subprogram may have been activated recursively a number of times and that the subprogram being executed may have been initially called from one subprogram and then later resumed from another. Thus it is not immediately obvious where control is to be transferred when a RETURN or RESUME is encountered.

b. Design a simulation for this control structure on a conventional computer. Specify the storage requirements for return and/or resume points and the manner in which this information is used during execution of the CALL, RESUME, and RETURN statements.

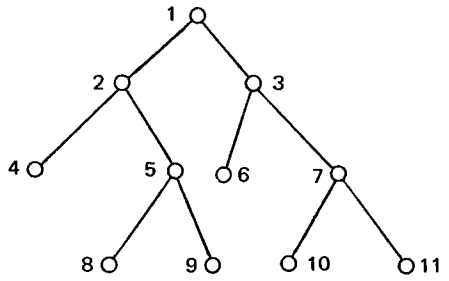
5-13. One of the arguments favoring retention of statement labels and goto statements is that other control structures may often be simulated using these primitive features. Simulation of simple coroutines in FORTRAN is an example. Design a simulation for coroutines in FORTRAN using the CALL statement, the ASSIGN statement (e.g., ASSIGN 20 to L;—where 20 is a statement number), and the *assigned* goto GO TO L, (list of statement numbers). The simulation should operate so that the CALL statement actually acts like a RESUME statement for subprograms in which the coroutine simulation is being used.

5-14. An alternative view of a data structure generator is as a coroutine-generator. A coroutine-generator is a coroutine which on initial activation is given a data structure as input parameter and returns the first element of the sequence of elements. On each subsequent resume call it generates the next element of the sequence and returns it. Design such a generator for

a. Generating the main diagonal on a FORTRAN, ALGOL, or PL/I matrix.

b. Generating the integer-valued leaf nodes on a LISP tree.

c. Generating the nodes breadth-first on a LISP tree. For example, the nodes in numbered order from the following tree:




# 6 DATA CONTROL

When writing a program one ordinarily is well aware of the operations that the program must execute and their sequence, but seldom is the same true of the operands for those operations. For example, an ALGOL program contains

$$X := Y + Z \times 2$$

Simple inspection indicates three operations in sequence: a multiplication, an addition, and an assignment. But what of the operands for these operations? One operand of the multiplication is clearly the number 2, but the other operands are marked only by the identifiers $Y$, $Z$, and $X$, and these obviously are not the operands but only designate the operands in some manner. $Y$ might designate a real number, an integer, or the name of a parameterless subprogram to be executed to compute the operand. Or perhaps the programmer has erred and $Y$ designates a Boolean value or a string or serves as a statement label. $Y$ may designate a value computed nearby, perhaps in the preceding statement, but equally as likely it may designate a value computed at some point much earlier in the computation, separated by many levels of subprogram call from the assignment where it is used. To make matters worse, $Y$ may be used as a name in different ways in different sections of the program. Which use of $Y$ is current here? In a nutshell, the central problem of data control is the problem of what $Y$ means in each execution of such an assignment statement. Because $Y$ may be a nonlocal variable, the problem involves scope rules for declarations; because $Y$ may be a formal parameter, the problem involves techniques for parameter transmis-

sion; and because $Y$ may name a parameterless subprogram, the problem involves mechanisms for returning results from subprograms. Each of these topics is taken up in turn in the following sections.

## 6-1. BASIC DATA CONTROL CONCEPTS

There is a striking contrast between the basic indirectness and potential ambiguity of data control and the explicitness of most sequence control. One must know the operations and their sequence or else one can hardly write a program, and so ordinarily the operations and sequence control structures are set into the program when it is written. Yet if at each step one also knew the operands for the operations, then writing the program would be pointless. One writes a program in order to apply the same basic sequences of operations to a variety of different data sets at different times. Thus references to data in a program must necessarily be indirect for the most part; when writing the program we do not know what the data will be. It is in this necessary indirectness that the potential for ambiguity in data references arises.

In conventional computer hardware the solution to the data control problem is particularly simple. The computer is designed with a fixed set of data storage locations (memory words and working registers), each of which may contain a single data item. Each location is designated by a unique name (memory address or register designator). Machine language operations must specify exactly the name of the location in which the operand is to be found or the result stored. Thus whenever a particular instruction is executed the operands for the operation are to be found in the same storage locations. This hardware organization is an example of a data control organization fixed at the time of definition of the machine language. The machine language programmer has no control over the association of names and storage locations; his programs must use the associations built into the hardware.

It is apparent that this hardware data control structure is entirely too inflexible. Some hardware features modify this structure partially, for example, by allowing the operand of an operation to be specified by a base address plus the contents of an index register. However, one must move to an assembly language to obtain a significantly more flexible data control structure. The assembly language programmer may choose his own names for data locations and need not specify which location is to be associated with which name, leaving this decision up to the assembler and loader.



Assembly language data control is still too restrictive, however. High-level languages, among other advantages, typically provide much more flexible data control structures, allowing multiple uses of the same name as well as multiple names for the same data location. The mismatch between conventional hardware and high-level programming languages is particularly apparent here. Software simulation of data control structures is the rule in implementations of high-level languages.

### Names and Identifiers

The central concern in data control is the question of the meaning of names. Taken very generally the term *name* may designate any expression in a program used to specify an operand for an operation. Used in this sense data control overlaps some of the questions of previous chapters. For example, an operand for an addition operation may be designated by giving an expression to be evaluated to compute the operand, as in $(U/V-A*B)+\ldots$ or $SQRT(2*Z)+\ldots$. In part, the meaning of such an expression name is tied to questions of sequence control (see Section 5-2). What is yet to be discussed is the meaning of references to simple identifiers, such as $U$, $V$, $Z$, and $SQRT$. Another aspect of *names* which has been partially discussed previously is that of subscripted variables, names for elements of data structures, e.g., $A[2,3]$, $B[I]$, or $EMPLOYEE.SALARY$. In Chapter 3 the distinction between referencing and accessing in subscripted variables was made. *Referencing* is the operation that determines which structure is associated with a particular structure name (e.g., $A$, $B$, or $EMPLOYEE$). *Accessing* is the operation which computes the designated element location in the structure given the subscripts and the access point to the entire structure found by a referencing operation. Accessing is discussed at length in Chapter 3. Referencing is a central concern in this chapter.

Restricting our concern with names to cases that have not arisen in other contexts leaves us for the most part with questions of "simple" noncomposite names. Simple names include

1. Names for simple variables.
2. Names for data structures, including input-output files.
3. Literals, i.e., names for data constants.
4. Subprogram names.
5. Primitive operation symbols, e.g., +, *.

6. Statement labels.

7. Formal parameters.

Two of these cases are relatively trivial. The meaning of literals and primitive operation symbols is ordinarily fixed in the language definition. Thus the programmer has no control over whether 10 represents ten (decimal), eight (octal), or two (binary) or whether + represents addition. Some exceptions exist, as, for example, in SNOBOL4's OPSYN feature, which allows programmer redefinition of primitive operation symbols. However, these cases are relatively rare and add few new concepts. The remaining cases of simple variables, data structure names, subprogram names, statement labels, and formal parameters provide our central focus. Rules for the syntax of such names differ between languages and between classes within the same language. For simplicity the term *identifier* is used for a simple name of any of these types in the discussion here.

#### Identifier Associations and Data Control Primitives

Data control is concerned in large part with *associations* (sometimes called *bindings*) between identifiers and program or data elements. Each association may be represented as a pair consisting of the identifier and its associated element or a pointer to the element. For our purposes here this simple view is sufficient. In language implementations a variety of representations for identifier associations at run time may be used.

During execution of a program the associations for a particular identifier, say $X$, typically follow the pattern:

1. *Create first association*. An initial association for $X$ is created by a declaration at the start of execution of the main program, or on entry to a subprogram.

2. *Reference association*. A reference to $X$ in the program invokes a *referencing operation* which uses the association created for $X$ to retrieve the associated program or data element (or a pointer to the element).

3. *Deactivate association*. Control is passed to another part of the program, causing the association for $X$ to be deactivated. While inactive the association for $X$ continues to exist but cannot be referenced. Deactivation is often caused by subprogram calls.

4. *Reactivate association*. The association for $X$ is reactivated as a result of control returning to an appropriate program segment. The



association may again be referenced. Deactivation and reactivation may occur many times.

5. *Destroy association*. The association for $X$ is destroyed and may no longer be referenced.

6. *Repeat steps 1-5*. Another association for $X$ may be created, referenced, etc.

It is simplest to assume that a given identifier has at most one active association at any point during program execution, so that a reference to $X$ has a unique meaning. This assumption is somewhat simplistic; various languages allow multiple identifier associations to be active simultaneously, using context to determine which association is appropriate for any given reference. For example, in SNOBOL4 the statement

$$X = X(2) :(X)$$

references $X$ as a label, simple variable, and subprogram name in the same statement. Context, however, determines which association is appropriate for each reference to $X$. Although such situations increase the complexity of the referencing operation, no new concepts are introduced. Thus only the case in which each identifier has at most one active association at any point need be considered in detail.

To associate an identifier with a data or program element requires an operation, termed a *naming* operation. Naming necessarily precedes referencing, for the naming operation sets up an association for an identifier, while the referencing operation uses the association to retrieve the object associated with a particular identifier. The operation of destroying an association for an identifier may be termed *unnaming*. Note that naming is distinct from *creating*, an operation discussed in Chapter 4. The two often occur together and thus are sometimes confused. The ALGOL declaration real $X$, for example, specifies both that a simple variable is to be created and that it is to be named $X$ (i.e., the identifier $X$ is to be associated with it). Similarly the declaration array A[$M$:$N$] creates an array and names it $A$. Creating may be separate from naming, however, as in the SNOBOL4 function call

```
ARRAY(10)
```

which creates an array and returns a pointer to it. The created array is not named by the creation function ARRAY. Naming without

creating is exemplified in argument transmission where the formal parameter identifier is associated with the corresponding actual parameter data object.

As with creating and naming, the operations of data structure destruction and unnaming may also be separated. Recall from Chapter 4 the problems of dangling references and garbage associated with the destruction of data structures. Dangling references arise when a data structure is destroyed before all the access paths to that structure have been destroyed. A primary source of access paths to data structures is through identifier associations. If a data structure is destroyed while still named, i.e., while still associated with an identifier, then a later attempt to reference the structure through that name will leave the referencing operation "dangling"—thus the term *dangling reference*. However, dangling references may also be created when pointers to a destroyed structure exist in other structures, and thus the problem is more general than simply one of data control.

Similarly garbage is often, although not exclusively, a question of data structure unnaming. Garbage arises when all access paths to a data structure (or part of a structure) are lost without the storage for the structure being recovered. The unnaming operation destroys access paths through identifier associations and thus may lead to garbage creation. However, garbage may be created in other ways as well, for example, by destroying pointers stored in other data structures. The general problem of the prevention and recovery of garbage is taken up in the next chapter in the context of storage management.

The operations of activation and deactivation of identifier associations are often confused with the operations of naming and unnaming. For an association to be used in referencing it must both *exist* (as a result of a naming operation) and be *active*. Inactive associations are those that exist but that temporarily cannot be referenced. The ALGOL block structure provides a typical example of the distinction. On block entry in ALGOL each declaration serves as a naming operation, creating a new association between an identifier and a simple variable, array, subprogram, or switch. Simultaneously any previously existing associations for the declared identifiers are *deactivated*. Referencing within the block uses the newly created active association for each identifier. Block exit causes unnaming for the identifier associations created on entry and *reactivation* of the associations deactivated on entry.

Five major data control operations concerned with identifier associations have now been discussed:



1. *Naming*. The operation of creating an association between an identifier and a program or data object.

2. *Unnaming*. The operation of destroying an association between an identifier and its associated object.

3. *Activating*. The operation of making active an existing association between an identifier and a program or data object and thus making the association available for use in referencing.

4. *Deactivating*. The operation of making inactive an existing association between identifier and program or data object.

5. *Referencing*. The operation of retrieving the data or program object currently associated with a given identifier using the unique currently active association for the identifier.

## 6-2. REFERENCING ENVIRONMENTS AND SCOPE RULES

A useful concept in the study of data control is that of *referencing environment*. At any point during execution of a program a certain set of identifier associations is active. The set of active associations is termed the *referencing environment* of that program point. Whenever a reference to an identifier occurs it is the referencing environment which determines the appropriate association for that reference. The operations of naming, unnaming, activating, and deactivating modify referencing environments. Our concern is with the pattern of referencing environments that occur during execution of programs in various languages.

A *scope rule* is a rule for determining referencing environments in a program. A scope rule ordinarily specifies the pattern of activation, deactivation, and unnaming associated with a particular naming operation, thus defining the points during program execution when a particular identifier association is active (i.e., is part of the referencing environment).

### Static and Dynamic Scope

Scope rules are usually classified as dynamic or static. A *dynamic scope rule* defines scope in terms of program execution. A typical dynamic scope rule specifies that the referencing environment at any point during program execution is determined by the most recent

A *static scope rule* defines scope in terms of the structure of the program when written, at translation time. For example, in ALGOL the referencing environment at any point during execution of a program is determined not by the pattern of naming during execution but by the pattern of declarations in enclosing blocks in the original program. In the following sections various static and dynamic scope rules are studied in depth. In general, dynamic scope rules are relatively easy to implement, but static scope rules, although more difficult to implement, allow production of considerably more efficient executable code. Static scope rules tend to be characteristic of languages such as FORTRAN, ALGOL, and PL/I in which execution speed is important, while dynamic scope rules are more common in software-interpreted languages such as SNOBOL4, LISP, and APL.

#### Global, Local, and Nonlocal References

References to identifiers are classified as *local references* if they use an association active only within the block or subprogram currently being executed. For example, in ALGOL a local reference is a reference to an identifier declared in the most recently entered block. A *global reference* is a reference to an association active throughout program execution. References to subprogram names in FORTRAN are examples of global references, as are references in ALGOL to identifiers declared in the outermost block. A *nonlocal reference*, as the name indicates, is any reference which is not local. In FORTRAN references are only local or global. In ALGOL nonlocal references may include references to identifiers declared in intermediate-level blocks between the outermost block and the innermost.

It is convenient also to use the term *local referencing environment* (or simply *local environment*) to designate those local associations introduced at the last change in referencing environment. The *nonlocal referencing environment* is the remainder of the referencing environment at any point. Typically the local environment of a subprogram contains formal parameters and local variables declared in the head of the subprogram. The nonlocal environment contains identifier associations that are shared with other subprograms.

The concepts necessary for an extended look at data control in programming languages are now at hand. Data control structures are almost always tied to subprograms. For this reason most of the discussion of various approaches to data control is organized in terms of the various subprogram control structures discussed in Chapter 5—simple nonrecursive subprograms, recursive subprograms, coroutines, interrupt routines, etc.

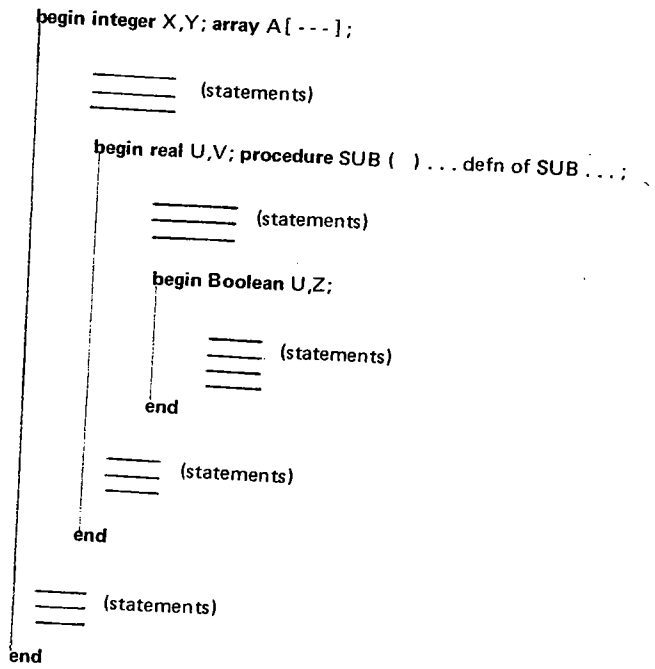


## 6-3. BLOCK STRUCTURE

The concept of block structure as found in *block-structured languages* such as ALGOL and PL/I deserves special mention. In a block-structured language each program or subprogram is organized as a set of nested blocks, usually delimited as in ALGOL by the symbols **begin** and **end**. Figure 6-1 shows schematically the structure of a typical ALGOL program. Each block begins with a set of declarations which serve two purposes:

1. *Naming.* Each declaration sets up associations for one or more identifiers. These form the local referencing environment for the block.

2. *Data structure creation.* Some of the declarations may also define data structures or simple variables to be created on block entry.



When properly handled, block structure allows use of a particularly straightforward stack-based storage management scheme (described in the next chapter), as well as providing a rather flexible data control structure. In addition block structure lends itself to use of static scope rules, thus allowing production of more efficient executable code. Because of these advantages, block structure has been adopted in a variety of languages.

Block structure may be considered as providing a special data control structure, operating within subprograms, which allows changes of referencing environment at arbitrary points during subprogram execution rather than only on entry and exit. However, a better view is to treat a block as a simple parameterless subprogram that has actually been coded in line at the point of call; i.e., the subprogram call statement has been replaced by the body of the subprogram. This view is preferable because the same problems arise in both blocks and simple parameterless subprograms and thus the cases may profitably be considered together.

## 6-4. SIMPLE PARAMETERLESS SUBPROGRAMS: LOCAL ENVIRONMENTS

Let us begin with simple data control structures and progress to the more complex cases. Consider first the usual subprogram hierarchy—a main program and set of subprograms which call each other in the ordinary manner. For the moment we shall ignore parameters and assume that recursive calls are not allowed. Blocks in block-structured languages as well as subprograms in FORTRAN and COBOL, ignoring parameters, fall into this category.

A subprogram has a local environment, consisting of those identifier associations that are activated on entry to the subprogram. The local environment ordinarily consists of the identifiers declared in the head of the subprogram (as well as formal parameters, but they are considered later). In addition a subprogram may have a nonlocal referencing environment, consisting of associations shared with other subprograms. Our first focus is on the local environment.

The different local environments of a set of subprograms may utilize many of the same identifiers but with different associations in each environment. As control is transferred back and forth between the subprograms the current referencing environment must be modified to reflect the appropriate local environment for referencing in each subprogram. Suppose that $P$, $Q$, and $R$ are three subprograms such that $P$ calls $Q$, which calls $R$, in a simple nested manner. Let us



focus on the local environment for $Q$, and ask the following questions:

1. What is the initial local environment set up on entry to $Q$ from $P$?
2. What happens to the local environment of $Q$ when $Q$ calls $R$?
3. What is the local environment of $Q$ when control returns from $R$?
4. What happens to the local environment of $Q$ when $Q$ returns control to $P$?

Questions 2 and 3 are perhaps the easiest to answer. Rather clearly the local environment of $Q$ should be *deactivated* (but not destroyed) when $R$ is called and *reactivated* on return from $R$. This choice corresponds with our intuitive notion that a subprogram call is only a temporary interruption of execution of the calling program.

Questions 1 and 4 lead to greater difficulty. In fact there is no generally accepted rule, but rather two distinct answers to these questions in general use:

*Approach 1. Deactivate the local environment on return to the calling program and reactivate it on reentry to the subprogram.* This solution is found in FORTRAN and COBOL. The local environment may be set up (i.e., local naming done) either during translation or on the first entry to $Q$. When $Q$ returns control to $P$ after its execution, the local environment is deactivated but retained. When $Q$ is called again, the same local environment is reactivated, and all variables have their old values, etc.

Retention of local environments between calls provides a sort of symmetry between subprograms. We may think of each subprogram as having its own attached local environment that exists throughout execution of the entire set of subprograms. For convenience think of these local environments as being set up initially by the translator. When execution begins the local environment of the main program is active, and the local environment of each subprogram is inactive. When the main program calls a subprogram such as $P$, then the local environment of $P$ becomes active and that of the main program inactive. As control continues to pass between subprograms the appropriate local environments are activated and deactivated in turn, but no local environment is ever destroyed.

*Approach 2. Destroy the local environment on return to the calling program and create a new one on reentry.* A quite different

solution is found in ALGOL (without own), LISP, SNOBOL4, and APL. When a subprogram is called, an entirely new local environment is created. When control is passed back to the calling program, the local environment is destroyed. Reentry to the subprogram causes creation of another local environment. If we recall the distinction between a subprogram definition and a subprogram activation (see Chapter 5), this approach amounts to setting up a new local environment for each activation of the subprogram. Approach 1 is based on a single local environment for each defined subprogram regardless of the number of activations of the subprogram during execution.

### Simulation of Local Environments

It is appropriate to think of a local environment as represented during execution by a table of local associations. Each entry in the table specifies an identifier and a pointer to the program or data element with which it is currently associated. Thus if identifier $A$ is associated with an array in the local environment of subprogram $Q$, then the table contains an entry for $A$ and a pointer to the array (or the array itself). For a simple variable named $X$ the table contains an entry for $X$ and a pointer to the location containing the value of $X$ (or the value directly). Identifiers which serve as subprogram names or statement labels present special difficulties; these cases are taken up in Section 6-9. Figure 6-2 shows a local environment table for a typical ALGOL block.

*Retained Local Environments.* Let us first consider the simulation appropriate for approach 1 to the handling of local environments. Each subprogram has a single local environment that exists

```
begin real X; integer Y;
      array A[ . . . ]; procedure SUB( . . . ) . . . ;
      ____
      ____  (statements)
      ____
end
```

Block declaration

| | |
|---|---|
| X | Value of variable |
| Y | Value of variable |
| A | Pointer to array |
| SUB | Pointer to sub-program code block |

Table

**Fig. 6-2.** Simple local environment table for an ALGOL block.



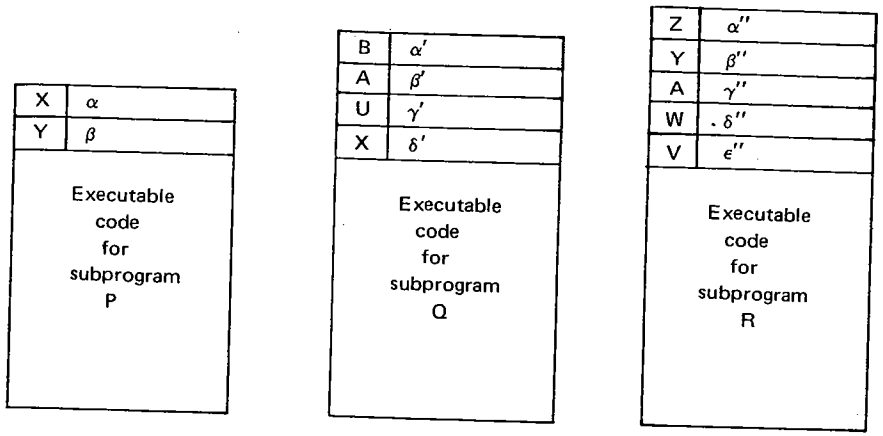



**Fig. 6-3.** Simple simulation for retained local environments.

throughout execution regardless of the number of activations of the subprogram. In this case it is appropriate to store the local environment table for a subprogram with the code for the subprogram as in Fig. 6-3. If these tables are set up during translation along with the block of executable code, then the referencing operation may be made quite efficient. Suppose that the program contains a reference to $X$. During translation it may be determined that the association for $X$ is the fourth entry, for example, in the local environment table. The referencing operation then, rather than searching the table for $X$ at execution time, may be set up to directly access the association for $X$. The referencing computation takes the base address of the table and adds a fixed *offset* to access the fourth entry. Now the table no longer needs to contain the identifiers themselves because the identifiers are never used during execution—no search of the table is ever necessary.

The simulation outlined above is typical of FORTRAN. In FORTRAN each subprogram during execution has an associated *local data block* which contains simple variables and arrays local to that subprogram. Each reference to a local variable or array in the subprogram is replaced during compilation by a direct *base address + offset* computation. The original identifiers in the program are not present during execution. Figure 6-4 illustrates this simplified structure.

Pairing the environment table with the executable code for a subprogram allows automatic shifting of the local environment

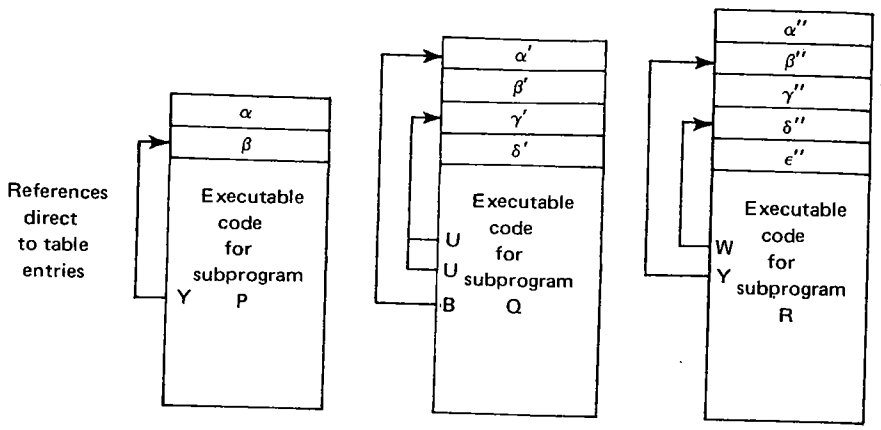



**Fig. 6-4.** Simplified local environment tables and referencing in FORTRAN.

during execution as control passes from subprogram to subprogram. Because each reference in a subprogram directly accesses the associated table defining the local environment, a subprogram call or return requires no explicit shifting of environment. In a sense the environment shifts have been *compiled in* to the executable code during translation and are implicit during execution.

Identifiers may not always be eliminated from the local environment table of a subprogram. Some languages, such as SNOBOL4 and LISP, allow an identifier to be read in during execution of a program and then used in a reference (e.g., using the $ operator in SNOBOL4). In such cases the identifiers must appear explicitly in the table, even though it may still be possible to compute direct accesses for most references.

*Destroyed and Recreated Local Environments.* Simulation of approach 2 in which the local environment is destroyed on subprogram exit and recreated on reentry may also utilize, in this simple situation, a local environment table paired with the code for the subprogram. The identifiers in the table remain the same between calls, but their associated values are destroyed on exit and new ones created on reentry. However, this solution does not generalize to recursive subprograms, and thus an alternative simulation is indicated where support of recursive calls is the ultimate goal.

The alternative simulation is based on use of a *central stack* of local environment tables. No space for an environment table is reserved with the code for a subprogram. Instead space is used in the



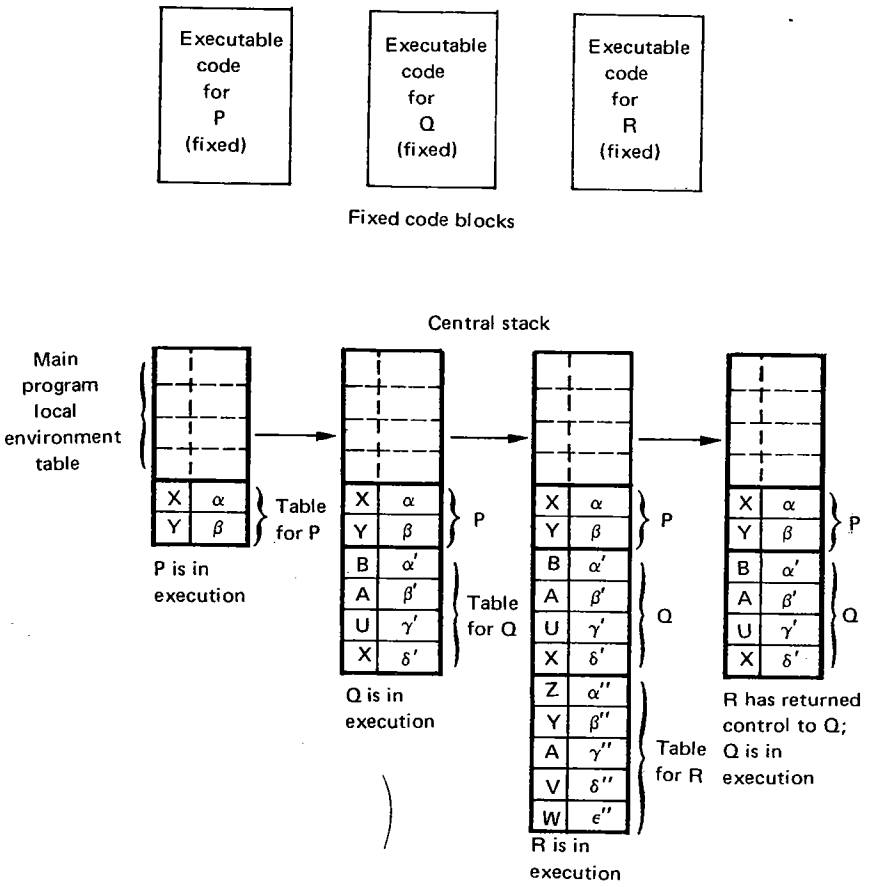



**Fig. 6-5.** Use of a central stack of local environment tables

central stack as needed for those subprograms which have actually been called. Initially the stack contains only the table for the main program. When the main program calls subprogram $P$ the table for $P$ is created and entered at the top of the stack. If $P$ calls $Q$, then $Q$'s table is stacked on top of $P$'s. When $Q$ terminates and returns control to $P$ then $Q$'s table is deleted from the stack. This stack simulation is illustrated in Fig. 6-5.

Referencing of local identifiers using the central stack is much the same as with the other technique. Although now the association table for subprogram $P$ is not created until $P$ is entered during execution, it is still possible to determine during translation the relative position of the association for $X$ within the association table.

Rather than search the table for the association for $X$ during execution the reference to $X$ may be compiled to directly access the appropriate table entry using the computation *base address of table + offset*. The base address of the table now varies during execution because the association table of $P$ may be at different positions in the central stack on different executions of $P$ (if $P$ is called from more than one routine). The offset for $X$ remains fixed. Again, if the language design does not allow creation of new identifiers during execution, then the identifiers themselves need not appear in the table, for they are not needed in referencing.

*Advantages and Disadvantages.* Both of the approaches to handling of local identifiers that we have discussed here are used in a substantial number of widely used languages. PL/I and ALGOL provide for some version of both techniques. Clearly each technique has advantages not shared by the other. Consider the situation from the programmer's viewpoint. Approach 1, in which local environments are retained from call to call, allows the programmer to write subprograms which are *self-modifying* (in the sense of Chapter 4) in that their results on each call are partially determined by their inputs and partially by the local data values computed during previous executions. Approach 2, in which local environments are destroyed on exit, does not allow any local data to be carried over from one call to the next. Subprograms in this case cannot be self-modifying; their actions may only be influenced by their inputs and by shared data, not by retained local data.

There are many cases in which approach 1 is the more natural. For example, consider a subprogram which is to maintain a table. Given an input parameter, it is to look up the input in the table, return the index of the entry if the input is already in the table, and otherwise enter the input and return the index of the new entry. Using approach 1 the table may be local to the subprogram. Since the table is retained from call to call, entries may be accumulated in the table over a sequence of calls. Approach 2 forces the table to be destroyed after each return from the subprogram if it is local. Thus the tabling routine cannot build the table over a sequence of calls unless it is defined to be nonlocal to the subprogram.

While approach 1 is perhaps the better technique in this simple case, it does not generalize as well to allow shared associations and recursion, as we shall see in the following sections. Approach 1 is somewhat more efficient to use in that no explicit change of environment is necessary when transferring control between subprograms. Approach 2 requires allocation and recovery of space on



the central stack on each entry and exit of a subprogram at some cost in execution speed. However, approach 2 provides a savings in storage space in that association tables exist only for those subprograms which are in execution or suspended execution. Association tables for all subprograms exist throughout execution in approach 1. Other advantages and disadvantages will become apparent as we try to generalize these approaches in the sections which follow.

## 6-5. SIMPLE PARAMETERLESS SUBPROGRAMS: NONLOCAL ENVIRONMENTS

Although data are ordinarily shared between subprograms through parameter transmission, there are numerous occasions in which parameters are not particularly useful. Consider a set of subprograms all making use of a common table of data. Each subprogram needs access to the table, yet transmitting the table as a parameter each time is not only inefficient but conceptually inelegant as well. Every language in Part II provides mechanisms for subprograms to share data without using parameters. Such data sharing is usually based on sharing of identifier associations. If subprograms $P$, $Q$, and $R$ all need access to the same variable $X$, then it is appropriate to simply allow the identifier $X$ to have the same association in each subprogram. The association for $X$ is then no longer part of the local environment for any one of the subprograms but has become a common part of the nonlocal environment of each of the subprograms. Data sharing through *nonlocal environments* is an important alternative to the use of direct sharing through parameter transmission. How are nonlocal environments for subprograms to be handled? Two basic approaches exist:

1. *Explicit specification of nonlocal environments.* The most direct approach requires explicit creation of shared associations. The PL/I EXTERNAL attribute provides such a feature. If two PL/I subprograms, $P$ and $Q$, tag the same identifier $X$ as EXTERNAL, then a global association is created for $X$ which may be referenced in both $P$ and $Q$ (and any other subprogram that declares $X$ as EXTERNAL). The FORTRAN COMMON block provides a slightly different structure. A COMMON block consists of a set of variables and arrays grouped into a single named *block*. The block name is global, but the identifiers associated with the individual variables and arrays are not. Any subprogram may gain access to a particular COMMON block by an explicit declaration using the appropriate

block name. If, in addition, the identical COMMON block declaration (including identical identifiers for the individual variables and arrays) is included in each subprogram sharing the same block, as is common practice in FORTRAN, the effect is as though the individual identifier associations were shared, so that a reference to nonlocal $X$ has the same association in each subprogram. The basic rule in explicit specification of nonlocal environments is that on subprogram entry a certain set of explicitly designated nonlocal associations is to be activated, as well as the usual local associations. On subprogram exit the same nonlocal associations are to be deactivated (but never destroyed).

2. *Implicit creation of nonlocal environments.* A more common approach provides for the implicit creation of a nonlocal environment for a subprogram when it is entered. The implicit nonlocal environment contains a set of associations shared with other subprograms. If a subprogram contains a reference to $X$, and no local association for $X$ exists, then the reference is taken to be nonlocal and the association for $X$ in the implicit nonlocal environment is retrieved. But what is the implicit nonlocal environment to be? Suppose that $P$ calls $Q$ which calls $R$ and that $R$ contains a nonlocal reference to $X$. Both $P$ and $Q$, and any number of other subprograms, may have a local association for $X$. In the absence of any explicit designation of which of these associations is to be used, what would be the appropriate choice? Three possibilities are

a. *Use the most recent association in the calling chain.*. If $Q$ calls $R$, and $Q$ has an association for $X$, then this association could be used when the nonlocal reference in $R$ is encountered. If $Q$ has no association for X, then we search back down the calling chain until some subprogram is found which has an association for $X$. This structure corresponds to the view that when a subprogram is called, an association for an identifier in the *calling program* is deactivated only if the same identifier occurs in the local environment of the *called routine*. The complete referencing environment of each subprogram then consists of its local environment plus the complete environment of the calling program with conflicting associations deactivated. Note that as a result the nonlocal environment of a subprogram is not fixed but varies depending on the point of call.

b. *Use the (global) associations in the main program.* We might consider that only associations in the main program may be shared, because they are intuitively global. Thus the nonlocal environment of a subprogram could consist only of the local environment of the main program with associations that conflict with local associations deactivated.



c. *Use a set of associations determined by the static program structure.* A third most important alternative, that of using the *static* structure of the program at compile time to determine the nonlocal environment, is taken up in Section 6-6.

#### Simulation of Nonlocal Environments

When nonlocal associations are explicitly specified in each subprogram the ordinary technique is to have the translator (usually the linking loader) collect these into a central table, with nonlocal references in each routine modified during loading to access this table. In cases where the data or program element which is the object of an association is also known at load time, e.g., subprogram code blocks, references may be set directly to the program or data object without the intervening association table. The translation process, particularly the loading and linking of independently translated subprograms, is made more complex by the necessity to collect and merge nonlocal associations explicitly declared in each subprogram. However, referencing during execution is straightforward, and because the nonlocal environment for all subprograms is the same, there is no need to shift nonlocal environments as control passes from subprogram to subprogram.

The use of the main program local environment as the implicit nonlocal environment for all subprograms leads to a similar simulation. The location of the execution-time association table for the main program is always fixed, being either associated directly with the code for the main program or being positioned at the bottom of the central stack, depending on the simulation of local environments used. The basic referencing rule when referencing an identifier $X$ is as follows: First check the local subprogram environment for $X$; if not found, then find $X$ in the local environment of the main program. Where both local environments are known at translation time, references may be computed directly as a base address + offset, and no search is required. Implicit Non local Environments

Simulation of implicit nonlocal environments based on use of the most recent association in the calling chain is more complex. Assume that local environments are to be created and destroyed on subprogram entry and exit, and that a central stack is used for the association tables during execution. A simple referencing rule may then be invoked: The appropriate association for a reference to $X$ is

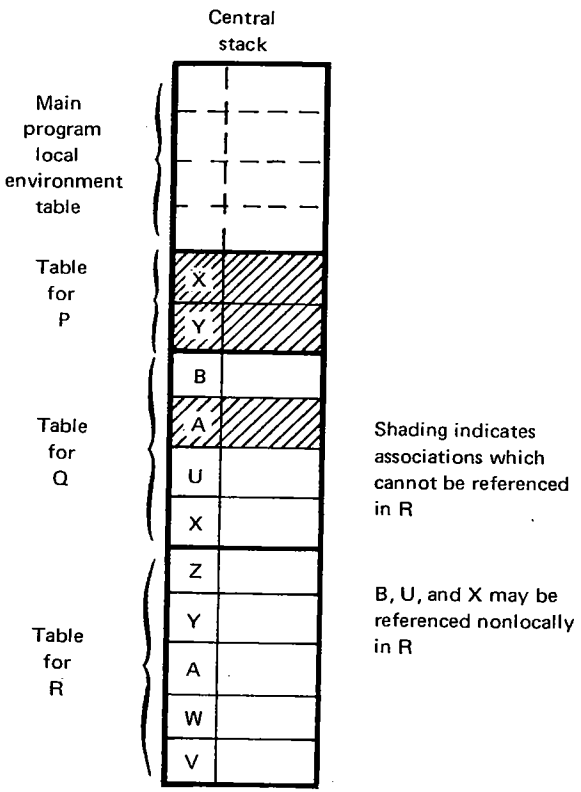



**Fig. 6-6.** Active referencing environment during execution of R (from Fig. 6-5).

found by searching the stack from the top down (i.e., search the local association table first). The first association found for $X$ in the stack is that most recently created, and thus is the proper one. Implicitly this directed search ignores conflicting associations created earlier in the subprogram calling chain. This referencing technique is illustrated in Fig. 6-6 for the example of Fig. 6-5. The LISP A-list described in Chapter 14 is based on this simulation.

A search down the subprogram calling chain may also be used where association tables are attached to the subprogram code block rather than being stored in a central stack. The search must utilize the chain of subprogram return points generated during subprogram calls to guide the search to the appropriate sequence of local association tables.

Either of these techniques for simulation of the *most recent association* form of implicit nonlocal environment introduces major execution-time inefficiency. It is the requirement of the *search* for



the most recent association which causes the difficulty. First, the search takes time. Local references may still be handled by the efficient base address + offset technique, but each nonlocal reference must invoke a search down a chain of local association tables. Because this chain is determined dynamically during execution and may vary between calls of the same routine, there is no way to avoid actually searching the chain. For example, suppose that subprogram $R$ contains a nonlocal reference to $X$. On one call, $R$ may be called from $Q$, which has a local association for $X$. On the next call $R$ may be called from $P$, which has no association for $X$, and then some association for $X$ farther back in the calling chain is needed. On yet a later call $R$ may be called from $S$ and the calling chain may contain no association for $X$ at all, causing a referencing error halt. The search also reintroduces the necessity of storing the identifiers themselves in the local association tables, because the position of the association for $X$ may differ in each local table. Thus no base address + offset computation is possible in nonlocal referencing.

How may searching for nonlocal references be avoided? Assuming we wish to retain the most recent association form of implicit nonlocal environment, a trade-off between the cost of referencing and the cost of subprogram entry and exit (plus some additional space) may be utilized. Note that both the simulations that required the search were direct extensions from the basic local environment simulations of the preceding section. Local environment tables were either stacked or stored directly with subprogram code, and the search to satisfy nonlocal references simply passed back through these local tables in the reverse order of call. As a result subprogram entry and exit were not more costly, but the nonlocal referencing was expensive because of the search. One might argue that ordinarily nonlocal referencing is likely to occur more frequently than subprogram entry and exit, i.e., the nonlocal environment is likely to be used more frequently than it is modified. In such circumstances a shift of the cost from referencing to subprogram entry and exit may be advantageous. The following simulation accomplishes this goal.

The new simulation involves augmentation of either of the local environment table simulations of the preceding section to include a central table common to all subprograms, the *central referencing environment table*. For simplicity we shall consider here the case in which local associations are not retained between calls but instead are created and destroyed on a central stack. This is the usual situation; the alternative based on retention of local environments is taken up in Problem 6-11.

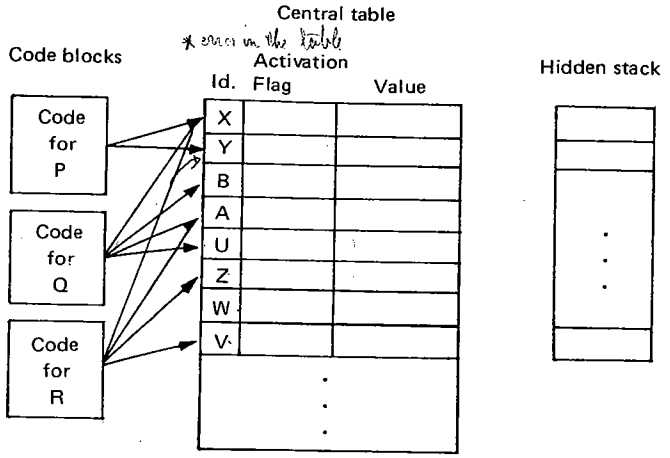
The central table is set up to contain at all times during program

execution *all the currently active identifier associations*, regardless of whether they are local or nonlocal. If we assume, also for simplicity, that the set of identifiers referenced in any of the subprograms may be determined during translation, then the central table is initialized to contain one entry for each identifier, regardless of the number of different subprograms in which that identifier appears. Each entry in the table also contains an *activation flag* which indicates whether or not that particular identifier has an active association, as well as space for a pointer to the object of the association.

All referencing in subprograms is direct to this central table using the base address + offset scheme described previously. Since the current association for identifier X is always located at the same place in the central table, regardless of the subprogram in which the reference occurs, and regardless of whether the reference is local or nonlocal, this simple referencing computation is adequate. Each reference requires only that the activation flag in the entry be checked to ensure that the association in the table is currently active. By use of the central table we have obtained our objective of relatively efficient nonlocal referencing without search.

Subprogram entry and exit is more costly, because each change in referencing environment requires modification of the central table. When subprogram $P$ calls $Q$, the central table must be modified to reflect the new local environment for $Q$. Thus each entry corresponding to a local identifier for $Q$ must be modified to incorporate the new local association for $Q$. At the same time if the old table entry for an identifier was active, the entry must be saved so that it may be reactivated when $Q$ exits to $P$. Because the entries that require modification are likely to be scattered throughout the central table, this modification must be done piecemeal, entry by entry. On exit from $Q$, the associations deactivated and saved on entry to $Q$ must be restored and reactivated. Again an execution-time stack is required, as in the earlier simulations, but it is used here as a *hidden stack* to store the deactivated associations. As each local identifier association is updated on entry to Q, the old association is stacked in a block on the hidden stack. On return from Q the top block of associations on the stack is restored into the appropriate positions in the central table. This central table simulation is shown in Fig. 6-7. An additional advantage accrues when using the central table if the language does not allow new references to be generated during execution. In this case, as was the case earlier in regards to local tables, the identifiers themselves may be dropped from the table, for they will never be used, having been replaced by the base address + offset computation. (In a sense the identifier is simply represented by its table offset during execution).






Basic referencing organization using central table

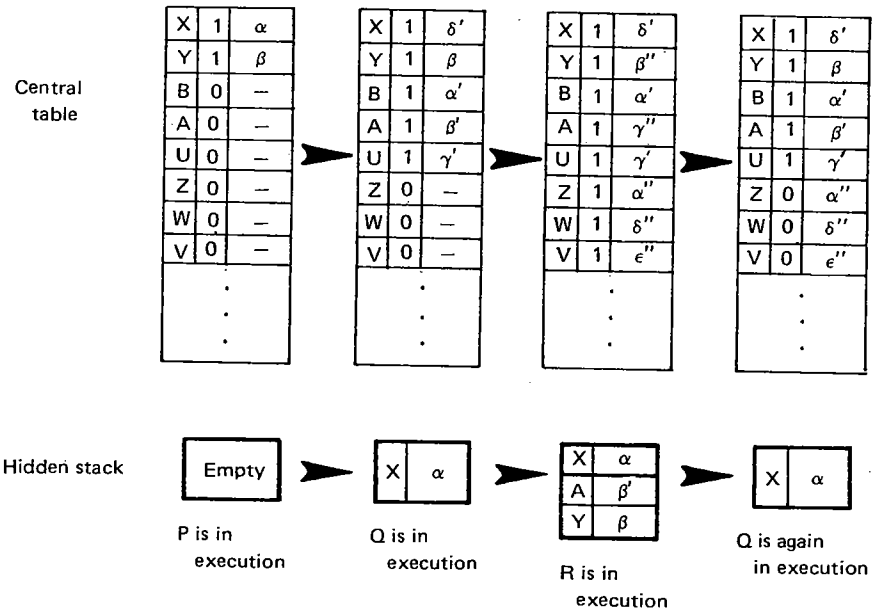



Table and stack transformation during execution (refer to Fig. 6-5)

**Fig. 6-7.** The central environment table simulation.

The central environment table has many variants. The SNOBOL4 implementation described in Chapter 15 is based on this simulation. Other variants are taken up in the problems at the end of this chapter.

## 6-6. NONLOCAL ENVIRONMENTS BASED ON STATIC PROGRAM STRUCTURE

ALGOL, PL/I, and many other languages utilize a technique for definition of nonlocal subprogram environments that differs substantially from those mentioned in the preceding section. This technique, which is based on the compile-time (static) program structure, warrants a separate discussion because of its complexity and importance.

Before discussing the technique, let us look first at some of the shortcomings of the simpler techniques of the preceding section. Two techniques, explicit specification of shared associations and implicit use of the main program's local environment, allow only a single *global* nonlocal environment common to all subprograms. Thus every reference is either local or global. This restricted case may be readily simulated but is rather inflexible. It is desirable to allow different routines to share associations without making them global to all routines. The third technique mentioned, that of using the most recent association, allows greater flexibility and in addition corresponds rather closely to our intuition about the meaning of nonlocal references in most cases. Unfortunately the most recent association technique has a major flaw in *compiled* languages such as ALGOL and PL/I where execution speed is a primary factor.

Consider the ALGOL program of Fig. 6-8. Subprogram *P* is called in two places, from statement *L1* and also statement *L2*. Consider the assignment *X* := *X* + *Y* in *P*. Because *P* contains no declaration for *X* or *Y*, the references in the assignment statement are nonlocal. Suppose that ALGOL used the most recent association rule; then on the first call of *P* from statement *L1* both *X* and *Y* are **real** variables, but on the second call from statement *L2* both are **integer** variables. Accordingly a real addition is required in the first case and an integer addition in the second. This run-time variability requires a run-time type check followed by a branch to the appropriate addition operation. Run-time type checking, however, is manifestly undesirable because of the time required to make the test and the space required to store the extra code. The result: A substantial and



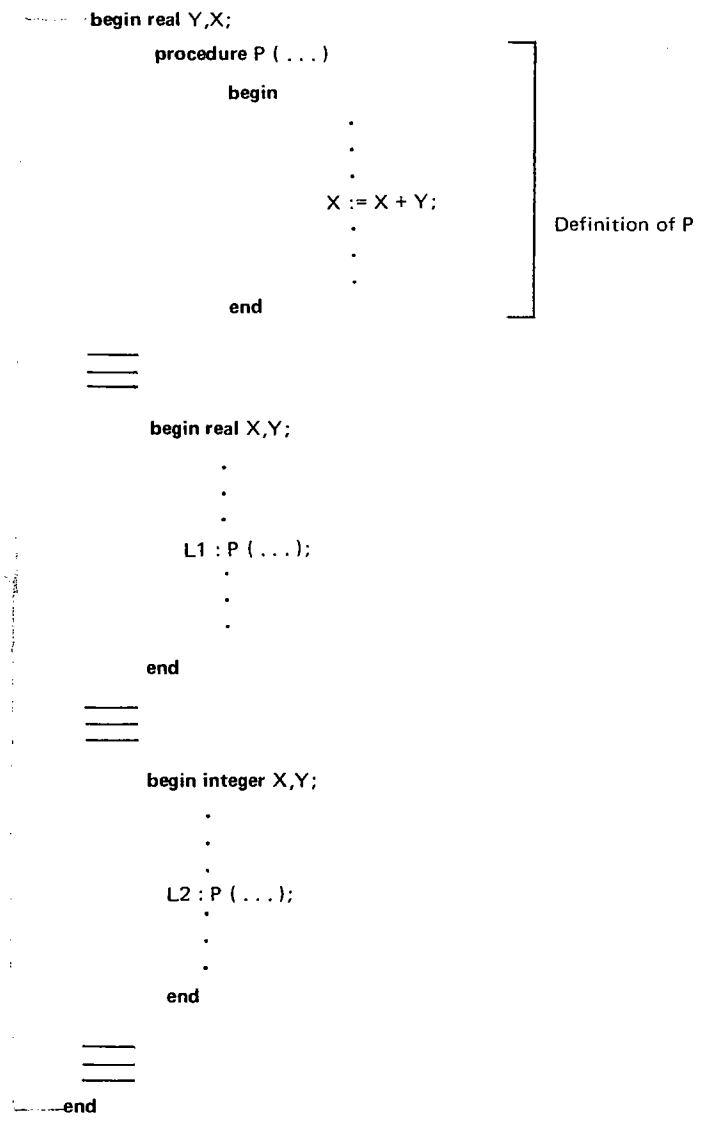



Fig. 6-8. An ALGOL procedure with nonlocal references.

unavoidable run-time cost if the most recent association rule is adopted in defining nonlocal environments. In languages such as LISP, SNOBOL4, and APL the rule is used because run-time type checking is needed anyway. However, in ALGOL and PL/I the cost

at run time is too high. An alternative which allows type checking to be done during translation is desirable.

The alternative adopted in ALGOL, and subsequently incorporated into PL/I and other languages, is rather subtle. If static (compile-time) type checking is to be possible for nonlocal references, then each nonlocal reference must be paired uniquely with a declaration *during translation*. To provide this pairing we must move back to the syntactic representation of sets of subprograms (and blocks) and introduce the concept of a syntactic nesting of definitions of subprograms. To this point we have said nothing about the syntax of subprogram definition. We assumed that a set of subprograms had been defined somehow and concerned ourselves only with the problems of run-time identifier associations. Now, however, we wish to associate type descriptors (at least) with identifiers during translation. We accomplish this as follows. First we adopt a nested syntax for subprogram and block definitions, so that a program consists of an outermost block containing nested block and subprogram definitions as in ALGOL. A typical program structure in ALGOL then looks like Fig. 6-8. Now we adopt the following rule for referencing nonlocal identifiers: The variable referenced by a nonlocal identifier in a block or subprogram definition is that variable declared within the innermost *physically containing* block or subprogram in the program *as written*. Now in our old example of Fig. 6-8, $Y$ in procedure $P$ is a reference to the variable $Y$ declared in the outermost block, not to either of the variables $Y$ declared in blocks from which $P$ is called.

Using this rule of referencing based on the compile-time nesting of subprogram and block definitions we can always associate each identifier with a declaration at compile time and thus avoid run-time type checking. The compile-time nesting of block and subprogram definitions is usually termed the *static block structure* of the program.

*Simulation.* The use of static block structure to determine the implicit nonlocal environment of a subprogram imposes a cost in the complexity of the run-time simulation needed. The usual simulation is based on a central stack of local environment tables, suitably modified to allow referencing based on the static block structure of the original program. We shall discuss this simulation here, leaving other simulations to be developed in the problems. This simulation is typical of many ALGOL implementations and is based on the assumption that local data are not retained between subprogram calls.



Recall from Section 6-4 the simple stack-based simulation for local environment tables. Initially the stack contains only the local environment table for the main program. As each subprogram is entered a local environment table is created on top of the stack, and on exit the top environment table on the stack is deleted. Assuming that subprogram calls are strictly nested in a last-in—first-out fashion, we may always be sure that the top local environment table on the stack at subprogram exit is the table for that subprogram. Referencing of local variables utilizes the base address + offset computation, where the base address is always that of the top local table on the stack. This part of the original simulation is still adequate.

Nonlocal referencing brings major problems, however. Consider Fig. 6-8 again. Figure 6-9 shows the contents of the stack when execution reaches the assignment $X := X + Y$ on the first call of $P$. Note the difficulty of referencing the appropriate association for $Y$ because the appropriate association is not the most recently created. It no longer suffices to simply search down the stack. The problem is that the sequence of local tables in the stack represents the *dynamic nesting* of subprogram *activations*, the nesting based on the execution-time calling chain. But it is the *static nesting* of subprogram *definitions* which now determines the nonlocal environment, and the stack as currently structured contains no information about static nesting.

To complete the simulation it is necessary to represent the static

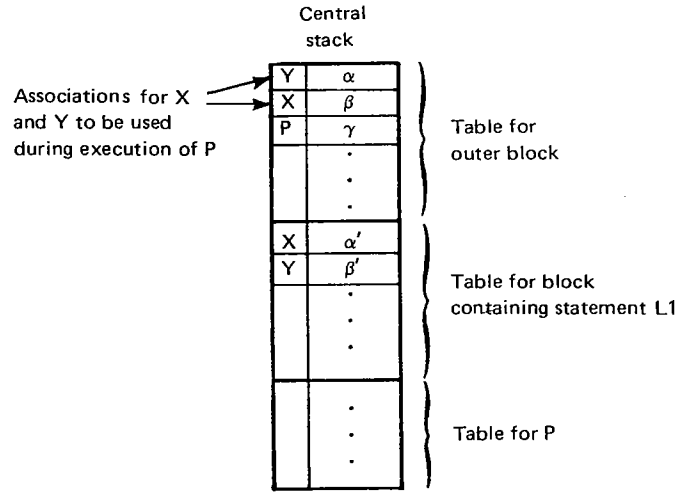



**Fig. 6-9.** Environment table stack during execution of P (Fig. 6-8).

block structure during execution in such a way that it may be used to control nonlocal referencing. Observe that in many respects the rule for nonlocal referencing in this case is similar to that for nonlocal referencing using the most recent association rule: To find the association to satisfy a reference to $X$ we search a chain of local environment tables until an association for $X$ is found. However, the chain of local environment tables to search is not that composed of *all* the local tables currently in the stack but only those currently in the stack *which represent blocks or subprograms whose definition statically encloses the current subprogram definition in the original program text*. The search then is still down the tables in the stack, but only a subset of those tables are actually part of the referencing environment.

*Static Chain Simulation*. These observations lead to the most direct simulation of the referencing environment: the *static chain* technique. Suppose that we modify the local environment tables in the stack slightly so that each table begins with a special entry, called the *static chain pointer*. This static chain pointer always contains the base address of another local table further down the stack. The table pointed to is the table representing the local environment of the statically enclosing block or subprogram in the original program.

The static chain pointers form the basis for a simple referencing scheme. To satisfy a reference to $X$ we begin with the current local environment on top of the stack. If no association for $X$ is found in the local environment, then we follow the static chain pointer in that local table down the stack to a second table. If $X$ is not in that table, the search continues down the static chain pointers until a local table is found with an association for $X$. The first one found is the correct association. Figure 6-10 illustrates the static chain for the ALGOL program of Fig. 6-8.

At any point during program execution the static chain pointers in the stack actually structure the local environment tables into a tree, as illustrated in Fig. 6-11. The referencing operation always begins looking for an association at one of the leaves of the tree, progressing down the branches toward the root in the search. One leaf of the tree is always the current local environment (the table on top of the stack), and it is from this leaf that the referencing search begins.

If a nonlocal reference involves a *search* of a sequence of local environment tables, then we have returned to one of the problems which caused difficulty before. Recall that one of the reasons for introducing the additional complexity of referencing based on the



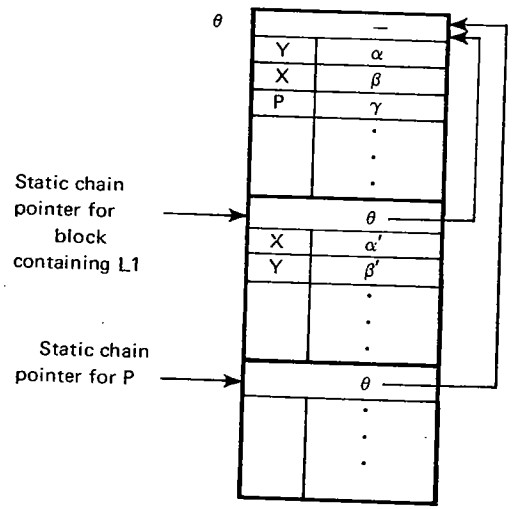



**Fig. 6-10.** Environment table stack of Fig. 6-9 with static chain pointers.

static block structure was execution efficiency, yet searching to satisfy references at run time is most inefficient. However, the search here may be eliminated without difficulty. To see how, we need a few preliminary observations.

First note that, for any subprogram $Q$, when $Q$ is in execution (and its local environment table is therefore on top of the stack) the length of the static chain leading from $Q$'s local table down the stack (and ultimately to the table for the main program) is *constant*. This length is constant regardless of the current size of the stack and regardless of the point of call of $Q$. Of course, the reason is simply that the length of the static chain is equal to the depth of static nesting of subprogram $Q$'s definition back in the original program at compile time, and this depth of nesting is fixed throughout execution. For example, if $Q$ is defined within a block which is directly contained within the outermost block of the program, then the static chain for $Q$ during execution always has length 3: $Q$'s local table, the local table for the directly containing block, and the local table for the outermost block (the main program). In Fig. 6-8 and 6-10, for example, the static chain for $P$ always has length 2.

Second, note that in this chain of constant length a nonlocal reference will always be satisfied at exactly the same point in the chain. For example, in Fig. 6-10 the nonlocal reference to $X$ in $P$ will always be satisfied in the second table in the chain. Again this fact is a simple reflection of the static program structure. The number of

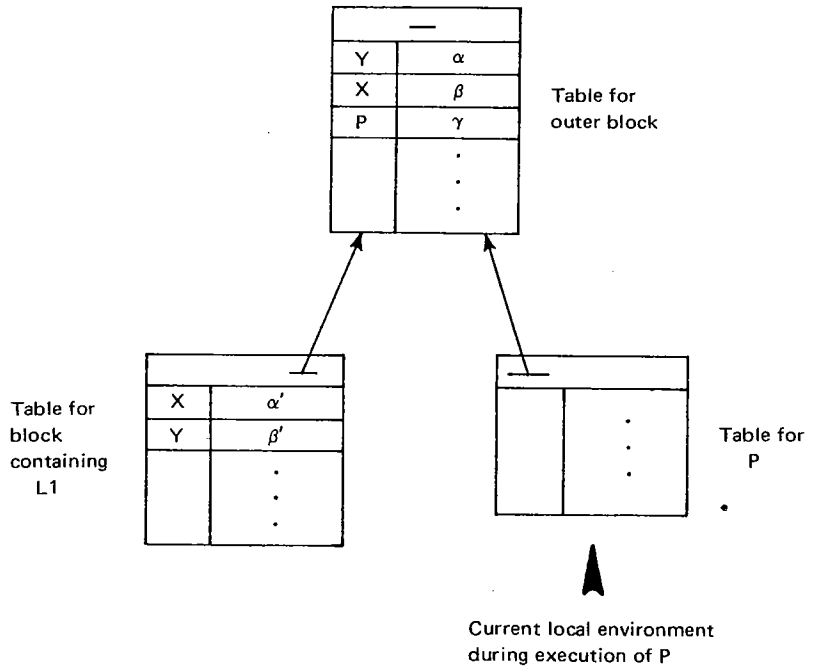



**Fig. 6-11.** Tree of local environment tables during execution of P.

levels of static nesting that one must go out from the definition of $P$ to find the declaration for $X$ is fixed at compile time.

Third, note that the position in the chain at which a nonlocal reference will be satisfied may be determined at compile time. For example, we may determine at compile time that a reference to $X$ in $P$ will be found in the second table down the static chain during execution. In addition, we know at compile time the relative position of $X$ in that local table. Thus, for example, at *compile time* we can conclude that the association for $X$ will be the second entry in the second table down the static chain during execution.

The basis for a fairly efficient referencing operation is now apparent. Instead of explicitly searching down the static chain for an identifier, we need only skip down the chain a fixed number of tables, and then use the simple base address + offset computation to pick out the appropriate entry in the table. In fact, under the usual assumption that new references cannot be generated during execution, we no longer need the identifiers represented explicitly at all, because no search is used. Instead it is natural to represent an identifier in the form of a pair, (chain position, offset), during



execution. For example, if $X$, referenced in $P$, is to be found as the third entry in the second table down the chain, then in the compiled code for $P$, $X$ may be represented by the pair (2,3). This representation provides a rather simple referencing algorithm.

The static chain technique allows straightforward entry and exit of subprograms. When a subprogram is called, its local environment table must be installed on top of the stack and the appropriate static chain pointer installed, pointing to a local table further down the stack. On exit it is necessary only to delete the top local table from the stack; no special action is needed. But how is the appropriate static chain pointer to install on entry to be determined? Suppose that subprogram $P$ is called from subprogram $Q$ and that $P$ was defined in the original program within block $B$. When $P$ is entered the appropriate static chain pointer is back to the local table for block $B$. At the point of call the local table for $Q$ is on top of the stack and that for $P$ is to be stacked on top of $Q$'s. How is it determined that the proper pointer is the one to $B$? Observe that the identifier $P$, the subprogram name, is itself referenced nonlocally in $Q$. If $P$ is represented by the pair (3,2), then the association of identifier $P$ with a pointer to its compiled code is to be found in the table three steps down the static chain for $Q$, at the second table entry. But that local table in which the entry for $P$ appears must be the table for block $B$. Thus when we reference $P$ at the point of subprogram call (in $Q$) we can also easily retrieve the appropriate static chain pointer to insert into the local table for $P$ when it is set up. This structure is illustrated in Fig. 6-12.

*The Display Simulation.* The necessity of following the static chain for each nonlocal reference is a drawback of the above technique. We may avoid this referencing cost by using an alternative simulation, but only at a cost on subprogram entry and exit. In this simulation the static chain is represented separately from the main stack. The current static chain is stored separately in a special "little stack," termed a *display*. The display contains a sequence of pointers to local tables in the main stack. At any given point during execution the display contains the same sequence of pointers that would have occurred in the static chain currently being used for nonlocal referencing in the old static chain model. Figure 6-13 illustrates the display for the example of Fig. 6-12.

Referencing using a display is particularly simple. Let us adopt a slightly modified representation for identifiers during execution. Again pairs of integers are to be used, but let the 3 in a pair like (3,2) represent the number of steps back from the *end* of the chain to the

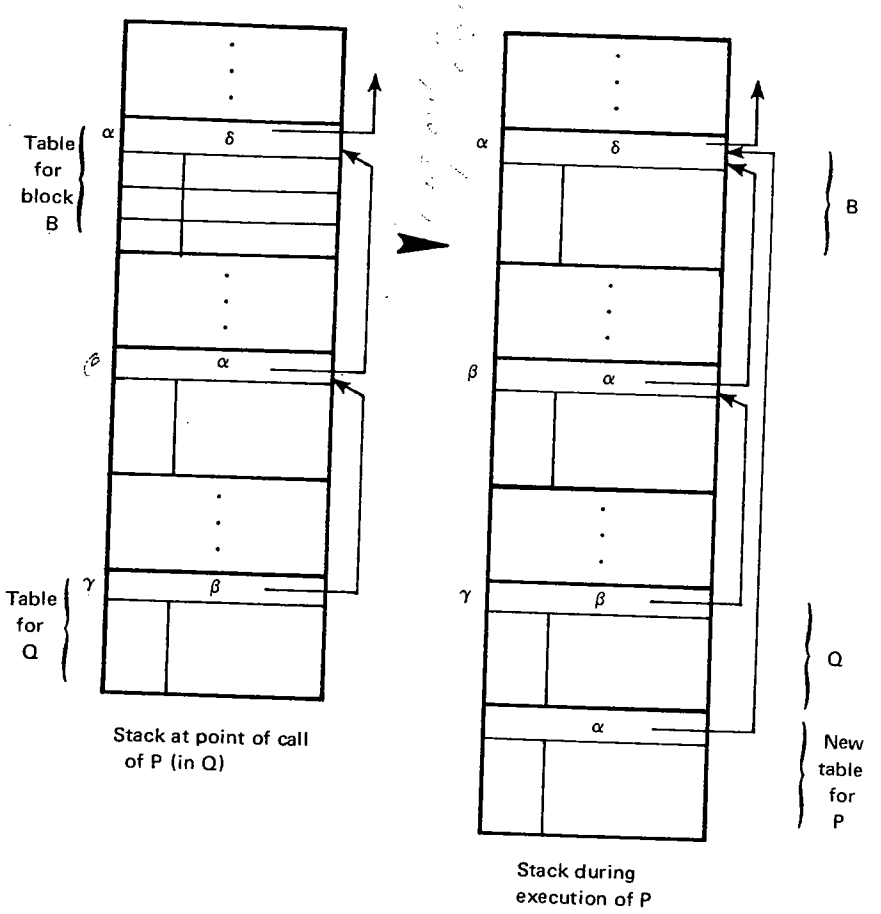



Fig. 6-12. Static chain creation during subprogram entry.

appropriate local table (rather than down from the start of the chain as before). The second integer in the pair still represents the offset in the table. Now given a nonlocal reference such as (3,2) the appropriate association is found in two steps:

1. Consider the first entry (3) as a subscript into the display. Thus *display* [3] contains a pointer to the appropriate local table, i.e., the base address of the local table.

2. Compute the location of the appropriate table entry as base address + offset, where the offset is the second entry in the pair.



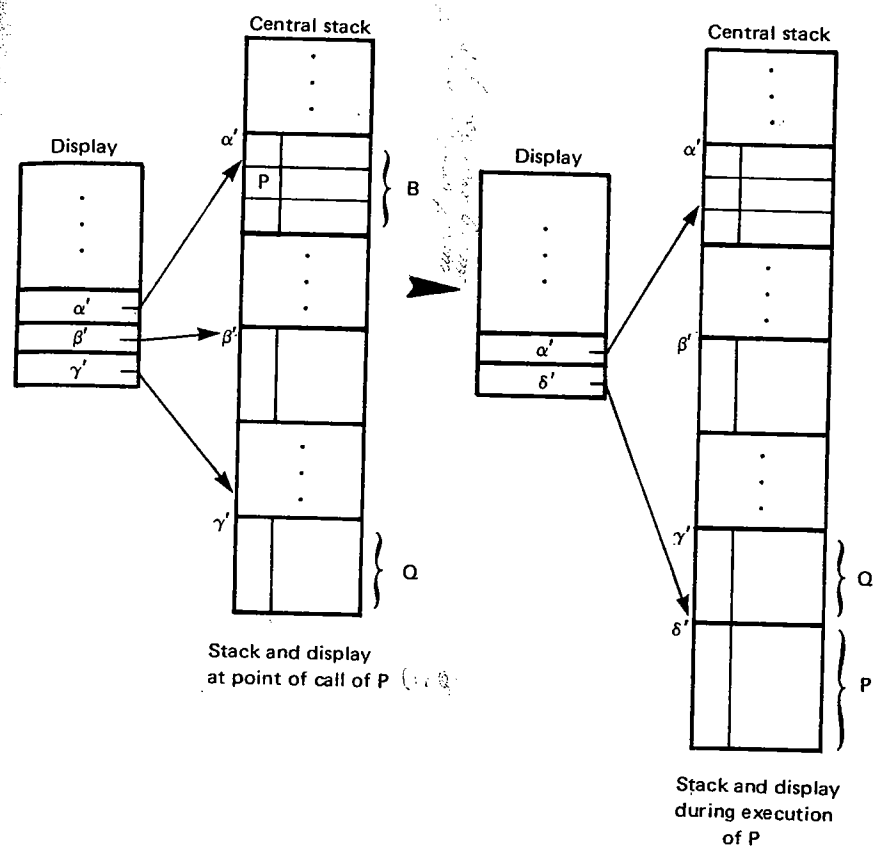



Fig. 6-13. Display modification during subprogram entry.

Ordinarily these two steps combine into one using indirect addressing through the display entry. In addition, it is often possible to place the display directly in high-speed registers during execution, thus giving only one memory access per identifier reference.

Although referencing is simplified using a display, subprogram entry and exit are more difficult. The display must be modified on each entry and exit to reflect the *currently active* static chain at all times. Consider again the subprogram $P$, defined in block $B$, which was called from subprogram $Q$. At the point of call the display contains the static chain for $Q$. This static chain contains a pointer to the local table for $B$ (otherwise $Q$ could not reference $P$). The static chain for $P$ contains only the local table for $P$ and the old chain from $B$ down. Thus on entry to $P$ the contents of the display from the

entry for $B$ to the top must be saved and later restored on exit. Ordinarily this partial static chain would be stored on the central stack with the local table for $Q$. The cost of saving and restoring the display is greater than with the simple static chain technique.

### Advantages and Disadvantages of Nonlocal Referencing Techniques

In this and the preceding sections various approaches to the sharing of associations through explicit and implicit nonlocal environments have been surveyed. The languages in Part II utilize these techniques, and various modifications, in many different ways. There clearly is no general agreement as to the "best" way to handle nonlocal referencing. Let us consider the three main techniques: (1) explicit designation of nonlocal associations, (2) implicit nonlocal environment using the *most recent association*, and (3) implicit nonlocal environment using the *static block structure*.

Explicit designation has the advantage of simplicity of simulation (at least as ordinarily set up), as only a single global environment is defined. In addition, explicit designation is likely to provide good error protection, as references which are neither local nor explicitly designated nonlocal may be detected during translation. Explicit designation, on the other hand, is less flexible in that only a single level of nonlocal referencing is provided. Also it may require a substantial amount of extra declaration when much nonlocal referencing is done.

Implicit most recent association environments are perhaps the most elegant, as they correspond closely to our intuitive understandings of the meanings of identifiers in mathematics (for example, as modeled in the lambda calculus). They are simple to understand and to simulate, but they may force type checking during execution, causing substantial inefficiency.

Static block structure environments allow static type checking and fairly efficient simulation during execution. However, they force the sort of nested block and subprogram definition characteristic of ALGOL programs, and thus the technique cannot easily be extended to separately translated subprograms (a problem not shared with either of the other techniques).

## 6-7. RECURSIVE PARAMETERLESS SUBPROGRAMS

More general subprogram control structures lead to variations on the data control structures described for simple subprograms. In the preceding sections we have identified two basic approaches to local environments:

1. Retention of local environments between calls.
2. Destruction and recreation of local environments between calls.

and three basic approaches to nonlocal environments:

1. Explicit specification.
2. Implicit dynamic (most recent association) specification.
3. Implicit static (static block structure) specification.

How do recursive subprograms affect the choice of referencing environments? Recall from Section 5-4 the distinction between subprogram *definition* and *activation*, which is central in recursion. A recursive subprogram may exist in many simultaneous activations at any point during execution. The referencing environment in different activations of the same subprogram in general may be different. Thus we are confronted with the problem of defining the referencing environment for *each activation* of a subprogram.

*Local Environments.* Each activation of a recursive subprogram should have its own local environment. For example, if $X$ is a local variable in $P$, and $P$ calls itself recursively, then we expect a new local variable named $X$ in each activation of $P$. This fact implies creation of a new local environment on each call. Similarly when control leaves $P$ the local environment should be destroyed. Retention between calls is also a possible technique, interpreting retention to mean that a single local environment is set up which is used by all activations of a given subprogram. However, retention is not particularly useful in recursion because one ordinarily desires a new set of local associations at each level of recursion. Thus recursion usually implies destruction and recreation of local environments on subprogram entry and exit. With either technique no special simulation is required to handle local environments. When the central stack simulation is used, however, the same stack may be used to store subprogram return points (see Section 5-4).

*Nonlocal Environments.* Nonlocal referencing in recursive subprograms causes no particular problem in any of the three approaches to nonlocal environments. Each extends immediately to recursion. Consider a nonlocal reference to $X$ in some activation of a

recursive subprogram $P$. In the explicitly specified nonlocal environment, $X$ is global and thus exists in only one activation. In the static block structure technique the reference to $X$ will retrieve the association for $X$ in the most recent activation of the subprogram in which $X$ was declared. The most recent association technique, of course, retrieves the association in the most recent activation in the calling chain.

## 6-8. REFERENCING ENVIRONMENTS IN COROUTINES, INTERRUPT ROUTINES, TASKS, AND SCHEDULED SUBPROGRAMS

Control structures more general than simple subprograms with recursion present new referencing environment requirements. We shall discuss the problems briefly, without attempting a thorough treatment.

### Coroutines

Local environments in coroutines obviously need to be retained from one RESUME call to the next; otherwise the coroutine would not be able to resume at the appropriate point in its computation. Nonlocal environments are more troublesome. Conceptually a set of coroutines operate all on the same level; there is no calling program-called program hierarchy. For this reason the most recent association implicit environment is unnatural; there is no calling chain on which to base a search for the appropriate reference. Either explicit specification or implicit static specification is appropriate. As few languages have incorporated coroutines directly, there is little experience to draw on here.

### Interrupt Routines

Subprograms initiated by interrupts present a different set of referencing problems. Local environments might be handled with any of the approaches mentioned. Nonlocal referencing in interrupt routines is particularly important, however. Consider the typical case of an interrupt routine $P$ which is to be executed when a particular type of error occurs in subprogram $Q$, say, for example, a subscript-out-of-range error. If $P$ is to function adequately, it must have access to information about the array associated with the error, its bounds, the values of the subscripts in the reference, etc. In



general these will be local to $Q$. Thus it is desirable that $P$ have access to $Q$'s local environment, or at least to some of the information available through that environment. As a second example, consider a trace routine that is to be activated whenever a particular subprogram is called. Again, the trace routine needs access to the local environment of the calling program if it is to output the current values of variables in that program. How is this access to be provided? One simple solution is that used in PL/I. Rather than allowing the interrupt routine direct access to the local environment of the subprogram in which the interrupt occurred, certain fixed pieces of data which are central to the processing of the particular type of interrupt are transferred to a global environment (language-defined). The interrupt routine may access this global environment but not the local environment where the interrupt occurred. This simple technique is adequate in many cases. Relatively little is known about general techniques for referencing in interrupt routines.

### Tasks

Parallel execution (tasks) again creates little difficulty in local environments. Under the usual assumption that multiple activations of the same subprogram may be executing in parallel it is natural to create a new local environment when a task is initiated and destroy it when the task terminates.

Nonlocal referencing is not difficult conceptually but may cause problems in storage management. Suppose that $P$ calls $Q$ as a separate task to be executed in parallel. The nonlocal environment of $Q$, if not defined explicitly, would be either the environment of $P$ (in the most recent association technique) or some part of the environment of $P$ (in the static block structure technique). The problem is that as both $P$ and $Q$ continue to execute, each may be expected to enter new blocks or call other subprograms, thus generating separate chains of new local environments. In the stack simulation, initiation of a new task leads to a fork in the stack. Each branch of the fork continues to grow as execution of the two parallel subprograms continues. Of course, each may initiate other tasks, causing further splitting of the stack. In general, a tree structure results. When a task terminates, that branch of the tree, from the leaf down to the last branch point, must be deleted. Because this tree structure grows and shrinks unpredictably during execution, a general storage management mechanism is required to simulate it; the simple stack technique no longer suffices.

### Scheduled Subprograms

General subprogram scheduling, as is found in simulation languages, causes the most difficulty in referencing. Each subprogram may exist simultaneously in many activations, each in various stages of execution. Termination of any activation of a subprogram may occur at an arbitrary point relative to other activations of the same subprogram and other subprograms. Thus there is no nicely nested pattern to subprogram calls and terminations. In addition, subprogram activations may be suspended and reactivated in a general coroutine-like manner. The result is an essentially random pattern of subprogram activation, suspension, reactivation, and termination.

Clearly deletion of lcoal environments is natural here on subprogram termination. But what is appropriate for nonlocal environments? We might use the dynamic environment at the point of call (more appropriately, point of *scheduling*), but that environment may no longer exist when the scheduled subprogram actually begins execution. An environment based on the static program structure does not answer the question of which activation is to be used. And a single global environment is somewhat inflexible, for it leaves the programmer to entirely control the sharing of data between subprograms.

No single solution is likely to be entirely adequate in this case. Different simulation languages provide different mechanisms. GPSS, for example, allows each activation of a routine (called a *transaction*) to access only local or global environments. The programmer must coordinate the sharing of data through the global environment common to all transactions.

## 6-9. SUBPROGRAMS WITH PARAMETERS: PARAMETER TRANSMISSION TECHNIQUES

Communication between subprograms through parameters is more common than communication through nonlocal environments. The intent—allowing data and program elements to be shared between subprograms—is the same in either case. Parameters are most useful when a subprogram is to be given different data to work on each time it is called. Use of a nonlocal environment is more appropriate when the same data are used on each call. For example, if subprogram $P$ is used on each call to enter a new data item into a table shared with other subprograms, then typically the table would



be accessed through a nonlocal reference in $P$, but the data item would be transmitted as an explicit parameter on each call.

In discussion of parameter transmission we must distinguish between *actual parameters* (or *arguments*) and *formal parameters*. *Formal parameters* are the identifiers used in a subprogram definition to name the data or program elements transmitted into the subprogram. Formal parameters are ordinarily specified in a *formal parameter list* at the beginning of the definition of the subprogram and are invariably restricted to be simple identifiers. *Actual parameters* are the expressions used at the point of call of a subprogram to specify the data or program items to be transmitted to the subprogram. The usual syntax is to simply write actual parameters in a list following the name of the subprogram, as in $SUB(X2*Y,17)$. As discussed in Section 5-2 this is the ordinary prefix representation for operations, in which the operands follow the operation symbol. Other notations may be used, e.g., infix notation. For simplicity we shall adopt the conventional prefix representation as basic and speak of *argument lists*, although for other syntactic representations arguments may not be represented explicitly in lists.

Parameter transmission provides a mechanism for allowing a subprogram access to nonlocal data and program elements through strictly local identifiers. Through parameters one may avoid the complexities of nonlocal environments. Formal parameters are local identifiers in a subprogram and thus form part of the local referencing environment. Parameter transmission provides a means of initializing these local identifiers to nonlocal values on subprogram entry, through a simple *positional correspondence* with the actual parameters specified at the point of call. At the point of call there is an actual parameter list; at the head of the subprogram definition there is a formal parameter list. At the time of subprogram entry the first formal parameter is associated with the data or program element designated by the first actual parameter; the second formal parameter is associated with the second actual parameter; etc. In general, it is required that there be exactly as many actual as formal parameters, although many languages relax this requirement by conventions for interpreting missing or extra actual parameters. It is through use of positional correspondence between formal and actual parameter lists that we avoid the necessity to share identifier associations through nonlocal environments.

The basic concept of parameter transmission through positional correspondence in parameter lists is straightforward. The complexity here is due largely to two aspects: the variety of types of actual

parameters and the variety of parameter transmission techniques. Let us begin with a description of the basic techniques for parameter transmission and then discuss the different varieties of actual parameters and their transmission using the various techniques.

### Basic Parameter Transmission Techniques

In the preceding chapter we looked briefly into the problem of evaluation rules for operands to operations: Should the operand be evaluated before transmission to the operation, or should the operand be transmitted unevaluated? We are now ready to look more deeply into this problem, because parameter transmission is simply the topic of evaluation rules expanded. An actual parameter specifies an operand for a subprogram (which is only a user-defined operation). Our concern is with what operand the subprogram actually receives and the manner of its association with the formal parameter. Three main types of parameter transmission have been used in programming languages: transmission by value, transmission by reference, and transmission by name. Each leads to a different evaluation rule for actual parameters.

*Transmission by Value.* In parameter transmission by value the basic rule is that the actual parameter is evaluated at the point of call. The *value* of the actual parameter is then transmitted to the subprogram and becomes the initial value associated with the corresponding formal parameter. For example, in CALL SUB(X), if X is a simple variable with the value 3 at the point of call, then the value transmitted is the constant 3. The corresponding formal parameter, say Y, is initialized to the value 3, and during execution of the subprogram SUB, Y is treated as a simple local variable. The major distinguishing characteristic of transmission by value is its restriction to transmission of data *to* a subprogram. In general, a subprogram cannot transmit results back to the calling program through a parameter transmitted by value. Thus transmission by value "protects" the calling program from side effects caused by assignments to a formal parameter within the called program. Exceptions occasionally arise, e.g., in SNOBOL4 all parameters are transmitted by value, but results may be returned through parameters by invoking indirect referencing through a variable name transmitted as a string.

*Transmission by Reference (Location or Simple Name).* In transmission by reference a pointer is transmitted, usually a pointer to a data location containing the value. In CALL SUB(X), for



example, transmission by reference would lead to transmission of a pointer to the location of variable X in the calling program, in contrast to the value of variable X which was transmitted in the transmission by value technique. The corresponding formal parameter Y in SUB is initialized to contain the pointer to X. Each reference to Y in SUB is treated as a reference to the location of X. Any assignment to Y in SUB will change the value of X back in the calling program. Thus transmission by reference allows data to be transmitted both into and back from subprograms.

*Transmission by Name.* The basic concept in parameter transmission by name is that of leaving actual parameters *unevaluated until the point of use* in the called subprogram. The parameters are to be transmitted unevaluated, and the called subprogram determines when, if ever, they are actually evaluated. Recall from our earlier discussion of uniform evaluation rules that this possibility was useful in treating operations such as the *if-then-else* conditional as ordinary operations. In primitive operations the technique is occasionally useful; in programmer-defined subprograms its utility is more problematic because of the cost of simulation. Parameter transmission by name plays a major role in ALGOL and is of considerable theoretical importance. It has not been widely used outside of ALGOL, being replaced in later languages by more appropriate mechanisms based on subprograms as actual parameters.

The basic transmission by name rule may be stated in terms of substitution: The actual parameter is to be substituted everywhere for the formal parameter in the body of the called program before execution of the subprogram begins. While this seems straightforward, consider the problem of even the simple CALL SUB(X). If the formal parameter in SUB is Y, then X is to be substituted for Y throughout SUB before SUB is executed. But this is not enough, because when we come to a reference to X during execution of SUB, the association for X referenced is that *back in the calling program*, not the association in SUB (if any). When X is substituted for Y we must also indicate a different referencing environment for use in referencing X. This is precisely the problem which arises with subprogram parameters in general, a topic which we shall take up in detail below.

Not surprisingly, the basic technique for simulating transmission by name is to treat actual parameters as simple parameterless subprograms (traditionally called *thunks*, a name coined by Ingerman [1961]). This technique allows a uniform handling of the problems of referencing environments for both *by name* parameters and

subprogram parameters. Whenever a formal parameter corresponding to a by name actual parameter is referenced in a subprogram, the thunk compiled for that parameter is executed, resulting in the evaluation of the actual parameter in the proper referencing environment and the return of the resulting value (or location) as the value of the thunk.

### Actual Parameter Types and Parameter Transmission

Different actual parameter types usually receive different treatment depending on whether transmission is by value, reference, or name. Understanding of the differences in effect between the three transmission mechanisms is aided by considering the various actual parameter types individually.

*Simple Variables.* (1) *By value.* The variable is evaluated at the point of call (i.e., the location associated with the variable name in the referencing environment at the point of call is found, and the contents of that location retrieved) and the value is transmitted to the subprogram. The value becomes the initial value of the corresponding formal parameter, which then is treated as a local simple variable throughout subprogram execution. An assignment to the formal parameter does not change the value of the actual parameter variable back in the calling program. (2) *By reference.* The location associated with the variable name in the referencing environment at the point of call is found. A pointer to this location is transmitted. A reference to the formal parameter in the called subprogram is treated as an indirect reference to the actual parameter location through this pointer. Assignment to the formal parameter changes the value of the actual parameter in the calling program. (3) *By name.* This case may be treated as transmission by reference.

*Constants.* (1) *By value.* The constant is transmitted and set as the initial value of the formal parameter. (2) *By reference.* A storage location for the constant is allocated and initialized to the constant value. A pointer to this location is transmitted. Note: Assignments to the formal parameter may cause difficulties if each actual parameter constant is not allocated a separate storage location. For example, suppose that SUB is called by CALL SUB(1,2,1), that the constant 1 is stored only once, and that two pointers to the location are transmitted as actual parameters. If the corresponding formal parameter list in SUB is (X,Y,Z), then an assignment to X may change the value of Z inadvertently. This error is prevalent in many implementations of FORTRAN where all uses of a constant in any subprogram are translated into references to a single location containing the constant. It becomes possible to change the value of a constant 1 to 2 during a subprogram call, so that any later reference to the constant 1, e.g., in X := X+1, evaluates to 2. (3) *By name.* This case may be treated as transmission by reference or value (but assignments to the formal parameter should be prohibited).



*Expressions (Other than Special Cases Listed Separately).* (1) *By value.* The expression is evaluated at the point of call and the resulting value transmitted. The formal parameter is treated as a local simple variable initialized to the transmitted value. (2) *By reference.* The expression is evaluated at the point of call and the value stored in a reserved location. A pointer to this location is transmitted. (3) *By name.* The expression must be reevaluated each time the corresponding formal parameter is referenced in the subprogram. A thunk is compiled which evaluates the expression in the environment of the calling program and returns the value of the expression as its value. A pointer to the thunk is transmitted. Assignments to the formal parameter are prohibited.

*Subscripted Variables with Constant Subscripts.* (1) *By value.* The designated data structure element is retrieved and its value transmitted. The formal parameter acts as a simple initialized local variable in the subprogram. (2) *By reference.* The location of the designated data structure element is retrieved and a pointer to this location transmitted. The formal parameter is treated identically as for a simple variable actual parameter. (3) *By name.* This case may be treated as identical to transmission by reference.

*Subscripted Variables with Nonconstant (expression) Subscripts.* (1) *By value.* The subscript expressions are evaluated at the point of call, the designated data structure element is accessed, and its value is transmitted. (2) *By reference.* The subscript expressions are evaluated at the point of call and the designated data structure element is accessed. A pointer to the location of the element is transmitted. The formal parameter is treated identically as for a simple variable actual parameter. (3) *By name.* The subscript expressions must be reevaluated each time the formal parameter is referenced in the called subprogram, and a possibly different data structure element must be accessed. A thunk is compiled which evaluates the subscript expressions in the environment of the calling program and returns a pointer to the appropriate data structure location. Assignment to the formal parameter in the subprogram is

allowed, but the thunk must be evaluated and assignment made to the location returned as its value.

*Data Structures (Other than Simple Variables).* (1) *By value.* When a data structure name, e.g., the name of an array, is given as an actual parameter, the interpretation is problematic. What is the "value" of an array? The interpretation adopted in ALGOL is that the array must be *copied* at the point of call, and a pointer to the copy must be transmitted to the subprogram. More commonly the treatment is identical to that of transmission by reference: A pointer to the designated data structure is transmitted. The advantage of copying the structure (the ALGOL interpretation) is that the original array is protected; assignments to the formal parameter (suitably subscripted) in the subprogram will modify the copy, not the original array. The disadvantage of copying is in the substantial extra storage required. (2) *By reference.* The data structure associated with the actual parameter identifier at the point of call is retrieved, and a pointer to the structure is transmitted. References to the associated formal parameter in the subprogram must be suitably subscripted to allow proper accessing of elements. Assignments to the subscripted formal parameter will modify the actual parameter data structure. Assignments to the formal parameter when unsubscripted are not allowed. (3) *By name.* This case may be treated as transmission by reference.

*Subprograms (Including "By Name" Parameters Compiled into Thunks).* When a subprogram name is used as an actual parameter a special case arises. The distinction between transmission by name, value, or reference essentially disappears: One expects a pointer to the block of executable code for the subprogram to be transmitted, perhaps also with a nonlocal referencing environment specification. The same technique applies when a by name actual parameter is compiled into a thunk. A pointer to the code for the thunk is transmitted, together with a specification of the nonlocal referencing environment in which the thunk is to be executed. In general, a thunk contains *only* nonlocal references, and thus the referencing environment specification is critical. An ordinary subprogram *may* contain nonlocal references. If not, there is no problem in its execution through a reference to the formal parameter; it suffices to simply transmit a pointer to the code. However, if it does contain nonlocal references, then the specification of nonlocal referencing environment is important.

The difficulty with nonlocal references in subprogram parameters is the following: When a reference to the formal parameter causes



execution of the subprogram actual parameter, the appropriate referencing environment is not that which is "natural" at the point of call. Instead it is usually the case that a different nonlocal referencing environment must be set up especially for execution of the subprogram. Where nonlocal environment specification is explicit (as in FORTRAN) there is no difficulty because all nonlocal references are ordinarily global and thus independent of the point of call or transmission. Where the nonlocal environment is implicit, however, reinstating the proper nonlocal environment at the point of call may be complex.

Consider first the most recent association rule for nonlocal environments. Suppose that subprogram $Q$, which contains a nonlocal reference to $X$, is transmitted by subprogram $P$ to subprogram $R$ as a parameter. $R$ then calls $Q$ using the appropriate formal parameter. Suppose also that $P$ and $R$ each have a local association for $X$. When $Q$ references $X$, should it get $P$'s $X$ or $R$'s $X$? In SNOBOL4 it would get $R$'s $X$; in some implementations of LISP it would get $P$'s. Simplicity of implementation would dictate the former. Recall the simple directed search of the stack of environment tables (or the direct look-up in a central environment table) associated with the most recent association rule. Without modification we could apply this same technique during execution of $Q$, always retrieving $R$'s local association if present. The difficulty with adoption of this simple interpretation is that it is rather unnatural for the programmer. Suppose that subprogram $R$ with local variable $X$ is a simple *generator* routine which accepts an array and a subprogram name as its two arguments. $R$ is to apply the subprogram to each element of the array in sequence. The programmer calls $R$ at one point in his program as CALL $R(A,Q)$, and perhaps later as CALL $R(B,Q')$. Suppose that $Q$ and $Q'$ contain nonlocal references to $X$ and $Z$, respectively. Execution of $Q'$ by $R$ proceeds without difficulty: The reference to $Z$ retrieves the association in effect at the point of call of $R$. But during execution of $Q$ each reference to $X$ inadvertently retrieves the association for local variable $X$ in $R$. Since this $X$ is supposedly unknown to the user of $R$, a subtle error has arisen which is difficult to detect. Clearly during execution of $Q$ and $Q'$ by $R$ it is desirable to reinstate the referencing environment that existed at the point where $Q$ or $Q'$ were transmitted to R as parameters. This would ensure that the local environment of $R$ would not interfere with nonlocal referencing in $Q$ and $Q'$.

How is this reinstatement of the referencing environment at the point of call to be simulated? Suppose that the simple stack of local environment tables is being used. When $Q$ is transmitted as a

parameter to $R$ we wish to record its nonlocal referencing environment at the point of transmission and transmit that information as part of the parameter $Q$ itself (along with the usual pointer to the code for $Q$). Then whenever $Q$ is called by $R$ (or other subprograms which $R$ may call) using the associated formal parameter, the appropriate nonlocal environment for $Q$ may be restored before execution of $Q$ is initiated. When $Q$ is transmitted as a parameter the environment existing at the point of transmission may be designated by a simple pointer to the current top of the stack. The actual parameter $Q$ then is represented as a pair: (code pointer, environment pointer). When $Q$ is actually called through the formal parameter the environment pointer for $Q$ is installed to represent the *top of the stack* during execution of $Q$, and the code pointer is used as the point to begin execution.

Let us consider now the static block structure specification for nonlocal environments. Recall that the nonlocal environment in this case depends on the point of definition of a subprogram in the original program during translation. Assume that the static chain mechanism is used to determine nonlocal referencing environments during execution. When $Q$ is transmitted to $R$ as a parameter, its referencing environment at the point of transmission is determined by the static chain appropriate to its point of definition. If we transmit with $Q$ a pointer to the head of this static chain and reinstate this pointer when $Q$ is called in $R$, then the reinstatement of the appropriate nonlocal environment for $Q$ is effectively achieved. Again $Q$ is represented as a pair, (code pointer, environment pointer), during argument transmission, with the environment pointer pointing to the local environment table of the point of definition of $Q$. This table is determined in exactly the same manner as if $Q$ were being called instead of $R$ at the point where $Q$ appears as a parameter.

*Labels.* Statement labels as actual parameters cause difficulties somewhat similar to those caused by subprogram parameters. Again there is no real distinction between transmission by name, value, or reference; one expects a pointer to a code position to be transmitted. In the called subprogram a goto the formal parameter leads to an ambiguity. When control is transferred to the statement labeled by the actual parameter, what is to be the referencing environment in effect (i.e., which activation of the subprogram containing the label is to receive control)? The usual interpretation is that the referencing environment in effect after the goto is to be that which would have resulted had the goto been executed at the point where the label was *transmitted* as a parameter. Thus a label parameter must carry with it



a designation of the referencing environment in effect at the point of transmission, exactly as with a subprogram parameter. The label parameter may be represented as a pair, (code pointer, environment pointer), with the called subprogram reinstating the appropriate environment (using the environment pointer) before executing a goto.

*Example 6-1.* The complexities of nonlocal referencing in the presence of subprogram and label parameters may be best understood by study of an example. Consider the ALGOL program, $B$, of Fig. 6-14 (adapted from a program given by Johnston [1971]). The reader should stop at this point and attempt to determine the result of the program himself before proceeding. Recall that nonlocal referencing in ALGOL is controlled by the static block structure rule.

For the simulation of referencing environments during execution of $B$ we shall adopt the static chain representation of Section 6-6. There are four separate local environments involved, for $B$, $P$, $Q_{top}$ (declared within $P$), and $Q_{bot}$ (declared below $P$). Each local environment may be represented by a local environment table containing one entry for each identifier declared (or used, in the case of labels) within the block or subprogram. These base table formats

```
Line
 1   B: begin integer N;
 2        procedure P(X,C); value C; procedure X; integer C; begin
 3             procedure Q(T); label T; begin
 4                  N := N + C;
 5                  X(K);
 6                  goto T
 7             end proc Q;
 8           J: if C > N then X(J) else P(Q,C+1);
 9           K: N := N + C;
10              goto L
11        end proc P;
12        procedure Q(L); label L; begin
13           N := N + 1;
14           goto L
15        end proc Q;
16          N := 3;
17          P(Q,4);
18      L: print (N)
19    end block B;
```

**Fig. 6-14.** An ALGOL program with label and procedure parameters.

| Static chain pointer | |
|---|---|
| N | |
| P | |
| Q | |
| L | |
| R.P. | |

Local table for B

| Static chain pointer | |
|---|---|
| X | |
| C | |
| Q | |
| J | |
| K | |
| R.P. | |

Local table for P

| Static chain pointer | |
|---|---|
| T | |
| R.P. | |

Local table for $Q_{top}$

| Static chain pointer | |
|---|---|
| L | |
| R.P. | |

Local table for $Q_{bot}$

R.P. = return point location

**Fig. 6-15.** Local environment table formats for program of Fig. 6-14.

are shown in Fig. 6-15. Note that each table has been extended to include, as additional entries, a static chain pointer and a return point location. In this representation, return points for the subprograms may be represented as label parameters, consisting of a pair, (code pointer, environment pointer). Return from a subprogram is equivalent to a **goto** to this label parameter, tagged *R.P.* in the local table. For clarity, identifiers are shown as directly present in the environment tables, although as explained in Section 6-6 this is not necessary in ALGOL; identifiers would ordinarily not be present explicitly.

Execution of $B$ proceeds through the following sequence of subprogram entries and exits:

Step

1. Enter $B$.
2. $B$ calls $P$, setting up activation $P_{init}$ (line 17).
3. $P_{init}$ calls $Q_{bot}$ through formal parameter $X$ (line 8).
4. $Q_{bot}$ returns control to $P_{init}$, statement $J$, through formal parameter $L$ (line 14).
5. $P_{init}$ calls itself recursively, setting up activation $P_{rec}$ (line 8).
6. $P_{rec}$ calls $Q_{top}$ through formal parameter $X$ (line 8).
7. $Q_{top}$ calls $Q_{bot}$ through formal parameter $X$ of $P_{init}$ (line 5).
8. $Q_{bot}$ returns control to $P_{init}$, statement $K$, through formal parameter $L$, also terminating the activations of $P_{rec}$ and $Q_{top}$ (line 14).
9. $P_{init}$ returns control to $B$, statement $L$ (line 10).



At each of these steps the referencing environment must be modified. Figure 6-16 shows the referencing environment effective between each of the steps. Note that whenever a procedure or label is transmitted as a parameter to a new subprogram activation, a pointer to the local table in which the actual parameter reference was satisfied is also transmitted. For example, in step 2 when $Q_{bot}$ is transmitted as an actual parameter to $P$, a pointer to the local table $B$ in which the association for $Q$ was found is also transmitted. In step 3 when $Q_{bot}$ is called by $P$ through the formal parameter $X$, the static chain pointer in the table for $Q_{bot}$ may be set to point to table $B$, thus installing the appropriate nonlocal environment for execution of $B$. The reader is encouraged to determine himself for each step how the next environment may be set up from the previous one, given only the previous environment, the table formats of Fig. 6-15, and the program statement executed. Note that the current local environment at each step is indicated by a special static chain head (SCH) pointer stored in a fixed location. The tree structure representation rather than the explicit stack representation has been used to show more clearly the various referencing environments at each point during execution. Observe that local tables are destroyed strictly in the reverse order of their creation, and thus the tables may be created and destroyed in a single stack.

## 6-10. TRANSMITTING RESULTS BACK FROM SUBPROGRAMS

While our main focus in the preceding sections has been on the problems of local data in subprograms and the transmission of data to subprograms either through parameters or nonlocal environments, there remains the question of transmitting results back from subprograms. Three main methods are used:

1. Function values.
2. Modifiable parameters.
3. Changes in values of nonlocal variables (side effects).

Transmitting results through modification of nonlocal variables presents no particular difficulty. We have implicitly assumed that possibility in the preceding discussion of nonlocal environments. Function values and modifiable parameters require further discussion.

Ptr(n) represents the code pointer to the executable code for the line numbered *n* in Fig. 6-14; other pointers are indicated by arrows

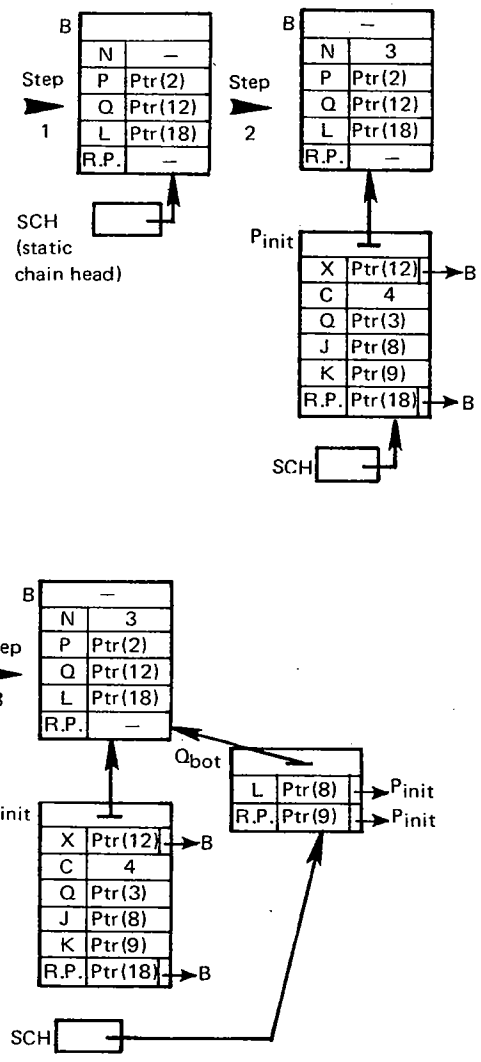



**Fig. 6-16.** Referencing environments during execution of program B (Fig. 6-14).



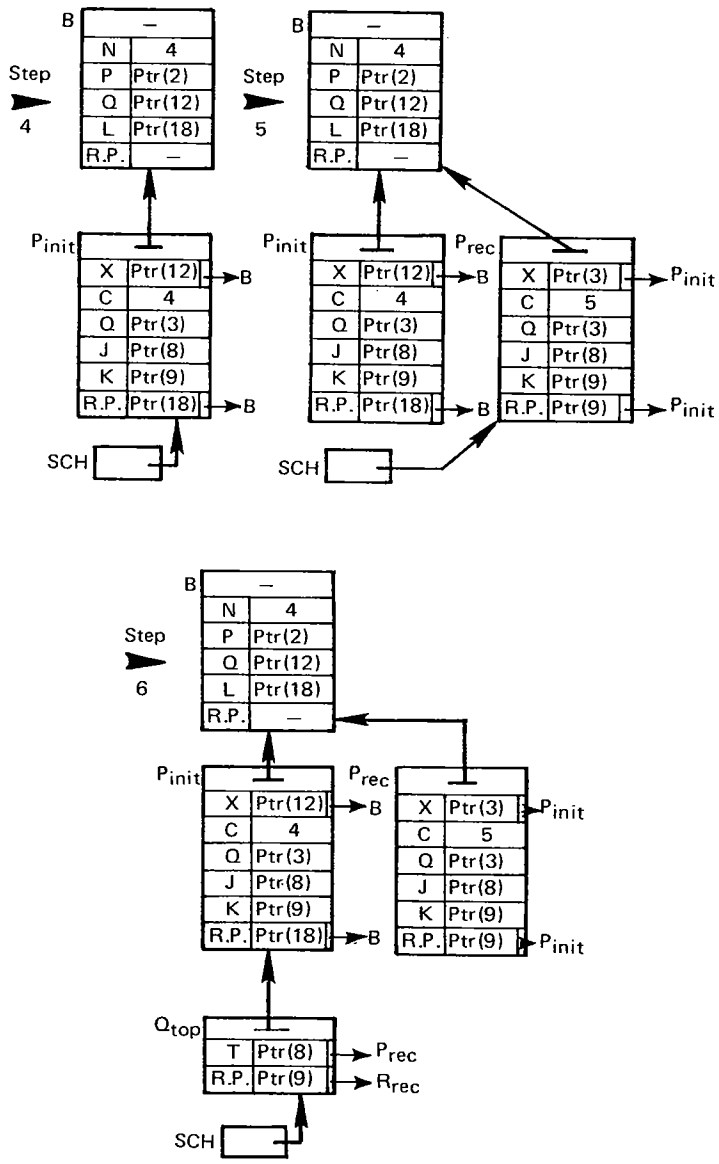
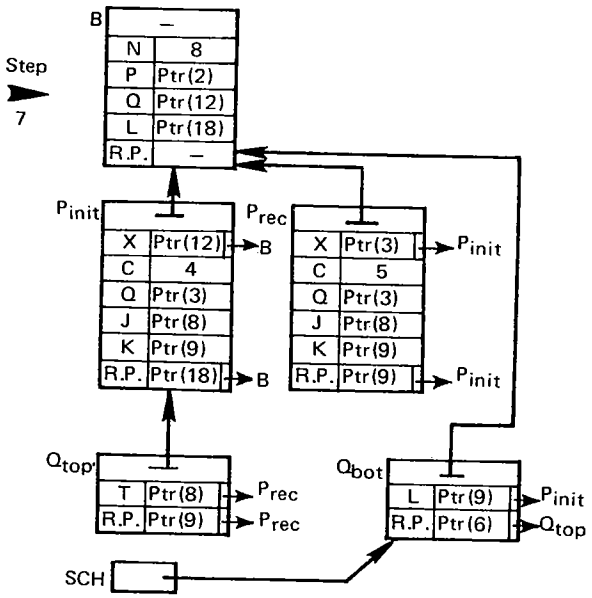



**Fig. 6-16** (continued)



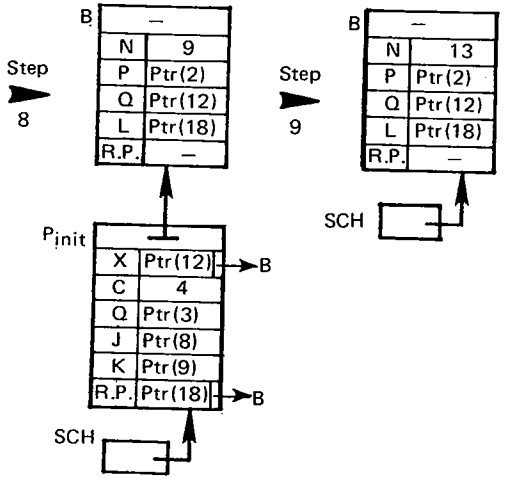




Fig. 6-16 (continued)



### Modifiable Parameters

Parameters transmitted by reference may serve as a way of transmitting results back from a subprogram. This is the usual technique in FORTRAN, for example. Of course, the actual parameter must be a modifiable object, such as a simple or subscripted variable. When an assignment is made in the called subprogram to a formal parameter the result is reflected in a modified value of the corresponding actual parameter.

Parameters transmitted by name, where the actual parameter is a simple or subscripted variable, also allow results to be transmitted back from a subprogram through assignments to the corresponding formal parameter. The mechanism is more complex, however, as the actual parameters in some cases must be treated as subprograms (thunks) which return pointers to locations to which the assignment is then made.

*Result Parameters.* A useful alternative is provided by result parameters. A *result parameter* is a modifiable parameter in a subprogram which is used only for returning results to the calling program. The actual parameter must of course be modifiable, e.g., a simple or subscripted variable. In the called subprogram the formal parameter, tagged as a result parameter, is treated as a simple local variable throughout execution. On termination of the called subprogram, however, the final value of the formal parameter is set as the value of the corresponding actual parameter on return to the calling program.

*Value-Result Parameters.* It is natural to combine parameter transmission by value and result, allowing a parameter to serve both for input to and output from a subprogram. At the time of call the actual parameter is evaluated; the value transmitted becomes the initial value of the formal parameter. During execution of the called subprogram the formal parameter acts as an ordinary local variable. On exit the final value of the formal parameter is stored as the value of the corresponding actual parameter.

Value-result parameters are similar to reference parameters where the actual parameter is a simple or subscripted variable. The difference is that with reference parameters any change in the value of a formal parameter is immediately reflected as a change in the value of the corresponding actual parameter. With value-result parameters the value of the actual parameter changes only on final exit from the called subprogram.

### Function Values

When a subprogram produces only a single result it is common to allow this result to be returned implicitly by treating the subprogram as a function. Often this is done by allowing the subprogram name to be treated as a variable to which assignments may be made during subprogram execution. The value of this *pseudovariable* on exit from the subprogram is returned as its *function value*. Another common technique is to explicitly designate the value to be returned by an expression at each point of exit of the subprogram, as is done in PL/I.

## 6-11. REFERENCES AND SUGGESTIONS FOR FURTHER READING

A central source of material on the problems of data control mechanisms and their associated implementation techniques is the volume edited by Tou and Wegner [1971]. In this volume the paper by Wegner [1971] provides a useful overview; the paper by Johnston [1971] presents the *contour model* for the representation of referencing environments, a model which is particularly useful for the clarification of environments based on static block structure; and the paper by Organick and Cleary [1971] describes a hardware implementation of a number of these concepts. Boyle and Grau [1970] treat static block structure referencing environments more theoretically.

The paper by Bobrow and Wegbreit [1973] provides an implementation structure suitable for handling referencing environments in languages using general sequence control structures such as coroutines. As a rule, in fact, most of the papers referenced in Chapter 5 that are concerned with subprogram sequence control mechanisms such as recursion, coroutines, parallel processes, and scheduled subprograms also discuss at length the related data control questions.

Parameter transmission techniques are often discussed in connection with compilation methods; see, e.g., Gries [1971]. The problems of label and procedure parameters are discussed in many of the papers in Tou and Wegner [1971] and also in papers by Reynolds [1970] and Moses [1970].

## 6-12. PROBLEMS

6-1. For the programs $P$, $Q$, and $R$, with local identifiers as given in Fig. 6-3, assume the following sequence of calls: $P$ calls $R$ which calls $Q$ which calls $R$ (recursively) which calls $P$ (recursively).



a. Show the contents of the central stack after this sequence of calls, as in Fig. 6-5, assuming the central stack simulation.
b. Show the contents of the central table and the hidden stack after this sequence of calls, as in Fig. 6-7.

6-2. Explain why it is *impossible* in a language with only parameters transmitted by value or name, such as ALGOL, to write a subprogram *SWAP* of two parameters which simply swaps the values of its two parameters (which must be simple or subscripted variables). For example, *SWAP* called by *SWAP*($X$,$Y$) should return with $X$ having the original value of $Y$ and $Y$ the original value of $X$. Assume that *SWAP* works only for arguments of type real.

6-3. One variant of the central referencing environment table simulation of Section 6-5 is sometimes used in LISP implementations. Each identifier (atom in LISP) is paired with a separate little stack (usually part of its property list) which contains all the associations for that identifier. The current association is at the top of an identifier's stack; other associations on the stack represent inactive associations. When a subprogram is entered which has a new association for an identifier $X$, the new association is stacked on top of $X$'s association stack. When execution of the subprogram is finished, the top entry on $X$'s stack is simply deleted, restoring the old association.
a. With this technique would a central stack still be needed? If so, what information would it need to contain?
b. Discuss the relative advantages and disadvantages of this technique as compared to the central table technique outlined in Section 6-5.

6-4. The central referencing environment table stimulation of Section 6-5 could not be applied to a language such as ALGOL, where nonlocal referencing is based on the static block structure, as easily as it could to LISP or SNOBOL4, where nonlocal referencing uses the most recent association rule. Why?

6-5. Consider the following ALGOL-like program:

```
begin integer Y;
    procedure P(X); integer X;
        begin X := X + 1; print(X,Y)end proc P;
    Y := 1;
    P(Y);
    print(Y)
end
```

Give the three numbers printed in the case that $Y$ is transmitted to $P$ (a) by value, (b) by reference, and (c) by value-result.

6-6. The following ALGOL-like program appears to do very little:

```
begin integer I; integer array A[1:2];
      procedure F(X,Y); integer X,Y;
         begin    X := X + 1;
                  Y := Y + 1;
                  print(X,Y);
                  X := X - 1;
                  Y := Y - 1;
         end proc F;
      I := 1;
      A[1] := 5;
      A[2] := 10;
      F(I,A[I]);
      print(I,A[1],A[2]);
      F(A[I],I);
      print(I,A[1],A[2])
end
```

Give the ten values printed in each case when parameter transmission is as follows:

a. *X* and *Y* by reference.
b. *X* by reference, *Y* by name.
c. *X* by name, *Y* by reference.
d. *X* and *Y* by name.

**6-7.** What is the result of execution of *B* in example 6-1 if the most recent association rather than the static block structure rule is used to determine the nonlocal referencing environment?

**6-8.** One common restriction on actual parameter lists is that the same simple or subscripted variable cannot appear more than once in the list; e.g., CALL SUB(X,X) is not allowed. Explain the difficulties which such repeated parameters may cause for the programmer when parameter transmission is by value-result, reference, and name.

**6-9.** Suppose that you wished to modify ALGOL so that nonlocal referencing was based on the most recent association rule rather than the static block structure. How would the organization of the run-time central stack of environment tables have to be modified? In particular

a. What information would need to be deleted from each procedure or block local environment table in the stack?
b. What information would need to be added to each local table?
c. Describe how nonlocal referencing would work in this modified structure.

**6-10.** *Jensen's device.* Parameters transmitted by name allow use of a programming trick known as *Jensen's device* (discussed at length in Rutishauser [1967]). The basic idea is to transmit by name as separate parameters to a subprogram both an expression involving one or more variables and the variables themselves. By adroit changes in the values of the variables



coupled with references to the formal parameter corresponding to the expression, the expression may be evaluated for many different values of the variables. A simple example of the technique is in the general-purpose summation routine SUM, defined in ALGOL as follows:

```
real procedure SUM(EXPR,INDEX,LB,UB); value LB,UB;
        real EXPR; integer INDEX,LB,UB;
   begin real TEMP; TEMP := 0;
     for INDEX := LB step 1 until UB do TEMP := TEMP + EXPR;
     SUM := TEMP
   end proc SUM;
```

In this program *EXPR* and *INDEX* are transmitted by name, and *LB* and *UB* by value. The call of *SUM*

$$SUM(A[I],I,1,25)$$

will return the sum of the first 25 elements of vector *A*. The call

$$SUM(A[I] \times B[I],I,1,25)$$

will return the sum of the products of the first 25 corresponding elements of vectors *A* and *B* (assuming that *A* and *B* have been appropriately declared). The call

$$SUM(C[K,2],K,-100,100)$$

will return the sum of the second column of matrix *C* from $C[-100,2]$ to $C[100,2]$.

a. What call to *SUM* would give the sum of the elements on the main diagonal of a matrix *D*, declared as real array $D[1:50,1:50]$?
b. What call to *SUM* would give the sum of the squares of the first 100 odd numbers?
c. Use Jensen's device to write a general-purpose *MAX* routine that will return the maximum value from a set of values obtained by evaluating an arbitrary expression *EXPR* containing an index *INDEX* which varies over a range from *LB* to *UB* in steps of *STEP* size (an integer).
d. Show how the effect of Jensen's device may be obtained by using subprograms as parameters in a language without parameter transmission by name.

**6-11.** How must the central referencing environment table simulation of the most recent association rule for nonlocal environments be modified if local environments for subprograms are retained between calls rather than being created on each entry and destroyed on exit? Assume that no recursion is allowed.

**6-12.** LISP and SNOBOL4 each allow new identifiers to be added to the referencing environment of a subprogram during its execution, as global

identifiers. Each language uses the most recent association rule for nonlocal referencing. Assuming that the central table simulation for referencing environments is used, explain the effect that the introduction of new identifiers has on the structure and use of the central table.

**6-13.** For the ALGOL program of Fig. 6-14, give the run-time representation of each identifier in each subprogram or block as an integer pair $(n, k)$, where $n$ represents the number of tables down the static chain and $k$ represents the offset within that table.

**6-14.** Propose an appropriate method of handling local and nonlocal referencing in *recursive coroutines* (Problem 5-12). Provide arguments as to why your techniques are the appropriate ones.

**6-15.** In LISP it is possible for a subprogram to *construct* a new subprogram during its execution and return the new subprogram as its result. The calling program may then execute the new subprogram with appropriate parameters just as though it had always existed. Suppose that the new subprogram contains a nonlocal reference to $X$. The association retrieved for $X$ during execution of the new subprogram might be

1. The association for $X$ in effect at the point of *call* of the new subprogram, or

2. The association for $X$ in effect at the point of *creation* of the subprogram.

Explain how each of these two choices might be simulated if the central stack method and the most recent association rule are used in LISP for the simulation of referencing.

# 7 STORAGE MANAGEMENT

Storage is one of the scarce resources in any computing system. Storage management is one of the central concerns of the programmer, language implementor, and language designer. In this chapter the various problems and techniques in storage management are considered in their relation to programming language design.

## 7-1. INTRODUCTION

Programming language design is strongly influenced by storage management considerations. Typically languages contain many features or restrictions which may be explained only by a desire on the part of the designers to allow one or another storage management technique to be used. Take, for example, the restriction in FORTRAN to nonrecursive subprogram calls. Recursive calls could be allowed in FORTRAN without change in the syntax, but their implementation would require a run-time stack of return points, a structure necessitating dynamic storage management during execution. Without recursive calls FORTRAN may be implemented with only static storage management. ALGOL is carefully designed to allow stack-based storage management, LISP to allow garbage collection, etc.

Storage management is one of the first concerns of the language implementor as well. While each language design ordinarily permits the use of certain storage management techniques, the details of the mechanisms, and their representation in hardware and software, are the task of the implementor. For example, while the LISP design

may point to a free space list and garbage collection as the appropriate basis for storage management, there are a number of different garbage collection techniques known. The implementor must choose or design one appropriate to the available hardware and software.

The programmer is also deeply concerned with storage management, but his position is somewhat anomalous. While it is of major importance to the programmer to design programs that use storage efficiently, he is likely to have little direct control over storage management. His program affects storage management only indirectly through the use or lack of use of different language features. His position is made more difficult by the tendency of both language designers and language implementors to treat storage management as a *machine-dependent* topic which should not be directly discussed in language manuals. Thus it is often difficult for a programmer to discover what storage management techniques are actually used. The cost of using different language features is often proportional to the cost of the storage management involved, and where the programmer cannot discover what storage management techniques are used he has no rational way to determine the relative costs of various algorithms for the solution of a given problem.

## 7-2. MAJOR RUN-TIME ELEMENTS REQUIRING STORAGE

The programmer tends to view storage management largely in terms of storage of his data structures and translated programs. However, run-time storage management encompasses many other areas besides these. Some, such as return points for subprograms, have been touched on in preceding chapters; others have not yet been mentioned explicitly. Let us look at the major program and data elements requiring storage during program execution.

*Translated User Programs.* A major block of storage in any system must be allocated to store the translated form of user programs, regardless of whether programs are hardware- or software-interpreted. In the former case programs will be blocks of executable machine code; in the latter case programs will be in some intermediate form.

*System Run-Time Programs.* Another substantial block of storage during execution must be allocated to system programs that support the execution of the user programs. These may range from simple



*library routines*, such as sine, cosine, or square root functions, to software interpreters or translators present during execution. Also included here are the routines that control run-time storage management. These system programs would ordinarily be blocks of hardware-executable machine code, regardless of the executable form of user programs.

*User-Defined Data Structures and Constants.* Space for user data must be allocated. This includes mainly data structures declared in or created by user programs, although constants used in programs must also be stored.

*Subprogram Return and (Reentry) Points.* Internally generated sequence control information, such as subprogram return points, coroutine resume points, or event notices for scheduled subprograms, must be allocated storage. As noted in Chapter 5 storage of these data may require only single locations, a central stack, or other run-time storage structure.

*Referencing Environments.* Storage of referencing environments (identifier associations) during execution may require substantial space, as, for example, the LISP A-list (Chapter 14). Some languages, such as FORTRAN, require little or no storage for this purpose.

*Temporaries in Expression Evaluation.* Expression evaluation requires use of system-defined temporary storage for the intermediate results of evaluation. For example, in evaluation of the expression X × Y + U × V the result of the first multiplication must be stored in a temporary while the second multiplication is performed. When expressions may involve recursive function calls, a potentially unlimited number of temporaries may be required to store partial results at each level of recursion.

*Temporaries in Parameter Transmission.* When a subprogram is called, a list of actual parameters must be evaluated and the resulting values stored in temporary storage until evaluation of the entire list is complete. Where evaluation of one parameter may require evaluation of recursive function calls a potentially unlimited amount of storage may be required, as in expression evaluation.

*Input-Output Buffers.* Ordinarily input and output operations work through buffers which serve as temporary storage areas where data are stored between the time of the actual physical transfer of the data to or from external storage and the program-initiated input and output operations. Often hundreds (or thousands) of memory locations must be reserved for these buffers during execution.

*Miscellaneous System Data.* In almost every language implementation, storage is required for various system data: tables, status information for input-output, and various miscellaneous pieces of state information (e.g., reference counts or garbage collection bits).

From this list it is clear that storage management concerns storage for much more than simply user programs and data. More importantly, much of the information requiring storage is hidden from the language user.

## 7-3. PROGRAMMER- AND SYSTEM-CONTROLLED STORAGE MANAGEMENT

To what extent should the programmer be allowed to directly control storage management? On the one hand, PL/I allows some direct control by the programmer through statements such as ALLOCATE and FREE, which allocate and free storage for programmer-defined data structures. On the other hand, many, if not most, high-level languages allow the programmer no direct control. Storage management is affected only implicitly through the use of various language features.

The difficulty with programmer control of storage management is twofold: It may place a large and often undesirable burden on the programmer, and it may also interfere with the necessary system-controlled storage management. No high-level language can allow the programmer to shoulder the entire storage management burden. For example, the programmer can hardly be expected to concern himself with storage for temporaries, subprogram return points, or other system data. At best he might control storage management for his data (and perhaps programs). Yet even simple allocation and freeing of storage for data structures, as in PL/I, is likely to permit generation of garbage and dangling references. Thus programmer-controlled storage management is "dangerous" to the programmer because it may lead to subtle errors or loss of access to available storage. Programmer-controlled storage management also may interfere with system-controlled storage management, in that special storage areas and storage management routines may be required for programmer-controlled storage, allowing less efficient use of storage overall.

The advantage of allowing programmer control of storage management lies in the fact that it is often extremely difficult for the system to determine when storage may be most effectively allocated and freed. The programmer, on the other hand, often knows quite



precisely when a particular data structure is needed or when it is no longer needed and may be freed.

## 7-4. STORAGE MANAGEMENT PHASES: INITIAL ALLOCATION, RECOVERY, COMPACTION, AND REUSE

It is convenient to identify three basic aspects of storage management:

1. *Initial allocation.* At the start of execution each piece of storage may be either already allocated for some use or free. If free initially, it is available to be allocated dynamically as execution proceeds. Any storage management system requires some technique for keeping track of free storage as well as mechanisms for allocation of free storage as the need arises during execution.

2. *Recovery.* Storage which has been allocated and used for awhile and which subsequently becomes available must be recovered by the storage manager for reuse. Recovery may be very simple, as in the repositioning of a stack pointer, or very complex, as in garbage collection.

3. *Compaction and reuse.* Storage recovered may be immediately ready for reuse, or compaction may be necessary to construct large blocks of free storage from small pieces. Reuse of storage ordinarily involves the same mechanisms as initial allocation.

Many different storage management techniques are known and in use in language implementations. It is impossible to survey them all, but a relative handful suffice to represent the basic approaches. Most techniques are variants of one of these basic methods.

## 7-5. STATIC STORAGE MANAGEMENT

The simplest form of allocation is *static allocation*, that is, allocation during translation which remains fixed throughout execution. Ordinarily storage for user and system programs is allocated statically, as is storage for I/O buffers and various miscellaneous system data. Static allocation requires no run-time storage management software, and, of course, there is no concern for recovery and reuse.

In the usual FORTRAN implementation all storage is allocated

statically. Each subprogram is compiled separately, with the compiler setting up a block of storage containing the compiled program, its data areas, temporaries, return point location, and miscellaneous items of system data. The loader allocates space in memory for these compiled blocks at load time, as well as space for system run-time routines. During program execution no storage management takes place.

Static storage allocation is efficient, because no time or space is expended for storage management during execution. However, it is incompatible with recursive subprogram calls, with data structures whose size is dependent on computed or input data, and with many other desirable language features. In the remaining sections of this chapter we shall discuss various techniques for *dynamic* (run-time) *storage management*. However, the reader should not lose sight of the importance of the static allocation method—for many programs static allocation is quite satisfactory. Two of the most widely used programming languages, FORTRAN and COBOL, are designed for strictly static storage allocation.

## 7-6. STACK-BASED STORAGE MANAGEMENT

The simplest run-time storage management technique is the stack-based technique. Free storage at the start of execution is set up as a sequential block in memory. As storage is allocated it is taken from sequential locations in this stack beginning at one end. Storage must be freed in the reverse order of allocation, so that a block of storage being freed is always at the top of the stack. This organization makes trivial the problems of storage recovery, compaction, and reuse.

A single *stack pointer* is basically all that is needed to control storage management. The stack pointer always points to the next available word of free storage in the stack block, representing the current top of the stack. All storage in use lies in the stack below the location pointed to by the stack pointer. All free storage lies above the pointer. When a block of $k$ locations is to be allocated the pointer is simply moved to point $k$ locations farther up the stack area. When a block of $k$ locations is freed, the pointer is moved back $k$ locations. There are no problems of compaction; compaction is automatic.

It is the strictly nested last-in—first-out structure of subprogram calls and returns in most languages that makes stack storage management an appealing technique. Many of the program and data



elements requiring storage are tied to subprogram activations. Stack storage is often appropriate for both subprogram return points and local environment tables for subprograms, as discussed in previous chapters. In addition, temporaries for expression evaluation and parameter transmission may ordinarily be allocated on the basis of subprogram activations. The translator usually can determine the number of temporaries needed for each activation of a subprogram. During execution temporaries may be allocated in a stack as each new subprogram activation is initiated and deleted on termination. Also important in many languages, e.g., ALGOL, is the restriction that programmer data structures may be created only on subprogram entry and must necessarily be destroyed on exit. This organization allows stack allocation for programmer data structures as well.

Grouping those elements associated with a subprogram activation which require stack allocation into a single *activation record* is a common technique. When a subprogram is called, a new activation record is created on the top of the stack. Termination causes its deletion from the stack.

Most ALGOL implementations are built around a single central stack of activation records for subprograms (and blocks), together with a statically allocated area containing user- and system-executable programs, input-output buffers, and constants. The structure of a typical activation record for an ALGOL subprogram is shown in Fig. 7-1. The activation record contains all the variable items of information associated with a given subprogram activation. Figure 7-2 shows a typical memory organization during ALGOL execution.

The use of a stack in a LISP implementation is somewhat different. Here also subprogram (function) calls are strictly nested and a stack may be used for activation records. Each activation record contains a return point and temporaries for expression evaluation and parameter transmission. Local referencing environments (A-list entries) might also be allocated in the same stack, except that the programmer is allowed to directly manipulate these associations. Therefore they are ordinarily stored in a separate stack, represented as a linked list, called the A-list. The stack containing return points and temporaries may then be hidden from the programmer and allocated sequentially. LISP implementation requires also a *heap* storage area which is managed through a free space list and garbage collection, with a special area and storage manager for *full-word* data items such as numbers. A typical LISP memory organization is illustrated in Fig. 7-3.

The use of a stack for subprogram activation records (or partial activation records as in LISP) is characteristic of implementation of

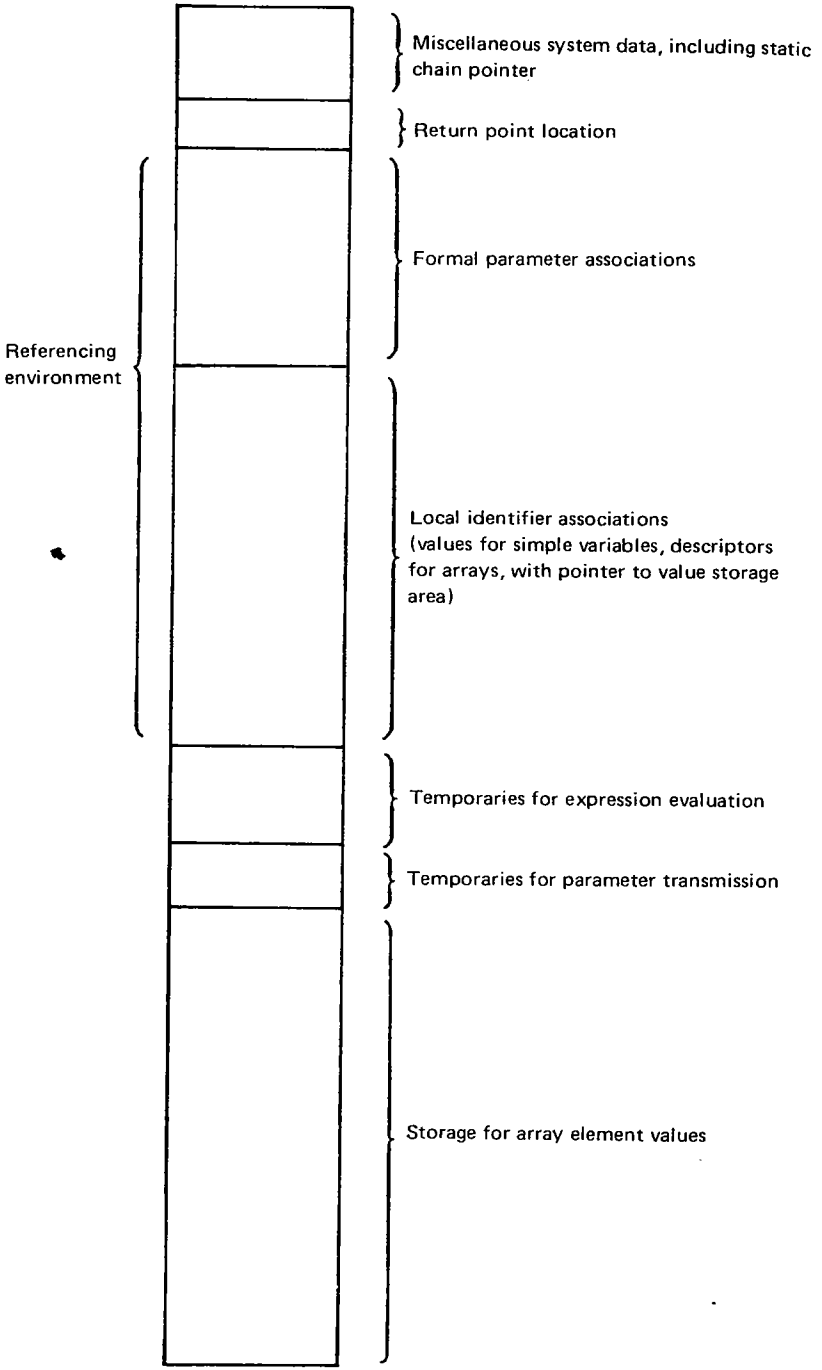



**Fig. 7-1.** Structure of a typical ALGOL subprogram activation record.





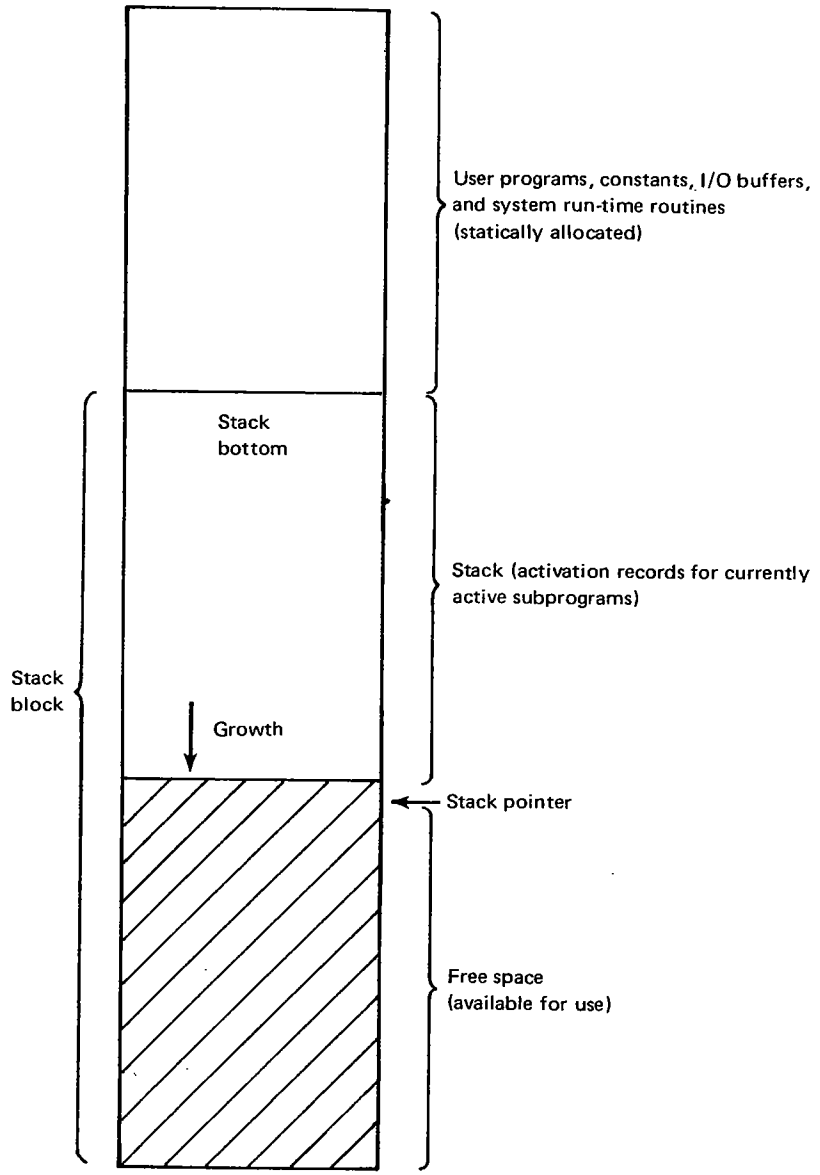



**Fig. 7-2.** Typical memory organization during ALGOL execution.

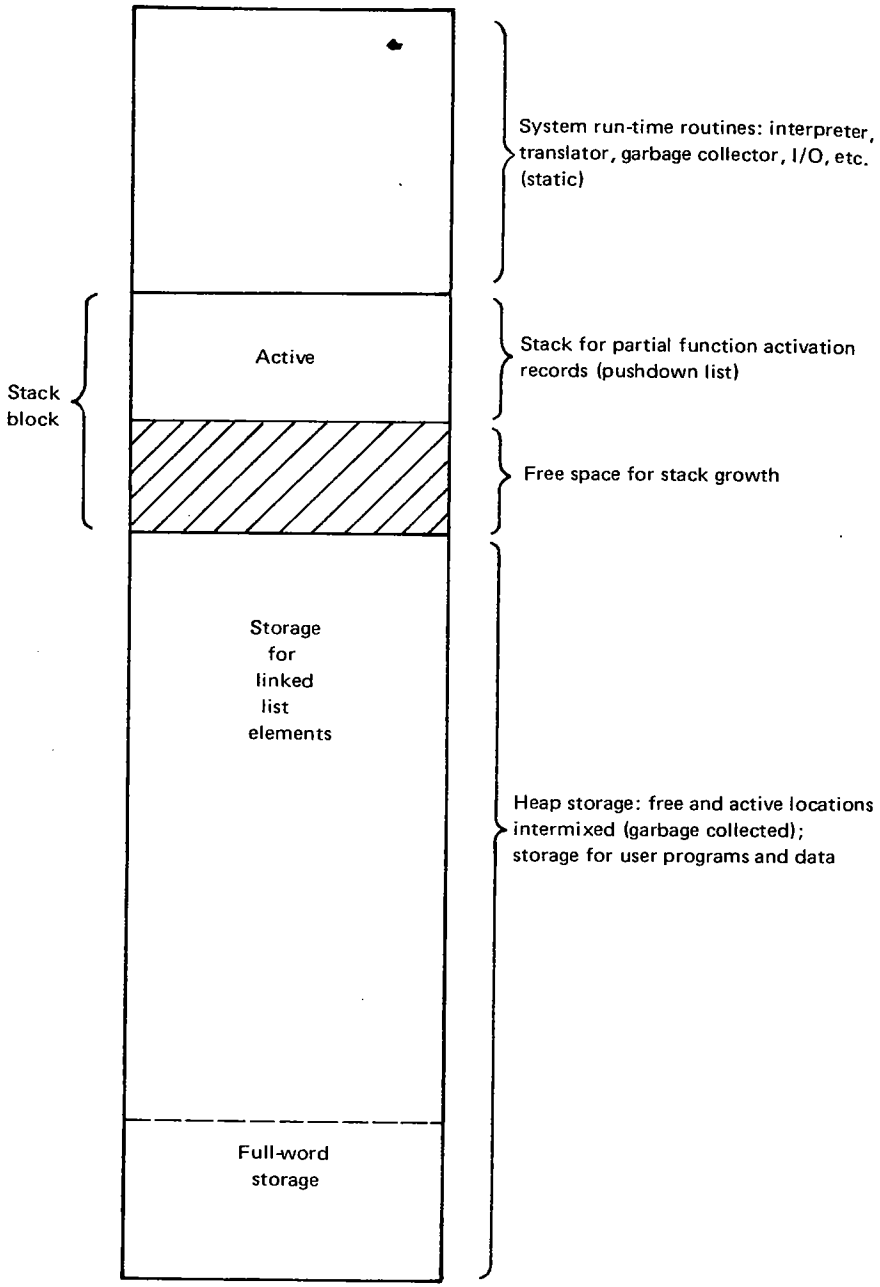



**Fig. 7-3.** Typical LISP memory organization during execution.



every language in Part II except FORTRAN and COBOL. Stack-based storage management is undoubtedly the most widely used technique for run-time storage management.

## 7-7. HEAP STORAGE MANAGEMENT: FIXED-SIZE ELEMENTS

The third basic type of storage management, besides static and stack-based management, may be termed *heap storage management*. A *heap* is a block of storage within which pieces are allocated and freed in some relatively unstructured manner. Here the problems of storage allocation, recovery, compaction, and reuse may be severe. There is no single heap storage management technique, but rather a collection of techniques for handling various aspects of heap storage management. We shall survey the basic techniques here, without attempting to be comprehensive.

The need for heap storage and storage management arises when a language requires storage to be allocated and freed at arbitrary points during program execution, as when a language allows creation, destruction, or extension of programmer data structures at arbitrary program points. For example, in SNOBOL4 two strings may be concatenated to create a new string at any arbitrary point during execution. Storage must be allocated for the new string at the time it is created. In LISP a new element may be added to an existing list structure at any point, again requiring storage to be allocated. In both SNOBOL4 and LISP, storage may also be freed at unpredictable points during execution.

It is convenient to divide heap storage management techniques into two categories depending on whether the elements allocated are always of the same fixed size or of variable size. Where fixed-size elements are used, management techniques may be considerably simplified. Compaction, in particular, is not a problem. We shall consider the fixed-size case in this section, leaving the variable-size case until the following section.

Assume that the fixed-size elements which are allocated from the heap and later recovered occupy $N$ words of memory each. Typically $N$ might be 1 or 2. Assuming the heap occupies a contiguous block of memory, we conceptually divide the heap block into a sequence of $K$ elements, each $N$ words long, where $K \times N$ = length of the heap block. This division of the heap into $K$ fixed-size elements forms the basis for our heap storage management. Whenever an element is needed one of these is allocated from the heap. Whenever an element is freed it must be one of these original heap elements.

### Initial Allocation and Reuse: Free Space Lists

Initial allocation from the heap might be accomplished with a simple *heap pointer*, similar to the stack pointer used earlier, which points to the next available free element in the heap at all times. Each time a new element is needed the heap pointer is advanced to point to the next heap element, and a pointer to the allocated element is returned. Note, however, that such a heap pointer cannot be of any help when an element is freed, because the element will in general lie at some arbitrary point back in the allocated part of the heap. Thus we are constrained to *advance* the heap pointer during allocation, using some other mechanism to keep track of storage which has been freed. Eventually the heap pointer reaches the end of the heap storage block. At that point we must begin to reuse storage back in the heap block which has been freed, moving to some new allocation technique (or compacting the free space at the end of the heap block and resetting the heap pointer, but this is seldom done with fixed-size elements).

How are we to keep track of the various elements back in the heap which have been freed? In general, these free elements will be scattered in some random pattern throughout the heap, intermixed with elements currently in use. The common technique is to maintain a linked list of these free elements, termed a *free space list*. A fixed storage location outside the heap is chosen as the head of the free space list. This location at all times contains a pointer to some free element in the heap (assuming that there is at least one). A pointer within that free element points to another free element, which links to yet another, etc.

Allocation of elements from such a free space list is simple. When an element is needed the list head location is accessed. The pointer there points to a free element which may be deleted from the list by taking the pointer it contains (which points to the second free element on the list) and storing this pointer in the list head, making this second element the new first list element. The deleted element is now available for use (Fig. 7-4).

When an element in use is freed the inverse process adds it to the head of the free space list. The pointer in the list head is stored in the newly freed element, and a pointer to this new element is stored in the list head (Fig. 7-4).

Because a free space list must be maintained anyway, it is often convenient to eliminate the heap pointer used for initial heap allocation and instead use the free space list as the source of elements

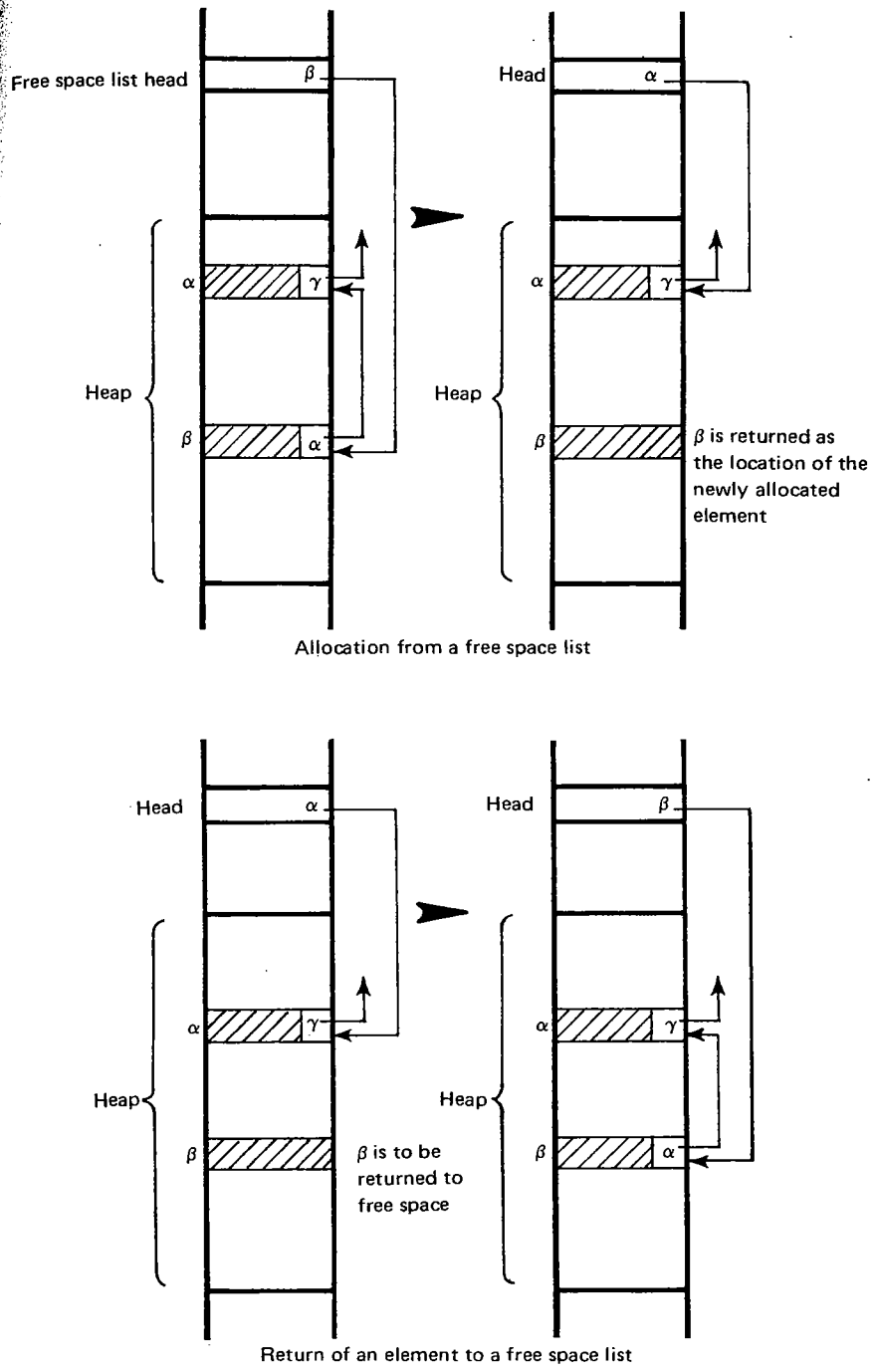



**Fig. 7-4.** Allocation and return with a free space list.

throughout execution. To accomplish this result, all the elements in the entire heap block must be chained together into an initial free space list at the start of execution. Figure 7-5 illustrates such an initial free space list as well as the list after allocation and freeing of a number of elements.

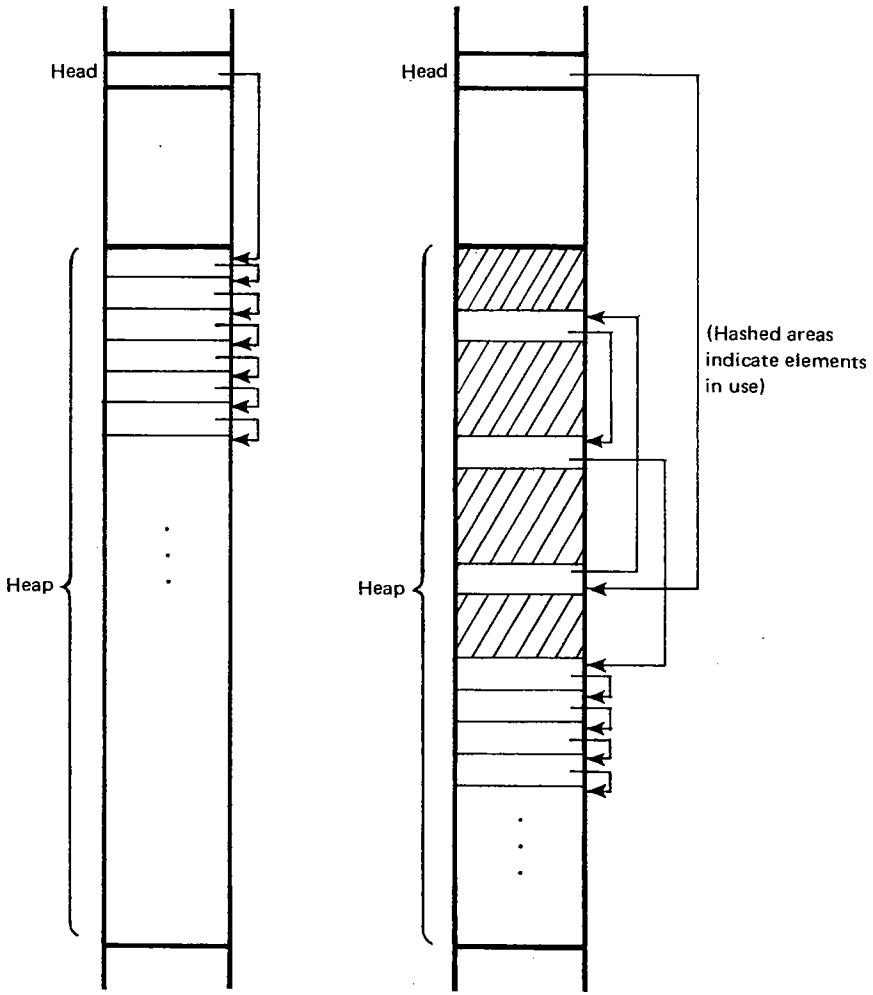



**Fig. 7-5.** Free space list structure .



### Recovery: Explicit Return, Reference Counts, and Garbage Collection

Return of newly freed storage to the free space list is simple, provided such storage may be identified and recovered. But identification and recovery may be quite difficult. The problem lies in determining which elements in the heap are available for reuse and therefore may be returned to the free space list. Three solutions are in fairly wide use.

*Explicit Return by Programmer or System.* The simplest recovery technique is that of *explicit return*. When an element which has been in use becomes available for reuse it must be explicitly identified as "free" and returned to the free space list. Where the space has been used for a programmer-defined data structure the programmer is provided with a FREE or ERASE command which is used to explicitly designate data structures to be returned. Where elements are used for system purposes, such as storage of referencing environments, return points, or temporaries, or where all storage management is system-controlled, each system routine is responsible for returning space as it becomes available for reuse, through explicit call of a FREE routine with the appropriate element as a parameter.

Explicit return would seem the natural recovery technique for heap storage, but unfortunately it is not always feasible. The reasons lie with two old problems: *garbage* and *dangling references*. We first discussed these problems in Chapter 4 in connection with destruction of data structures. If a structure is destroyed (and the storage freed) before all access paths to the structure have been destroyed, the remaining access paths become dangling references. On the other hand, if the last access path to a structure is destroyed without the structure itself being destroyed and the storage recovered, then the structure becomes garbage. In the context of heap storage management a dangling reference is a pointer to an element that has been returned to the free space list (or a pointer to an element that has been returned and later reallocated for another purpose). A garbage element is one that is available for reuse but not on the free space list and which thus is inaccessible.

Both garbage and dangling references are potentially troublesome for a storage management system. If garbage accumulates, available storage is gradually reduced until the program may be unable to continue for lack of known free space. Dangling references may cause chaos. If a program attempts to modify through a dangling

reference a structure that has already been destroyed, the contents of an element on the free space list may be modified inadvertently. If this modification overwrites the pointer linking the element to the next free space list element, the entire remainder of the free space list may become garbage. Even worse, a later attempt by the storage allocator to use the pointer in the overwritten element leads to completely unpredictable results; e.g., a piece of an executable program may be allocated as "free space" and later modified. Similar sorts of problems arise if the element pointed to by the dangling reference has already been reallocated to another use before a reference is made.

Recovery of heap storage by explicit return often leads to the potential to create garbage and dangling references. For example, consider the PL/I statements

```
ALLOCATE ELEM SET(P)     (allocates an element from
                          free space and sets variable P
                          to contain a pointer to it)
P = Q                    (destroys the only pointer to
                          the element, leaving it as
                          garbage)
```

or

```
ALLOCATE ELEM SET(P)
Q = P                    (copies the pointer in P into Q)
FREE P -> ELEM           (destroys the element
                          pointed to by P, freeing the
                          storage for reuse; the
                          pointer in Q is not destroyed,
                          however, leaving a dangling
                          reference)
```

It is easy in such cases for the programmer inadvertently to create garbage or dangling references, with the resulting sometimes dire consequences.

It may be equally difficult for the run-time system to avoid creating garbage or dangling references. For example, in LISP, linked lists are a basic data structure. One of the primitive LISP operations is CDR, which, given a pointer to one element on a linked list, returns a pointer to the next element in the list (see Fig. 7-6). The element originally pointed to *may* have been freed by the CDR operation, provided the original pointer given CDR was the only pointer to the



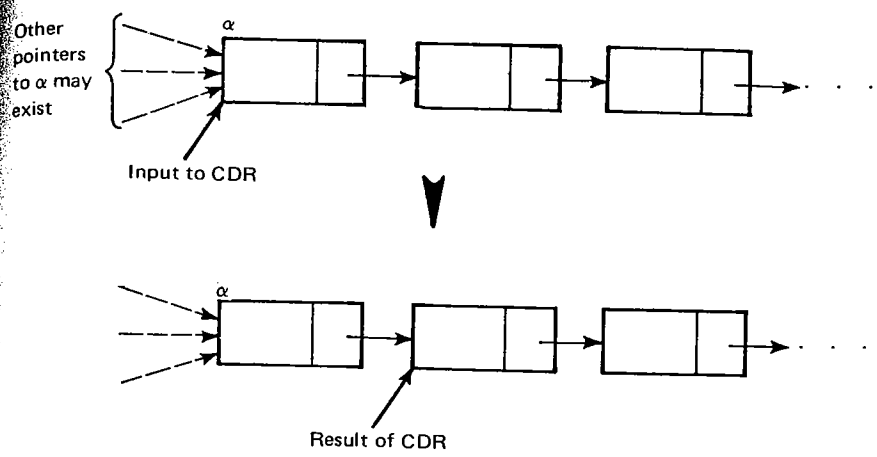



**Fig. 7-6.** The LISP CDR operation.

element. If CDR does not return the element to the free space list at this point, it becomes garbage. However, if CDR does return the element to free space and other pointers to it exist, then they become dangling references. If there is no direct way to determine whether such pointers exist, then the CDR primitive must potentially generate garbage or dangling references.

Owing to these problems with explicit return, alternative approaches are desirable. One alternative, called *garbage collection*, is to allow garbage to be created but no dangling references. Later if the free space list becomes exhausted, a *garbage collector* mechanism is invoked to identify and recover the garbage. A second alternative, that of *reference counts*, requires explicit return but provides a way of checking the number of pointers to a given element so that no dangling references are created.

*Reference Counts.* The use of reference counts is the simpler of the two techniques, so we shall take it up first. The basic concept is this: Within each element in the heap allow some extra space for a *reference counter*. The reference counter of an element contains at all times an integer, the *reference count*, indicating the number of pointers to that element which exist. When an element is initially allocated from the free space list its reference count is set to 1. Each time a new pointer to the element is created its reference count is increased by 1. Each time a pointer is destroyed the reference count

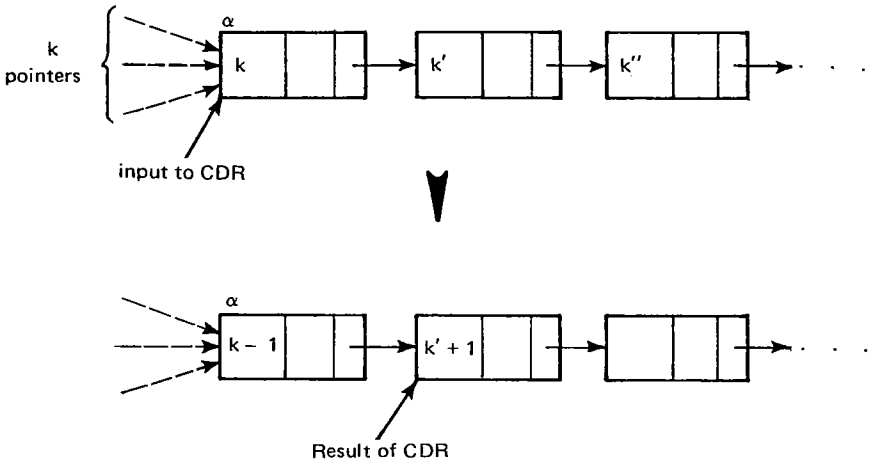



**Fig. 7-7.** The LISP CDR operation with reference counts.

is decreased by 1. When the reference count of an element reaches zero the element is free and may be returned to the free space list.

Reference counts allow both garbage and dangling references to be avoided in most situations. Consider the LISP CDR operation again (Fig. 7-6). If each list element contains a reference count, then it is simple for the CDR operation to avoid the previous difficulties. CDR must subtract 1 from the reference count of the element originally pointed to by its input. If the result leaves a reference count of zero, then the element may be returned to the free space list, and if nonzero, then the element is still pointed to by other pointers and cannot be considered free (see Fig. 7-7).

Where the programmer is allowed an explicit FREE or ERASE statement, reference counts also provide protection. The result of a FREE statement is only to decrement the reference count of the structure by 1. Only if the count then is zero is the structure actually returned to the free space list. A nonzero reference count indicates that the structure is still accessible and that the FREE command should be ignored.

The reference count technique fails in the case of circularly linked groups of elements. Consider a simple circular linked list as in Fig. 7-8, with one pointer to a list element from outside. If this pointer is destroyed, the reference count of the list element pointed to becomes 1, yet as this pointer was the only path to the structure from outside,



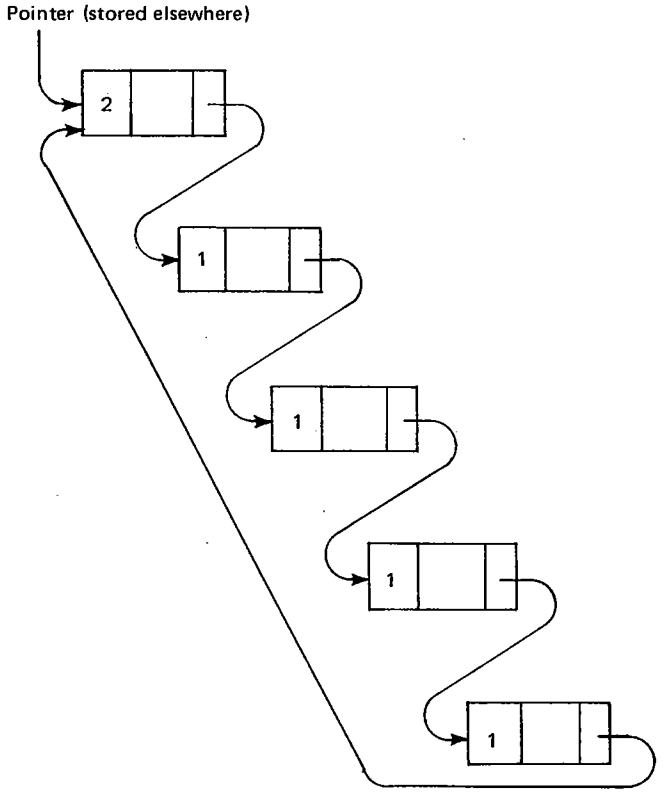



**Fig. 7-8.** Circular list with reference counts.

the structure as a whole has become inaccessible and thus garbage. Unfortunately, none of the list elements has a reference count of zero so no recovery is possible.

The difficulty with circular structures is not the most important difficulty associated with reference counts, however. In many cases circular structures are not allowed or occur so infrequently that the potential for garbage generation is not serious. More telling is the *cost of maintaining* the reference counts. Reference count testing, incrementing, and decrementing must go on continuously throughout execution, often causing a substantial decrease in execution efficiency. Consider, for example, the simple assignment P := Q, where P and Q are both pointer variables. Without reference counts it suffices to simply copy the pointer in Q into P. With reference counts we must do the following:

1. Access the element pointed to by P and decrement its reference count by 1. Test the resulting count, and if zero, return the element to the free space list.

2. Copy the pointer in Q into P.

3. Access the element pointed to by Q and increment its reference count by 1.

The total cost of the assignment operation has been increased substantially. Any similar operation which may create or destroy pointers must modify reference counts also. In addition, there is the cost of the extra storage for the reference counts. If extra space exists in heap elements already, this storage may be no problem. More commonly an extra location would be necessary in each element to contain the reference count. Where elements are only one or two locations in length to begin with, storage of reference counts may substantially reduce the storage available for data.

The cost of maintaining reference counts may be reduced sharply in many cases by restricting pointer use so that only certain elements may have reference counts other than 1 (or some other constant). In such a case reference counts need be maintained only for those special elements with variable reference counts. Consider, for example, a list-processing system in which each linked list is considered a separate data structure. If we provide each list with a special *header element* and allow pointers from other lists to point only to this header and never to any internal list element, then only the header element needs a reference count. The internal list elements always have a constant 1 reference count (in the case of a singly linked list, or 2 in the case of a doubly linked list). Moreover, the header reference count need only count pointers from outside the structure. This is illustrated in Fig. 7-9. Individual elements may be deleted from such a list and returned immediately to the free storage list without concern for reference counts. When the reference count of the header is reduced to zero (when a pointer stored in another list is destroyed, for example) then the entire list—header and all the elements—may be returned to the free list. Care must be taken, however, that each element of the freed list is checked for pointers to other lists. Whenever such a pointer is found the reference count of the list pointed to must be decremented and tested for zero as well.

As one restricts the use of reference counts to larger structures the cost in both storage and processing time decreases, but there is a corresponding loss in flexibility in the manner in which structures



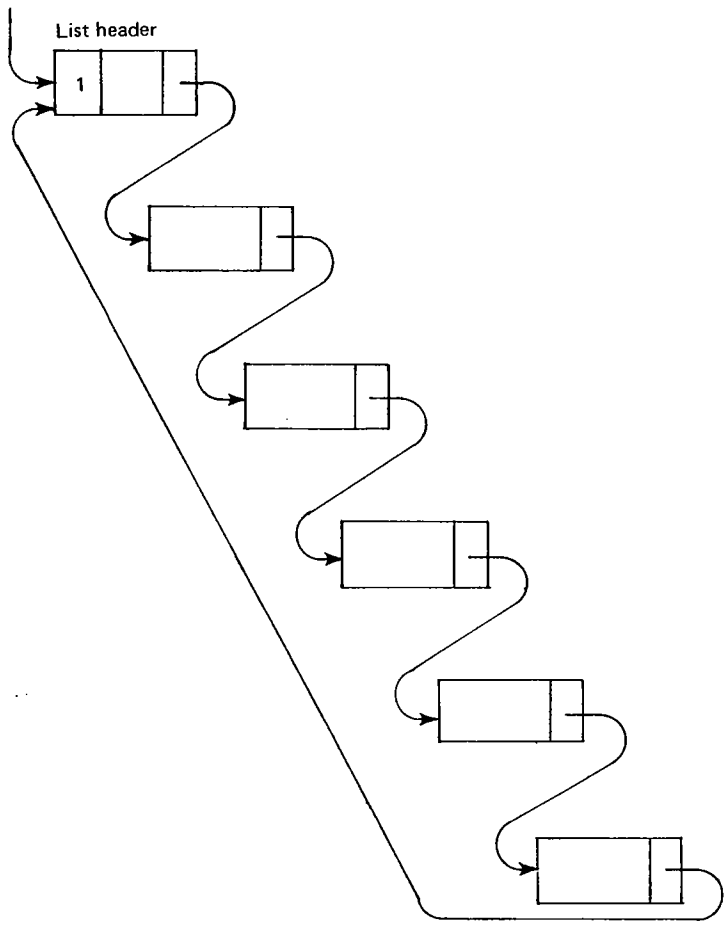



**Fig. 7-9.** Circular list with special header.

may be linked (because linking must be done through the header). Note also that the problem of circular linkages *within* a structure disappears; i.e., a single structure may contain circular linkages. Only circular linkages *between* structures cause difficulty.

*Garbage Collection.* Returning to the basic problems of garbage and dangling references, we may readily agree that dangling references are potentially far more damaging than garbage. Garbage accumulation causes a drain on the amount of usable storage, but dangling references may lead to complete chaos because of random

modification of storage in use. Of course, the two problems are related: Dangling references result when storage is freed "too soon," and garbage when storage is not freed until "too late." Where it is infeasible or too costly to avoid both problems simultaneously through a mechanism such as reference counts, garbage generation is clearly to be preferred in order to avoid dangling references. It is better not to recover storage at all than to recover it too soon.

The basic philosophy behind garbage collection is simply to allow garbage to be generated in order to avoid dangling references. When the free space list is entirely exhausted and more storage is needed, the computation is suspended temporarily and an extraordinary procedure instituted, a *garbage collection*, which identifies garbage elements in the heap and returns them to the free space list. The original computation is then resumed, and garbage again accumulates until the free space list is exhausted, at which time another garbage collection is initiated, etc.

Because garbage collection is done only rarely (when the free space list is exhausted), it is allowable for the procedure to be fairly costly. Two stages are involved:

1. *Marking active elements.* In the first stage each element in the heap which is active, i.e., which is part of an accessible data structure, must be marked. Each element must contain a *garbage collection bit* set initially to "on." The marking algorithm sets the garbage collection bit of each active element "off."

2. *Collecting garbage elements.* Once the marking algorithm has marked active elements, all those remaining whose garbage collection bit is "on" are garbage and may be returned to the free space list. A simple sequential scan of the heap is sufficient. The garbage collection bit of each element is checked as it is encountered in the scan. If "off," the element is passed over; if "on," the element is linked into the free space list. All garbage collection bits are reset to "on" during the scan (to prepare for a later garbage collection).

The marking part of garbage collection is the most difficult. Since the free space list is exhausted when garbage collection is initiated, each element in the heap is either active (i.e., still in use) or garbage. Unfortunately, inspection of an element cannot indicate its status, because there is nothing intrinsic to a garbage element to indicate that it is garbage. Moreover, the presence of a pointer to an element from another heap element does not necessarily indicate that the element pointed to is active; it may be that both elements are



garbage (recall the circular list of Fig. 7-8). Thus a simple scan of the heap which looks for pointers and marks the elements pointed to as active does not suffice.

When is a heap element active? Clearly, an element is active if there is a pointer to it from *outside the heap* or in another *active* heap element. If it is possible to identify all such outside pointers and mark the appropriate heap elements, then an iterative marking process may be initiated which searches these active elements for pointers to other unmarked elements. These new elements are then marked and searched for other pointers, etc. A fairly disciplined use of pointers is necessary because three critical assumptions underlie this marking process:

1. Any active element must be reachable by a chain of pointers beginning outside the heap.

2. It must be possible to identify every pointer outside the heap which points to an element inside the heap.

3. It must be possible to identify within any active heap element the fields which contain pointers to other heap elements.

If any of these assumptions are unsatisfied, then the marking process will fail to mark some active elements. The result will be recovery of active elements and thus the generation of dangling references.

The manner in which these assumptions are satisfied in a typical LISP implementation is instructive. First each heap element is formatted identically, usually with two pointer fields and a set of extra bits for system data (including a garbage collection bit). Since each heap element contains exactly two pointers, and these pointers are always in the same positions within the element, assumption 3 is satisfied. Second, there is only a small set of *system data structures* which may contain pointers into the heap (the A-list, the OB-list, the pushdown list, etc.). Marking starting from these system data structures is guaranteed to allow identification of all external pointers into the heap, as required by assumption 2. Finally it is impossible to reach a heap element other than through a chain of pointers beginning outside the heap. For example, a pointer to a heap element cannot be computed by addition of a constant to another pointer. Thus, assumption 1 is satisfied.

Satisfying the assumptions necessary for garbage collection may be difficult. Consider assumption 3. It requires that every heap element have the same format, that the position of pointers within an element be tagged, that a format designator be stored in each

element, or that the marking algorithm "know" where the pointers are in any element it reaches, using some external rules about the structure of data and pointer chains. Such special requirements for garbage collection place an extra burden on the designer of a language implementation in addition to those imposed directly by the language design.

Given that the assumptions above are satisfied, how is the actual marking to be done? The basic algorithm is obvious: Begin with the pointers outside the heap and exhaustively follow chains of pointers through the heap until every active element has been marked. An algorithm with a temporary stack might be used:

1. Enter in the stack all the external pointers to heap elements. Before each pointer is stacked, test if the element to which it points is marked. If already marked, then the pointer is a duplicate and need not be stacked. If not marked, mark it.

2. Take the top stack pointer. Pop it off the stack and stack all pointers to unmarked heap elements contained in the element to which it points. As each new pointer is added to the stack, mark the element to which it points.

3. Repeat step 2 until the stack is empty.

This procedure is straightforward but immediately raises another problem: Where is space for the stack to be found? Recall that garbage collection was initiated because we were out of known free space in the heap. We might have a separate storage area for the stack, but then this space would be lost for other purposes. Moreover, this stack area would have to be rather large if we wished to ensure that garbage collection could always be completed without overflowing the stack (see Problem 7-4). As a result a stack-based marking algorithm is not entirely appropriate, although conceptually simple.

An elegant alternative is an algorithm given by Schorr and Waite [1967]. Beginning with an external pointer into the heap a pointer chain is traversed to the end. As each pointer is traversed the element reached is marked and the pointer *reversed*. When the end of a chain is reached the reversed pointers allow traversal back out of the chain. In the process, side chains are traversed in a similar manner. The algorithm requires two traversals of each pointer chain (one in each direction), but only two extra registers (rather than a stack) are needed for temporary storage. In addition, each element must have space for an extra *tag field* big enough to hold an integer equal to the length of an element.



In more detail the Schorr and Waite algorithm (slightly modified) may be described as follows. Assume for simplicity that each pointer in an active heap element is contained in a separate location within the element. (Problem 7-5 treats the case of more than one pointer per location within an element.) All heap elements have a garbage collection bit set "on" initially. The two extra registers are designated CE (for *current element*) and LE (for *last element*). Initially CE is set to contain one of the external pointers to a heap element and LE is set to NIL. Marking proceeds as follows:

1. *Mark the current element.* Test the garbage collection bit of the current element, the element pointed to by CE. If "off," then the element has already been processed; go to step 3. If "on," set it to "off" and set the tag field to zero.

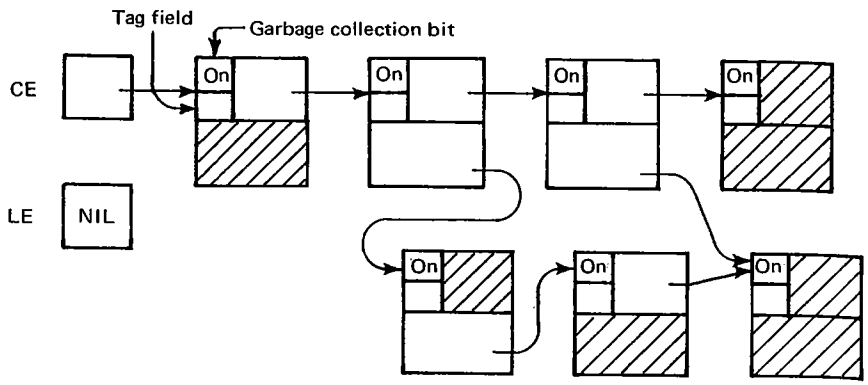
2. *Follow a pointer to a new element.* Let $j$ be the contents of the tag field of the current element. Scan the current element, beginning at the $j$ + 1*st* location, for pointers to other heap elements. If none are found, go to step 3. If a pointer is found in the $k$th location, set the tag field to $k$ and do a circular transfer of pointers between CE, LE, and the $k$th location. CE gets the pointer found in the $k$th location, LE gets the original contents of CE, and the $k$th location gets the original contents of LE. Return to step 1.

3. *Retrace a chain to a preceding element.* If LE is NIL, go to step 4. Otherwise return to the element pointed to by LE. If the tag field of this element contains $k$, do a (reverse) circular transfer of pointers between the $k$th location, LE, and CE. LE gets the pointer in the $k$th location, CE gets the original contents of LE, and the $k$th location gets the pointer originally in CE. Return to step 2.

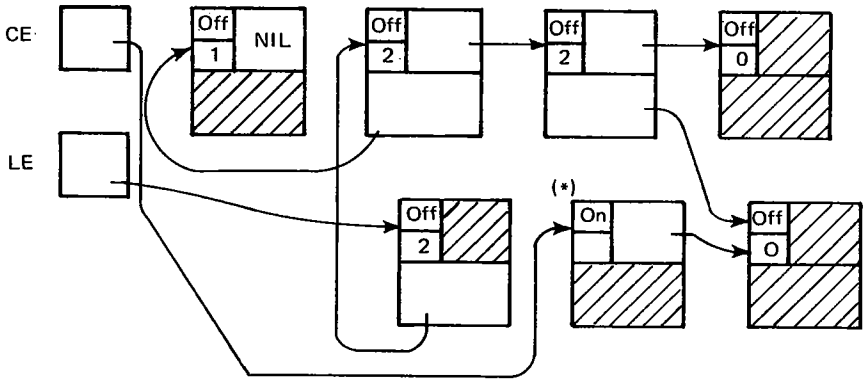
4. *Move to the next structure to be marked.* All the active elements reachable from the original external pointer have now been marked. Proceed to set CE to the next external pointer into the heap and begin again at step 1.

Figure 7-10 illustrates the marking of a list structure by this algorithm.

This algorithm is considerably less efficient than the simple stack marking algorithm since each list must be traversed twice rather than only once. Schorr and Waite suggest that a better garbage collection technique would combine the stack marking algorithm and the marking algorithm just given, using a fixed-size block as a stack until it was full and then using the latter algorithm. Many other marking algorithms are known as well; see Knuth's discussion and analysis [1968].




Structure at the start of marking
(fields not containing pointers are shaded)




Structure during marking, immediately before marking of the element (*)

**Fig. 7-10.** Marking during garbage collection.

One of the striking features of garbage collection as a method of storage recovery is that its cost is *inversely* proportional to the amount of storage recovered (roughly speaking); i.e., the less storage recovered, the more it costs to garbage collect. The reason is that most of the cost of garbage collection is in the marking phase, and marking becomes more complex as the number of active elements in the heap increases. In contrast the cost of recovery using reference counts is directly proportional to the amount of storage recovered. As a result of this property of garbage collection the technique



becomes very expensive when the heap is nearly full of active elements. In fact often when a program is about to use up all the heap (and thus halt for lack of storage) it will first initiate a number of expensive and relatively fruitless garbage collections, each of which produces only a few free locations which are immediately consumed when execution resumes, leading immediately to another garbage collection. The assumption is often made in designing garbage collection algorithms that any program which has almost used up the heap will almost surely soon use it up entirely, and thus a garbage collection which produces fewer than some minimum number of free locations should probably cause program termination anyway.

## 7-8. HEAP STORAGE MANAGEMENT: VARIABLE-SIZE ELEMENTS

Heap storage management where variable-size elements are allocated and recovered is more difficult than with fixed-size elements, although many of the same concepts apply. Variable-size elements arise in many situations. For example, if space is being used for programmer-defined data structures stored sequentially, such as arrays, then variable-size blocks of space will be required, or activation records for subprograms might be allocated in a heap (e.g., in a simulation language) in sequential blocks of varying sizes.

The major difficulties with variable-size elements concern reuse of recovered space. Even if we recover two five-word blocks of space in the heap, it may be impossible to satisfy a later request for a six-word block. This problem did not arise in the simpler case of fixed-size blocks; recovered space could always be immediately reused.

### Initial Allocation and Reuse

With fixed-size elements it was appropriate to immediately split the heap into a set of elements and then base initial allocation on a free space list containing these elements. Such a technique is not acceptable with variable-size elements. Instead we wish to maintain free space in blocks of as large a size as possible. Initially then we consider the heap as simply one large block of free storage. A *heap pointer* is appropriate for initial allocation. When a block of $N$ words is requested the heap pointer is advanced by $N$ and the original heap

pointer value returned as a pointer to the newly allocated element. As storage is freed behind the advancing heap pointer it may be collected into a free space list.

Eventually the heap pointer reaches the end of the heap block. Some of the free space back in the heap must now be reused. Two possibilities for reuse present themselves, because of the variable size of the elements:

1. Use the free space list directly for allocation, searching the list for an appropriate size block and returning any leftover space to the free list after the allocation.

2. Compact the free space by moving all the active elements to one end of the heap, leaving the free space as a single block at the end and resetting the heap pointer to the beginning of this block.

Let us look at these two possibilities in turn.

*Reuse Directly from a Free Space List.* The simplest approach, when a request for an $N$-word element is received, is to scan the free space list for a block of $N$ or more words. A block of $N$ words can be allocated directly. A block of more than $N$ words must be split into two blocks, an $N$-word block, which is immediately allocated, and the remainder block, which is returned to the free space list. The basic idea is straightforward. A number of particular techniques for managing allocation directly from such a free space list are known:

1. *First-fit method.* When an $N$-word block is needed the free space list is scanned for the *first* block of $N$ or more words, which is then split into an $N$-word block, and the remainder, which is returned to the free space list.

2. *Best-fit method.* When an $N$-word block is needed the free space list is scanned for the block with the *minimum* number of words greater than or equal to $N$. This block is allocated as a unit, if it has exactly $N$ words, or is split and the remainder returned to the free space list.

The first-fit and best-fit methods may be combined if the free space list is maintained with blocks in order of *increasing size*. However, ordering by *memory location* may be more desirable for purposes of compaction (see below). Knuth [1968] analyzes both the first-fit and best-fit techniques and concludes that the first-fit method is generally to be preferred for a variety of reasons.



### Recovery with Variable-Size Blocks

Before considering the memory compaction problem, let us look at techniques for recovery where variable-size blocks are involved. Relatively little is different here from the case of fixed-size blocks. Explicit return of freed space to a free space list is the simplest technique, but the problems of garbage and dangling references are again present. Reference counts may be used in the ordinary manner.

Garbage collection is also a feasible technique. Some additional problems arise with variable-size blocks, however. Garbage collection proceeds as before with a marking phase followed by a collecting phase. Marking must be based on the same pointer chain following techniques. The difficulty now is in collecting. Before, we collected by a simple sequential scan of memory, testing each element's garbage collection bit. If the bit was "on," the element was returned to the free space list; if "off," it was still active and was passed over. We should like to use the same scheme with variable-size elements, but now there is a problem in determining the boundaries between elements. Where does one element end and the next begin? Without this information the garbage cannot be collected.

The simplest solution is to maintain along with the garbage collection bit in the first word of each block, active or not, an integer *length indicator* specifying the length of the block. With the explicit length indicators present, a sequential scan of memory is again possible, looking only at the first word of each block. During this scan, adjacent free blocks may also be compacted into single blocks before being returned to the free space list, thus eliminating the partial compaction problem discussed below (see Problem 7-7).

Garbage collection may also be effectively combined with full compaction to eliminate the need for a free space list altogether. Only a simple heap pointer is needed in this case (see below).

### Compaction and the Memory Fragmentation Problem

The problem that any heap storage management system using variable-size elements faces is that of memory *fragmentation*. One begins with a single large block of free space. As computation proceeds this block is progressively fragmented into smaller pieces through allocation, recovery, and reuse. If only the simple first-fit or best-fit allocation technique is used, it is apparent that free space blocks continue to split into ever smaller pieces. Ultimately one reaches a point where the storage allocator cannot honor a request

for a block of $N$ words because no sufficiently large block exists, even though the free space list contains in total far more than $N$ words. Without some compaction of free blocks into larger blocks execution will be halted by a lack of free storage faster than necessary.

Depending on whether active blocks within the heap may be shifted in position, one of two approaches to compaction is possible:

1. *Partial compaction.* If active blocks *cannot* be shifted (or if it is too expensive to do so), then only adjacent free blocks on the free space list may be compacted.

2. *Full compaction.* If active blocks *can* be shifted, then all active blocks may be shifted to one end of the heap, leaving all free space at the other.

*Partial Compaction.* Partial compaction is the simplest technique. By partial compaction we mean the combining of two or more adjacent free blocks into a single larger free block. Where storage is recovered by garbage collection, partial compaction is automatic (see Problem 7-7). Where storage is freed piecemeal (through explicit return, with or without reference counts), concern with compaction is necessary. Observe that because blocks are returned to the free space list in essentially random order it is quite likely that two blocks which are adjacent in memory will become free. Because they are freed at different times, it is not apparent that they are adjacent in memory. It is desirable to compact such adjacent free blocks into single blocks as they occur. The simplest way to do this is to maintain the free space list in *order of memory location*, as in Fig. 7-11. When a block is returned to the free space list its position is found in the ordering and the preceding list entry checked to determine adjacency. If the two blocks are adjacent, they are combined into one larger block which is entered into the free space list at the same place (but only after checking that the following list entry may not also be combined into the new block).

A much faster algorithm using an unordered free space list and requiring no search of the free space list when a block is returned is possible. Knuth [1968] gives the details of the technique. The requirements are

1. A doubly linked free space list (each block containing pointers to its successor and predecessor in the list),

2. A reserved bit in the first and last word of each active heap block set to "active," and



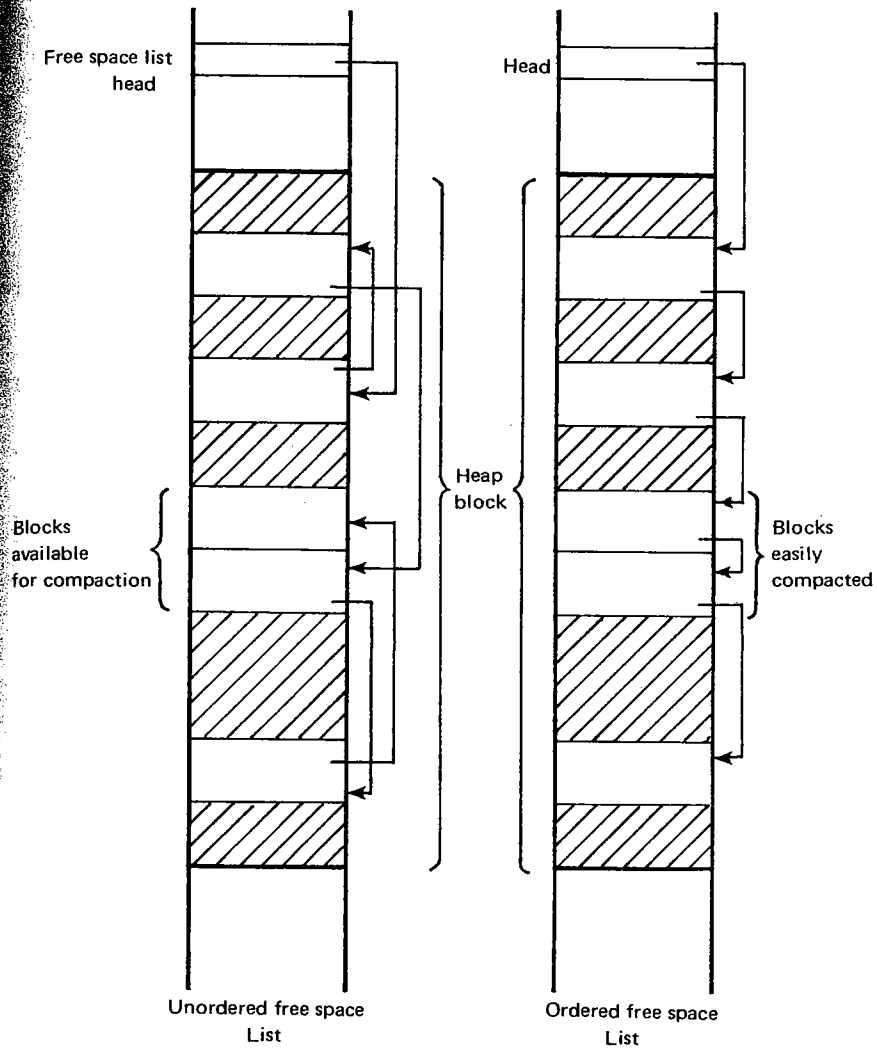



**Fig. 7-11.** Ordered and unordered free space lists.

3. A similar pair of reserved bits in each free block set to "free."

The reader may easily work out the details himself (see Problem 7-8).

*Full Compaction.* Full compaction of the heap through shifting of all active blocks to one end is more difficult. Partial compaction

may be accomplished without concern for the structure of active blocks or pointers external to the heap. The difficulty in full compaction lies in readjusting pointers to the active heap elements that are shifted to new locations. Clearly, if a block $B$ is moved to a new location, then any pointer to $B$ must be modified to point to $B$'s new location. Full compaction requires

1. Identification of all external pointers to heap elements, and

2. Identification of all pointers to another heap element from an active heap element.

Unlike partial compaction, full compaction must be treated as an extraordinary procedure, as with garbage collection. The computation in progress must be suspended while compaction takes place. Full compaction has some of the other characteristics of garbage collection as well because of the need to mark active heap blocks. It is perhaps simplest to combine the two techniques, utilizing a single marking procedure followed by a compaction and a readjustment of pointers. Garbage is identified and compacted simultaneously. Such a technique is used in many SNOBOL4 implementations and is worth considering in greater detail.

The SNOBOL4 technique requires reservation within each block, whether active or free, of (1) a garbage collection bit, (2) a block-length designator, and (3) a *compaction pointer* field. Each block's compaction pointer initially points to the first location of that block. Full compaction proceeds in four steps:

1. *Marking.* Marking of active blocks proceeds exactly as in garbage collection, setting the garbage collection bit of each active block "off."

2. *Compaction pointer setting.* A sequential scan of the heap block is made, moving two pointers. One pointer, $P$, is advanced through the heap, pointing to the first location of each block in turn. The second, $Q$, initially points to the beginning of the first block. As each block is encountered by pointer $P$ its garbage collection bit is checked. If "on," the block is garbage and $P$ is advanced to the next block. If "off," then the block is active. Its compaction pointer is set to the current value of the pointer $Q$, and then both $P$ and $Q$ are advanced by the length of the block just checked. After being set, the compaction pointer of a block indicates the position the block *will have* after compaction.



3. *Pointer resetting.* The pointer chains followed in marking are followed again, replacing each pointer by the compaction pointer of the block pointed to, before following the original pointer.

4. *Block shifting.* A second sequential scan of memory is performed, shifting the contents of each block up to begin at the location specified by its compaction pointer (and resetting garbage collection bits). After completion of the scan all active blocks will be at one end of the heap and all free space at the other. The heap pointer may now be reset to the beginning of the free space and the suspended computation resumed. Figure 7-12 illustrates the technique.

This section has only touched on some basic heap storage management problems and techniques. Many alternative techniques are known. Knuth [1968] describes and analyzes a number of these.

## 7-9. REFERENCES AND SUGGESTIONS FOR FURTHER READING

Storage management considerations are a part of many of the papers concerned with control structures which are referenced in Chapters 5 and 6. The texts by Harrison [1973] and Donovan [1972] are useful general references. Gries [1971] treats stack-based storage management in some detail. Bobrow and Wegbreit [1973] consider a stack management technique for a variety of control structures.

Ross [1967] describes a general-purpose run-time storage management system used in a number of language implementations. Techniques for heap storage management have been widely studied. Knuth [1968] analyzes a number of techniques, including the important *Buddy system*. Harrison [1973] describes a number of garbage collection methods. Particular techniques are taken up in Schorr and Waite [1967], Hansen [1969], and Griswold [1972].

Storage management using both external storage and central memory is an important topic in operating systems design and is becoming increasingly important in programming language implementation. The use of *overlays* constructed by the programmer is a simple and widely used technique. However, many recent hardware designs include *virtual memory* structures based on *paging* or *segmentation* which insulate the programmer from concern with

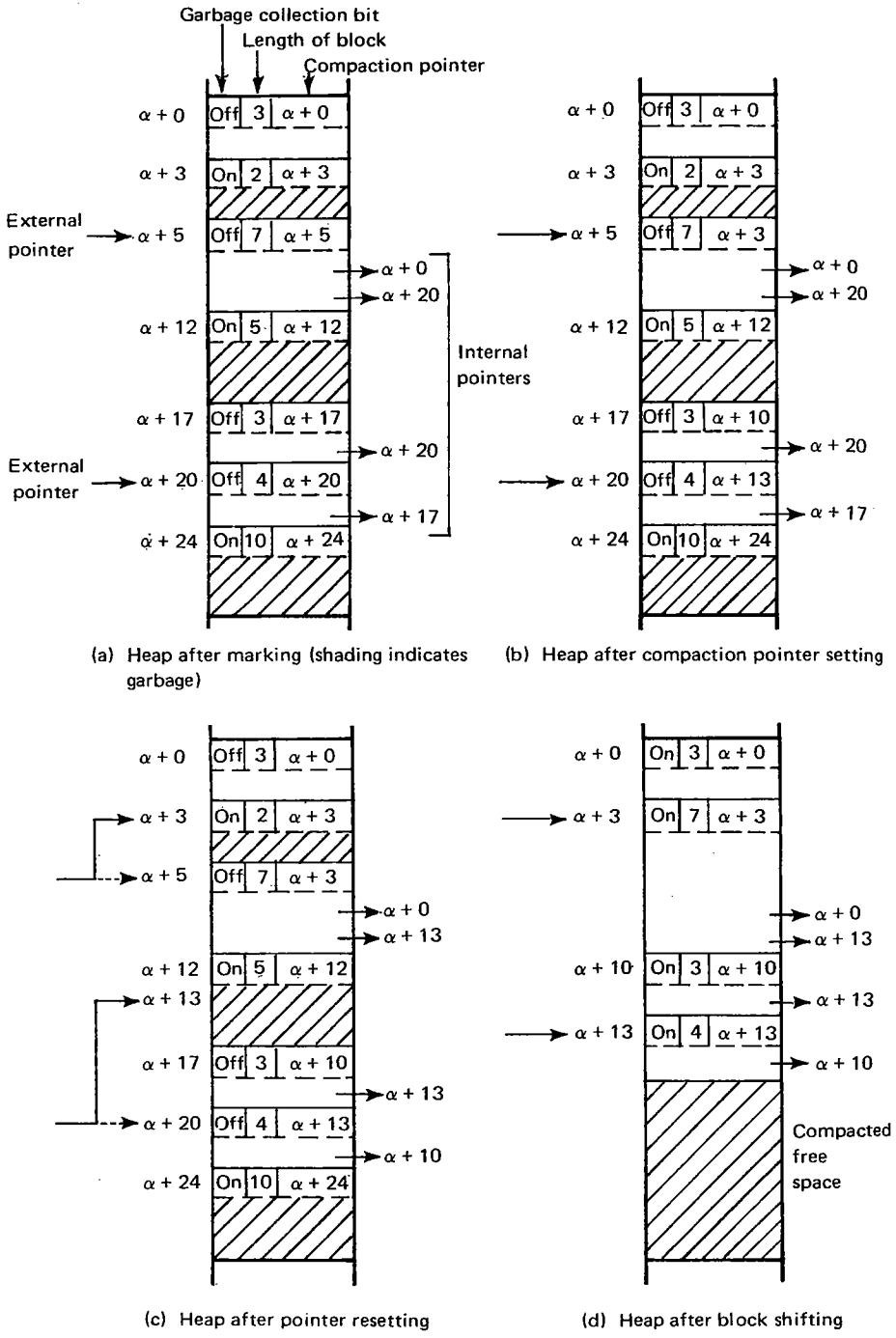



**Fig. 7-12.** Garbage collection with full compaction.





storage management. Two survey papers by Denning [1970, 1971] and a paper by Sayre [1969] serve as useful introductions to these topics. The problem of heap storage management in virtual memory systems is considered by Bobrow and Murphy [1967] and Baecker [1972], among others.

## 7-10. PROBLEMS

7-1. Analyze the storage management techniques used in a language implementation available to you. Consider the various elements requiring storage mentioned in Section 7-2. Are there other major run-time structures requiring storage besides those mentioned in Section 7-2?

7-2. Fill in the details of the steps in the marking of the list structure of Fig. 7-10 using the Schorr and Waite marking algorithm.

7-3. In the SLIP list-processing extension to FORTRAN each list has a special header (containing a reference count). When a list is freed, instead of returning all the list elements to the head of the free space list, only the list header is returned, and it is placed at the *end* of the free space list (using a special pointer to the end of the free space list). Thus the cost of returning a list to free space is minimal. The list elements are returned to the free space list only when the header of the list reaches the top of the free space list. What is the advantage of this technique for shifting the cost of freeing the list elements from the time of *recovery* of the list to the time of *reuse* of the storage?

7-4. Perform a worst-case analysis to determine the maximum size of the stack necessary for marking during garbage collection using the simple stack-based marking algorithm of Section 7-7. Assume that a heap element may contain at most two pointers to other heap elements, that there are at most $J$ external pointers into the heap, and that the heap contains space for $N$ elements altogether. How much space would have to be reserved for the garbage collection stack to ensure that garbage collection could be completed without stack overflow?

7-5. Modify the Schorr and Waite marking algorithm to allow each location within an element to contain possibly two pointers rather than only a single pointer.

7-6. *Full-word* data items such as numbers present special problems in garbage collection. Ordinarily the data themselves take up the entire heap element, with no extra bit available for garbage collection marking. It is usual in such cases to separate all such full-word elements into a special section of the heap and store all the garbage collection bits for these full-word elements in a special packed array (a *bit vector*) outside the heap. Assuming that the full-word data items are numbers (and thus contain no pointers to other heap elements), design a garbage collection algorithm

which allows the possibility of pointers to full-word items in the heap. The algorithm should include marking and collecting. Maintain a separate free space list for the full-word portion of the heap.

7-7. Give an algorithm for the *collection* of marked blocks during garbage collection in a heap with variable-size elements. Assume that the first word of each block contains a garbage collection bit and a length indicator. Compact all adjacent blocks during the collection.

7-8. Without consulting Knuth's volume, design an algorithm for return of a block to free space and partial compaction using an unordered doubly linked free space list, variable-size blocks, and a reserved active-free bit in the first and last words of each block.

7-9. *APL array storage.* Design a storage management system for a heap $B$ of variable-size blocks under the following assumptions:

1. $B$ is used only for value storage blocks for arrays of real numbers (one word per number). Arrays always have at least two elements.

2. Each array block is accessible only through a single external pointer stored in the array descriptor. All array descriptors are stored in a block $A$ separate from the heap $B$.

3. Requests for blocks to be allocated from the heap occur at random (and frequently) during execution.

4. Blocks are explicitly returned to "free" status when arrays are destroyed. This also occurs randomly and frequently during execution.

5. Permanent loss of storage through memory fragmentation cannot be tolerated. (Note that a one-word free block can never be reused.)

Your design should specify

a. The *initial organization* of the heap block $B$, together with any special external structures needed for storage management (e.g., free space list heads).
b. The *storage allocation* mechanism, given a request for a block of $N$ words from $B$.
c. The *storage recovery* mechanism, given a call on the storage manager with the address of a block of $M$ words to be freed.
d. The *compaction* mechanism, if any, including a specification of how it works and when it is invoked.

# 8 OPERATING ENVIRONMENT

In the preceding five chapters we have taken up the basic components of programming languages and their associated virtual computers. Data, operations, control structures, and storage management combine to define the internal structure of programs during execution. One important aspect has escaped other than brief mention, however: the aspect of what we shall term *operating environment*.

The operating environment of a program consists of those elements external to the program which may be used to communicate with the program, i.e., which serve as the interface between the program and the "outside world." For a typical set of programs this operating environment may consist only of a set of input devices from which data may be obtained for processing and a set of output devices on which results may be written. We might be even more restrictive and consider only the operating environment of a single subprogram. Such an environment would include not only I/O devices but also the nonlocal referencing environment of the program. However, we have considered the problems of these *internal operating environments* associated with particular subprograms at length in Chapter 6; our concern in this chapter is with the larger operating environments associated with the complete set of main program and subprograms which combined form a "program" in the usual sense.

## 8-1. DATA FILES

The most basic interactions of a program with its operating environment take place through data files, files used during execu-