AR);       deallocate inner AR
EP := M[EP].SL;                  reinstall outer block
```

Figure 6.15 Block Entry-Exit with Static Chain

only one link is required. To put it another way, a procedure may be activated in a context other than that in which it is defined; hence, we distinguish between the *environment of definition* and the *environment of call.* This *remote activation* potential of the procedure is made possible by the fact that a procedure has a name and hence may be activated anywhere that name is visible.[5] In contrast, a block has no name; it is *anonymous.* The result is that a block can be activated in only one context—the context in which it is textually nested. Therefore, for a block, the environment of definition and the environment of activation are the same—the immediately surrounding block (or procedure). Thus, to be perfectly precise, the link in a block's activation record should be called neither the static link nor the dynamic link for it is in fact both.

- ***Exercise 6-17:*** Draw the state of the stack and registers after each of the steps in Figure 6.15.

- ***Exercise 6-18*:*** Design an activation-record format for some computer with which you are familiar. Write out the instruction sequences for block entry-exit, procedure call-return, and variable access. Design the activation-record format to optimize these operations. Describe the different choices and trade-offs you have to make.

- ***Exercise 6-19**:*** Investigate the call-return sequences used by some computer and language for which you can obtain the appropriate documentation. Relate the instructions and activation-record formats to the models discussed in this chapter. How are the basic functions (save state of caller, etc.) accomplished?

Blocks Require Display Updating

Block entry-exit is also quite simple in the display implementation. First, let us consider block entry. Figure 6.16 shows the display and stack before and after entry to a block at level *snl.* We can see that space for the block's activation record has been allocated on the top of the stack and that the display entry `D[`*snl*`]` has been set to point to this activation record. The

[5] As we have seen, many languages, including Algol and Pascal, allow a procedure to be passed as a parameter to another procedure, which means that a procedure can even be activated from environments in which its name is not visible.

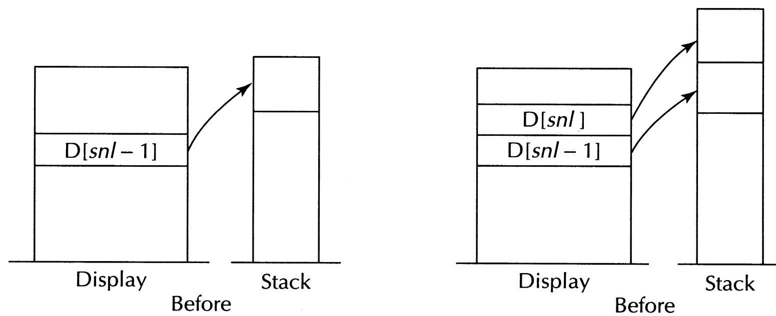


Figure 6.16 Block Entry with Display Method

previous contents of D[*snl*] must be saved in the block's activation record since it may contain a valid activation record pointer. Therefore, the code for block entry is

```
M[SP].EP := D[snl];        save display element
D[snl] := SP;              add inner AR to display
SP := SP + size(locals);   allocate space for inner AR
```

This requires three memory references (for display updating).

Block exit is even simpler: All that is required is to deallocate the block's activation record and restore the display:

```
SD := SP - size(locals);   deallocate inner AR
D[snl] := M[SP].EP;        restore display element
```

This requires two memory references for restoring the display.

■ ***Exercise 6-20*:*** Explain in detail why it is necessary to save and restore D[*snl*] during block entry-exit. Write in skeleton form a program that would not work correctly if this were not done.

6.5 SUMMARY

In this chapter we have seen one of the most important concepts in the implementation of programming languages—the idea of an activation record. It can be formally defined:

> **The Activation Record**
>
> An activation record is an object holding all of the information relevant to one activation of an executable unit.

What is an "executable unit"? Most frequently it is a procedure, function, or program, although in later chapters we will see other examples, such as coroutines and tasks. Generally,

it is a part of a program that includes some code and a name context. Executable units are capable of communicating with other executable units.

We have seen that in languages that provide recursive procedures there can be several *instances* of a procedure in existence at one time. One of these may be active and the others will all be suspended. Each of these instances, or *activations*, has a separate, private naming context (the local and nonlocal variables), although all of the instances of one procedure share the same code, because the variables can be changed but the code cannot. The state of an activation can be completely specified by giving its naming context (the *ep*) and by specifying the place in the code where it is executing (the *ip*).

The state of execution of a computer is also specified by an *ep-ip* pair; this pair is called the *locus of control.* The locus of control specifies the current instruction being executed (the `IP` register) and the context of its execution (the `EP` register).

When a procedure is called, a new activation is created, which in turn requires the creation of a new activation record to hold the local context. In many languages (including Algol, Pascal, and Ada), when an activation returns (or exits by a nonlocal **goto**), its activation record is destroyed. This deallocation is possible since these languages do not permit reactivating a deactivated procedure or accessing the local context of a deactivated procedure. There are some languages that do allow these things, and in these the activation records must be preserved after the procedure returns. (This is called a *retention* strategy as opposed to a *deletion* strategy; see Berry, 1971.)

In the languages we have studied, at most one activation can be active at a time; the others are all suspended, awaiting reactivation. Some languages permit more than one activation to be active at a time, which is called *parallel* or *concurrent* programming. In these languages there may be several *loci of control,* one for each (real or virtual) processor. Ada (discussed in Chapters 7 and 8) is an example of such a language. Activations that can execute concurrently with each other are usually called *processes* or *tasks.*

Finally, we have seen that the use of activation records generally leads to two-coordinate addressing methods. For example, a variable is addressed by a pair comprising an environment pointer and a relative offset; the environment pointer provides an access path to the activation record containing the variable, and the relative offset locates the variable within that activation record. Similarly, when a procedure is passed as a parameter, it is a closure, an *ep-ip* pair, that is passed; the *ep* specifies the environment of definition of the procedure (an activation record), and the *ip* specifies the code to be executed. Since a thunk (Chapter 3) is essentially a procedural parameter, *name* parameters are also implemented by passing closures. Similarly, label parameters are implemented by *ep-ip* pairs; the *ep* specifies the context of the label, and the *ip* specifies the code to be executed.

In succeeding chapters we will see other language constructs that are implemented with activations records and two-coordinate addressing. Thus, these are important ideas that should be understood thoroughly.

EXERCISES*

1. Describe in detail the implementation of Algol's dynamic arrays. Estimate the number of memory references required to subscript a two-dimensional array and compare with the number of memory references required for a Pascal or FORTRAN array.

2. Describe in detail the implementation of flexible arrays, that is, arrays whose size can change while the array is in existence.

3**. Design a run-time organization that simplifies nonlocal **gotos**.

4**. Pick an existing computer and critique its support for block-structured languages. Suggest changes that would improve its support.

5. Estimate the number of memory references required for procedure call and return in FORTRAN. Compare this with the references required for Pascal procedures and discuss the costs and benefits of the two mechanisms.

6. Estimate the cost of variable accessing if variable names were looked up at run-time rather than being converted to static-distance/offset pairs at compile-time.

7. Describe a run-time organization for *dynamically* scoped block-structured languages. How can you avoid character string comparisons every time a variable is accessed?

8. It is often useful to allow procedures in block-structured languages to call FORTRAN subprograms and vice versa. Describe in detail the implementation of this capability. Discuss run-time organization, information needed by the compilers, required language extensions, and so forth.

9. Produce detailed code sequences for some existing computer for variable accessing and procedure call and return using the static chain method.

10. Do the same for the display method.

11. Do the same for the shallow binding method.

12*. The C language avoids the problems of environment access by disallowing nested environments (although recursive procedures are permitted). Discuss the pros and cons of this solution.