

**IBM****Data Processing Techniques****Techniques for Processing Data Lists in PL/I**

This manual illustrates techniques for processing simple and complex data lists. It is a sequel to *Introduction to the List Processing Facilities of PL/I* (GF20-0015) and assumes knowledge of that manual. Illustrative programs were processed by the PL/I (F) Compiler (Version 5) under control of the IBM System/360 Operating System (Release 18.6).

Data lists are made up of based variable structures that contain data plus pointers that link the structures. List-processing techniques are useful for handling data that has logical complexities not conveniently represented by conventional PL/I array and structure representation.

This manual is intended for the experienced programmer.

First Edition (January 1971)

Copies of this and other IBM publications can be obtained through IBM branch offices.

A form has been provided at the back of this publication for readers' comments. If this form has been removed, address comments to: IBM Corporation, Technical Publications Department, 112 East Post Road, White Plains, New York 10601.

## Preface

This manual serves as a sequel to the companion text *Introduction to the List Processing Facilities of PL/I* (GF20-0015). That text describes the facilities for processing lists in PL/I. The present manual extends the discussion of simple data lists to data lists of arbitrary complexity and shows how to develop hierarchical collections of subroutines and functions for manipulating such lists. To make the discussion as self-contained as possible, review material has been included on the PL/I facilities for organizing and processing lists (Chapter 1) and on the essentials of subroutines and functions (Appendix 1). Appendix 2 summarizes the language facilities in PL/I for processing lists.

List-processing techniques are advantageous for organizing and manipulating data whose structure is not conveniently represented with PL/I arrays and structures.

Structured data of this type occurs in many nonnumeric applications, such as information storage and retrieval, engineering design, computer-software production, text editing, and artificial intelligence.

The advanced nature of this manual requires the reader to be an experienced programmer who has studied the companion text mentioned above and who is skilled in the use of subroutines and functions. References to particular implementations of PL/I are held to a minimum, but information on the F-level facilities for processing lists appears in *IBM System/360: PL/I Reference Manual* (GC28-8201) and *IBM System/360 Operating System: PL/I (F) Programmer's Guide* (GC28-6594). A bibliography provides additional sources of information on list organizations and their applications.



	<i>Page</i>		<i>Page</i>
<b>Chapter 1: Introduction</b> . . . . .	<b>1</b>	<b>Manipulating List Components</b> . . . . .	<b>18</b>
<b>PURPOSE AND PLAN OF THIS MANUAL</b> . . . . .	<b>1</b>	<i>Obtaining the Size of a Data List</i> . . . . .	<b>18</b>
<b>REVIEW OF FACILITIES FOR PROCESSING</b>		<i>Inserting Data Items into Data Lists</i> . . . . .	<b>19</b>
<b>LISTS IN PL/I</b> . . . . .	<b>1</b>	<i>Getting the Values of Data Items in Data</i>	
<b>Pointer Variables</b> . . . . .	<b>1</b>	<i>Lists</i> . . . . .	<b>25</b>
<i>Obtaining Values for Pointer Variables</i> . . . . .	<b>1</b>	<i>Deleting Data Items from Data Lists</i> . . . . .	<b>28</b>
<i>Using Pointer Variables in Assignment</i>		<i>Removing Data Items from Data Lists</i> . . . . .	<b>33</b>
<i>Statements</i> . . . . .	<b>2</b>	<i>Replacing Data Items in Data Lists</i> . . . . .	<b>34</b>
<i>Using Pointer Variables in Operational</i>		<i>Searching for Data Items in Data Lists</i> . . . . .	<b>37</b>
<i>Expressions</i> . . . . .	<b>2</b>	<i>Interchanging Data Items in Data Lists</i> . . . . .	<b>39</b>
<b>Based Variables</b> . . . . .	<b>2</b>	<b>Manipulating Sublists and Lists</b> . . . . .	<b>39</b>
<i>Qualifying Based Variables with Pointer</i>		<i>Inserting Sublists and Lists into Data Lists</i> . . . . .	<b>40</b>
<i>Variables</i> . . . . .	<b>3</b>	<i>Deleting Sublists and Lists from Data Lists</i> . . . . .	<b>43</b>
<i>Restrictions on Based Variables</i> . . . . .	<b>3</b>	<i>Assigning Sublists and Lists to Data Lists</i> . . . . .	<b>43</b>
<i>Contextual Declarations of Pointer</i>		<i>Linking Data Lists</i> . . . . .	<b>47</b>
<i>Variables</i> . . . . .	<b>4</b>	<i>Splitting Data Lists</i> . . . . .	<b>47</b>
<i>Based Storage</i> . . . . .	<b>4</b>	<i>Catenating Data Lists</i> . . . . .	<b>47</b>
<b>Area Variables</b> . . . . .	<b>5</b>	<i>Searching Data Lists for Sublists</i> . . . . .	<b>53</b>
<i>Allocating and Freeing Based Storage in an</i>		<i>Testing Data Lists for Equality</i> . . . . .	<b>53</b>
<i>Area</i> . . . . .	<b>6</b>	<i>Comparing Data Lists</i> . . . . .	<b>56</b>
<i>Emptying an Area</i> . . . . .	<b>6</b>	<i>Reversing Data Lists</i> . . . . .	<b>57</b>
<i>The AREA ON-condition</i> . . . . .	<b>7</b>	<i>Sorting Data Lists</i> . . . . .	<b>57</b>
<i>Linking Allocations of Based Storage in</i>		<i>Converting Character Strings to and from</i>	
<i>an Area</i> . . . . .	<b>7</b>	<i>Data Lists</i> . . . . .	<b>60</b>
<b>DATA LISTS</b> . . . . .	<b>9</b>	<b>Manipulating Lists Recursively</b> . . . . .	<b>60</b>
<b>Chapter 2: Simple Data Lists</b> . . . . .	<b>10</b>	<i>Recursive Deletion of Data Lists</i> . . . . .	<b>63</b>
<b>ORGANIZING SIMPLE DATA LISTS</b> . . . . .	<b>10</b>	<i>Recursive Computation of Data List Sizes</i> . . . . .	<b>66</b>
<b>PROCESSING SIMPLE DATA LISTS</b> . . . . .	<b>10</b>	<i>Recursive Searching for Data Items in Data</i>	
<b>Creating a List of Available Storage</b>		<i>Lists</i> . . . . .	<b>69</b>
<b>Components</b> . . . . .	<b>11</b>	<i>Recursive Linking of Data Lists</i> . . . . .	<b>69</b>
<i>AREA_OPEN Subroutine</i> . . . . .	<b>11</b>	<i>Recursive Testing for Equality of Data</i>	
<b>Manipulating the Elements of List Components</b> . . . . .	<b>12</b>	<i>Lists</i> . . . . .	<b>70</b>
<i>Obtaining the Address of a List Component</i> . . . . .	<b>12</b>	<i>Recursive Comparison of Data Lists</i> . . . . .	<b>71</b>
<i>Assigning Values to the Elements of List</i>		<b>Using Simple Data Lists</b> . . . . .	<b>72</b>
<i>Components</i> . . . . .	<b>14</b>	<i>Editing Cash Values</i> . . . . .	<b>72</b>
<i>Obtaining the Values of Elements in List</i>		<i>Removing Edit Symbols from Cash Values</i> . . . . .	<b>73</b>
<i>Components</i> . . . . .	<b>16</b>	<i>Expanding a Multiple Assignment Statement</i> . . . . .	<b>75</b>
<i>Examples</i> . . . . .	<b>18</b>		

	<i>Page</i>
<i>Expanding a Picture Specification . . . . .</i>	75
<i>Contracting a Picture Specification . . . . .</i>	79
<i>Adding Variable-Length Integers . . . . .</i>	79
<i>Subtracting Variable-Length Integers . . . . .</i>	82
<i>Gathering Declare Statements in a Procedure . . . . .</i>	84
<b>REVIEW OF SIMPLE DATA LISTS . . . . .</b>	<b>87</b>
<b>SUMMARY . . . . .</b>	<b>88</b>
<b>Chapter 3: Complex Data Lists . . . . .</b>	<b>89</b>
<b>MORE GENERAL DATA IN LIST COMPONENTS . . . . .</b>	<b>89</b>
<b>DESCRIPTIVE DATA IN LIST HEADS . . . . .</b>	<b>89</b>
<b>ALTERNATIVE METHODS FOR LINKING LIST COMPONENTS . . . . .</b>	<b>93</b>
<b>Two-Way Lists . . . . .</b>	<b>93</b>
<b>Circular Lists . . . . .</b>	<b>94</b>

	<i>Page</i>
<b>Multidirectional Lists . . . . .</b>	<b>91</b>
<b>ADVANCED APPLICATIONS OF COMPLEX DATA LISTS . . . . .</b>	<b>91</b>
<b>List Representation of Structural Formulas in Chemistry . . . . .</b>	<b>91</b>
<b>List Representation of Geometric Figures . . . . .</b>	<b>101</b>
<b>List Representation of a Chessboard . . . . .</b>	<b>101</b>
<b>REVIEW OF COMPLEX DATA LISTS . . . . .</b>	<b>101</b>
<b>SUMMARY . . . . .</b>	<b>111</b>
<b>Appendix 1: Review of Facilities for Subroutines and Functions in PL/I . . . . .</b>	<b>111</b>
<b>Appendix 2: Summary of List-Processing Facilities . . . . .</b>	<b>111</b>
<b>Bibliography . . . . .</b>	<b>111</b>
<b>Index . . . . .</b>	<b>111</b>

## Chapter 1: Introduction

### PURPOSE AND PLAN OF THIS MANUAL

This manual extends the discussion of list processing presented in the companion manual *Introduction to the List Processing Facilities of PL/I* (GF20-0015). That manual describes the facilities for organizing and manipulating lists in PL/I and illustrates the advantages of lists over the conventional methods for organizing storage with arrays and structures.

The present manual develops techniques for creating data lists of any desired complexity. It begins with a review of the facilities for organizing lists in PL/I and applies these facilities in subsequent chapters to the construction of data lists, which are collections of noncontiguous data items linked by attached address elements; the data items associated with a data list always appear within the body of the list. Techniques for processing data lists are presented in subroutine and function form. (Appendix 1 contains a review of PL/I facilities for subroutines and functions.)

Version 5, Release 18.6 of the PL/I (F) Compiler produced the coded examples shown as printouts in this manual. The encoded examples were all compiled and subjected to elementary machine tests to perform as intended, except for four examples that illustrate specialized organizations of complex data lists. These were desk-checked, compiled, and executed. Note, however, that programming efficiency, speed of execution, core storage utilization, etc., are frequently sacrificed in the examples and discussions to simplify the presentation and to improve readability.

The program examples are purely illustrative in nature; no attempt was made to develop "production" code.

The list-processing programmer may wish to compare the convenience of using subroutines and functions to manipulate list components against the better performance of inline code.

### REVIEW OF FACILITIES FOR PROCESSING LISTS IN PL/I

The facilities for processing lists in PL/I deal with four types of variables: AREA, BASED, OFFSET, and POINTER, each of which is discussed in detail in the companion manual *Introduction to the List Processing Facilities of PL/I* (GF20-0015). The following discussions review the main characteristics of three of these variables: POINTER, BASED, and AREA.

### Pointer Variables

The value of a *pointer variable* is an absolute address, which specifies the actual location of a data item in storage. Declaration of an identifier with the POINTER attribute establishes the identifier as a pointer variable.

#### EXAMPLES:

```
DECLARE
  P POINTER,
  (Q,R) POINTER EXTERNAL STATIC,
  T(5) POINTER INTERNAL,
  V(-2:2,-3:3) POINTER,
  1 A,
  2 X CHARACTER(15),
  2 Y POINTER,
  1 TABLES,
  2 I(5) POINTER,
  2 J(0:4) POINTER;
```

As shown in these examples, PL/I allows pointer variables to be individual element variables or elements of arrays and structures. A pointer variable can have any storage class and scope, and the usual default rules for these attribute types also hold for a pointer variable.

#### Obtaining Values for Pointer Variables

A pointer variable must receive an absolute address for its value before the variable can be manipulated. One way of obtaining an absolute address for a pointer variable is through the built-in functions ADDR and NULL. A reference to the built-in function ADDR has the form:

ADDR (argument)

The value returned by ADDR is the absolute address of the specified argument. As an example, consider the assignment statement:

P=ADDR(X);

Assume that P is a pointer variable and X is a data variable. Then the reference ADDR(X) obtains the absolute address of the storage location allocated for X, and the statement assigns this absolute address as the value of pointer P.

An argument variable in a reference to ADDR must be an identifier that specifies one of the following types of variables:

1. Element variable
2. Array
3. Element of an array
4. Major or minor structure
5. Element of a structure

It is also possible for an element constant to appear as an argument of ADDR. In this case, the PL/I compiler creates a dummy-argument variable for the constant, and ADDR returns the absolute address of the dummy argument. (See Appendix 1 for a discussion of dummy arguments.)

A reference to the built-in function NULL uses no arguments and has the form:

NULL

This function returns a null address value, which does not identify any location in storage. A null address is used to clear pointer variables and to test for unallocated storage.

Note that PL/I does not provide explicit address constants for pointer variables.

#### *Using Pointer Variables in Assignment Statements*

PL/I permits pointer variables to appear in the following forms of the assignment statement:

1. element-pointer-variable=element-pointer-expression;
2. pointer-array=element-pointer-expression;
3. pointer-array=pointer-array;

Two or more variables separated by commas may appear on the left side of these statements to permit multiple assignment. An element-pointer expression on the right side of an assignment statement must be either an element-pointer variable or a function reference (built-in or programmer-defined) that specifies an element-pointer value, that is, a single absolute address.

Assignment of an element-pointer expression to a pointer array causes the value of the expression to be assigned to every element of the pointer array. When a pointer array appears on the right side of an assignment statement, the number of dimensions and the bounds for each dimension of the array on the right must be identical to those of the receiving pointer array on the left.

EXAMPLES:

DECLARE

```
(P,Q,R,T(5),V(-2:2,-3:3)) POINTER,  
1 A, 2 X CHARACTER(15), 2 Y POINTER,  
1 TABLES, 2 I(5) POINTER, 2 J(0:4) POINTER;
```

The following statements illustrate possible assignments of pointer values that involve the above declarations:

1. P = ADDR(A);
2. Q,R = NULL;
3. T(4), V(-2,-3) = P;
4. A.Y = T(4);
5. TABLES.I, TABLES.J = NULL;
6. T = TABLES.I;

Statements 2, 3, and 5 perform multiple assignments. The last three statements show name qualification applied to pointer variables.

#### *Using Pointer Variables in Operational Expressions*

PL/I allows only two operators to use pointer variables as operands: the comparison operators equal (=) and not equal ( $\neq$ ). As a result of this restriction, arithmetic operations cannot be performed on absolute addresses.

EXAMPLES:

Assume that A and B are arithmetic variables and that P,Q,R,S, and T are pointer variables; then the following statements contain permissible uses of pointer variables as operands.

1. IF P = NULL THEN GO TO L;
2. A = (Q  $\neq$  R);
3. IF (T=NULL) | ((T=ADDR(B)) & (S=NULL)) THEN P = Q;

#### **Based Variables**

Although the value of a pointer variable specifies the absolute address of a data item, the pointer itself provides no information about the data item. For example, a pointer variable can specify the absolute address of an array A, but neither the pointer name nor the pointer value indicates the dimensions of A or the characteristics of its elements.

To associate descriptive information with the address value of a pointer variable, PL/I provides a special type of variable called the *based variable*, which is declared with the attribute:

BASED (element-pointer-variable)

The element-pointer variable within the BASED attribute cannot be a based variable, nor can it be subscripted.

Before reference can be made to a based variable, an address value must be assigned to the pointer variable specified in the associated BASED attribute. Note that declaration of a based variable does not assign an address value to its associated pointer variable.



A reference to a based variable applies the attributes of the based variable to the storage location specified by the associated pointer variable. Consider the declaration:

```
DECLARE NAME CHARACTER(15) BASED(P);
```

This statement declares the 15-position character string called NAME to be a based variable and associates the pointer variable P with NAME. For example, let NAME appear in the assignment statement:

```
NAME = 'JOHN';
```

This statement assigns the character-string constant 'JOHN' to a 15-position storage area at the location given by P. The four characters of the constant are positioned to the left in the area and are followed by eleven blank characters.

The BASED attribute is a storage-class attribute along with AUTOMATIC, CONTROLLED, and STATIC. Appearance of BASED in a DECLARE statement, however, does not produce an allocation of storage. Only when an absolute address is assigned to the pointer variable related to the based variable does storage become associated with the based variable. Consider, for example, the previous declaration of the based variable NAME. Not until an absolute address is assigned as the value of pointer P does storage become associated with NAME. When this association occurs, NAME is said to be "based on" P.

The value of the pointer variable in a BASED attribute can specify a location of any storage class and data type, including an array or a structure. When applied to a structure, the BASED attribute must appear at level 1 and, consequently, applies to all members of the structure. Care must be taken, though, when changing the value of the pointer related to a based variable to assure compatibility between the attributes of the based variable and the data at its newly assigned location.

#### *Qualifying Based Variables with Pointer Variables*

PL/I allows a based variable to be associated with more than one storage area at the same time. This multiple association is possible because a based variable by itself does not specify a data item, but only a description of storage. It is the combination of pointer variable and based variable that determines the location and description of a data item.

Since only one pointer variable can appear in a BASED attribute, some other facility is required for simultaneously associating two or more pointers with the same based variable. The PL/I facility that permits this multiple association is called *pointer qualification*. It is used to distinguish among two or more storage areas associated with the same based variable, and allows other pointers to

override the pointer that was specified in the declaration of the based variable.

Pointer qualification is denoted by a composite symbol that resembles an arrow. The symbol consists of a minus sign followed immediately by a greater-than symbol ( $->$ ). This composite symbol, however, does not signify an operation; its function is similar to that of the period symbol used in the qualified name of a structure element. When used, the pointer qualification symbol must always appear between two references. The reference on the left must be either an element-pointer variable or a reference to the built-in function ADDR. When it is an element-pointer variable, it cannot be subscripted, nor can it be of the based-storage class. The reference on the right of the pointer qualification symbol must be a based variable.

A pointer qualification symbol applies the storage description of the based variable on its right to the storage location specified by the pointer value on its left; the pointer originally declared with the based variable is overridden. As an example, consider the assignment statement:

```
A->B = C->B;
```

Assume that B is a based variable, and A and C are non-based, element-pointer variables. The expression  $C->B$  refers to a data item that has the attribute declared for B and the location specified by C. Similarly, the expression  $A->B$  determines the location and attributes of the area to which the data item is assigned. Thus, the pointer qualification symbols in this example associate the attributes declared for the based variable B with the two distinct storage areas addressed by pointers A and C. This use of the arrow symbol ( $->$ ) to associate a based variable with a specific pointer variable constitutes pointer qualification.

#### *Restrictions on Based Variables*

The following restrictions apply to based variables:

1. The EXTERNAL attribute cannot appear in the declaration of a based variable, but a based variable can be qualified by an external pointer variable.
2. Based variables cannot have the INITIAL attribute, nor can arrays of based labels be initialized by subscripted label prefixes.
3. Data-directed input and output cannot transmit the value of a based variable.
4. The BASED attribute cannot be specified for the parameters of subroutines or functions.
5. The CHECK ON-condition cannot be applied to a based variable.
6. The VARYING attribute cannot be applied to a based variable.

### Contextual Declarations of Pointer Variables

The appearance of an identifier in one of the following contexts serves as a contextual declaration of the identifier as a pointer variable:

1. In a BASED variable
2. On the left of a pointer qualification symbol (->)
3. In the SET option of READ, LOCATE, and ALLOCATE statements (discussed later)

A contextually declared pointer variable receives the AUTOMATIC storage class and INTERNAL scope by default. If different attributes are desired, they must appear in an explicit declaration along with the POINTER attribute. The pointer variable contextually declared with a based variable does not receive the null pointer value as a result of the based declaration, and only the INITIAL CALL form of the INITIAL attribute is allowed in explicit declarations of pointer variables.

### Based Storage

Based variables can be used with ALLOCATE and FREE statements for direct control of storage allocation.

Allocation of storage for a based variable is performed with the ALLOCATE statement, which has the basic form:

```
ALLOCATE based-variable;
```

When executed, this statement allocates storage for the based variable and assigns the absolute address of the allocated storage to the pointer variable specified in the BASED attribute of the based variable. The attributes associated with the based variable determine the amount of storage that is allocated.

EXAMPLE:

```
DECLARE
  TABLE(5) BASED(P) FIXED DECIMAL(3);
```

```

.
.
.
ALLOCATE TABLE;
```

These statements declare TABLE to be a based array and allocate storage for the array. Each of the five elements in the array is a three-digit fixed-point decimal integer. The location of the allocated storage for the array is automatically assigned to pointer P, which appears in the BASED attribute of TABLE.

Reallocation of storage for a based variable does not free previously allocated storage for the variable. Instead, both the old storage and the new storage are available provided the old value of the associated pointer is saved before it is automatically replaced by the location of the new storage. Several allocations of storage for the same

based variable may be distinguished by appropriate pointer qualification.

EXAMPLE:

```
DECLARE
  T POINTER,
  SWITCH BIT(2) BASED(P);
ALLOCATE SWITCH;
T= P;
ALLOCATE SWITCH;
T->SWITCH = '11'B;
SWITCH = '10'B;
.
.
.
```

In this example, SWITCH is a based variable that represents a two-position bit string. T and P are pointer variables. After each allocation of storage for SWITCH, pointer P contains the address of the allocated storage. Pointer T receives the address of the first allocation before the second allocation is executed. The statement T->SWITCH = '11'B; assigns the bit-string constant '11'B to the first storage location allocated for SWITCH.

As illustrated in the preceding example, each allocation of storage for a based variable assigns the address of the new allocation to the pointer variable specified in the BASED attribute associated with the based variable. When two or more allocations of storage are performed concurrently for the same based variable, the addresses of previous allocations must be saved in separate pointer variables; otherwise, the previous addresses will be lost.

So far, the address of a previous allocation has been saved by the assignment statement. But PL/I also provides the SET option in an ALLOCATE statement as an alternative method for assigning the address of an allocation to a pointer variable. An ALLOCATE statement with a SET option has the following form:

```
ALLOCATE based-variable SET
(element-pointer-variable);
```

This statement allocates storage for the based variable and assigns the address of the allocated storage to the pointer variable specified in the SET option. The pointer variable must represent a single pointer value; it cannot be the name of an array of pointers or a structure of pointers.

An ALLOCATE statement without a SET option is treated as having an implicit SET option that applies to the pointer variable in the BASED attribute of the allocated variable. An explicit SET option allows the programmer to specify a pointer variable different from the one given in the BASED attribute of the allocated variable. This other

pointer receives the address of the allocated storage, and the pointer variable in the BASED attribute remains unchanged.

EXAMPLE:

```

DECLARE
  P POINTER,
  VALUE BASED(Q) FLOAT;
ALLOCATE VALUE;
ALLOCATE VALUE SET(P);
.
.
.

```

The first ALLOCATE statement allocates storage for VALUE and assigns the location of the storage to pointer Q. The second ALLOCATE statement allocates additional storage for VALUE and assigns the location of this new storage to pointer P. The value of pointer Q and the storage allocated by the first ALLOCATE statement remain unchanged by the second allocation.

Storage allocated for a based variable is freed for possible reuse by the FREE statement, which has the following form:

```
FREE based-variable;
```

This statement frees the storage currently associated with the based variable. The program obtains the address of this storage from the current value of the pointer variable declared in the BASED attribute of the based variable. The attributes of the based variable determine the amount of storage that is freed.

EXAMPLE:

```

DECLARE
  P POINTER,
  ITEM BASED(Q) CHARACTER(10);
ALLOCATE ITEM;
ALLOCATE ITEM SET(P);
.
.
.
FREE ITEM;
FREE P->ITEM;
.
.
.

```

In these statements, P and Q are pointer variables, and ITEM is a character-string based variable. Two allocations of storage occur for ITEM. Pointer Q contains the location of the first allocation; pointer P, the second. The first free statement frees the storage for ITEM at the location speci-

fied by Q. The second FREE statement frees the storage for ITEM at the location specified by P.

A based variable can be used to free a particular allocation of storage only if that storage has been allocated for a based variable with the same attributes, including array bounds, string lengths, and arithmetic precisions. An attempt to free a based variable for which storage has not been allocated produces unpredictable results.

### Area Variables

PL/I provides a type of variable called the *area variable*, which reserves storage for allocations of based variables. The area variable permits based allocations to be grouped as a unit for convenient input/output transmission or assignment to another area while maintaining the separate identity of each allocation. The following discussion describes how area variables are used to group based allocations.

An identifier becomes an area variable when it is declared with the AREA attribute, which has the form:

```
AREA(size-expression)
```

The value of the size expression determines the size of the area in bytes. However, the expression is optional; when it is not used, an implementation-defined size is assumed by the PL/I compiler.

Although an area variable reserves storage for allocations of based variables, it can have any storage class. The size of an area with static storage class must appear in the AREA attribute as an unsigned fixed-point decimal integer constant. The AREA attribute is not restricted to element identifiers; it can also be used with array and structure identifiers. PL/I also provides for area arguments and parameters in subroutines and functions, and the asterisk notation can be used to denote the size of an area parameter. The DEFINED attribute permits an area to be defined on another area through overlay or corresponding defining; both areas, however, must have the same size.

EXAMPLES:

```

DECLARE
  A STATIC AREA (32767),
  B AREA,
  C AREA(N),
  D AREA(S) CONTROLLED,
  E AREA(10000) BASED(P),
  F(5) AREA(400),
  1 G,
  2 H AREA(100),
  2 I AREA(200),
  2 J AREA(300),
  K AREA DEFINED B,
  L AREA(*);

```

This DECLARE statement specifies that:

1. A is a static area variable that reserves 32767 bytes of storage.
2. B is an automatic area variable that reserves an implementation-defined amount of storage.
3. C is an automatic area variable whose size depends on the value of N current at the time the block to which it is internal is activated.
4. D is a controlled area variable whose size depends either on the value of S at the time an ALLOCATE statement allocates storage for D, or on a size specification in the ALLOCATE statement, which overrides S.
5. E is a based area variable that reserves 10,000 bytes of storage for each allocation.
6. F is an array that contains five area elements, each of which reserves 400 bytes of automatic storage.
7. G is an area structure that contains the three areas H, I, and J. H reserves 100 bytes of automatic storage, I reserves 200 bytes, and J reserves 300 bytes.
8. K is an area defined on area B.
9. L is an area parameter that assumes the same size as its associated area argument in a subroutine or function invocation.

#### *Allocating and Freeing Based Storage in an Area*

The ALLOCATE statement uses the IN option for based allocations within a specified area:

```
ALLOCATE based-variable
SET(pointer-variable) IN(area-variable);
```

This statement allocates storage for the based variable within the specified area and assigns the location of the allocated storage to the pointer variable. The IN option, however, is not required; when it is not used, the based variable is allocated in a storage area provided by the operating system.

The FREE statement, as applied to a based allocation in a specified area, has the form:

```
FREE based-variable
IN(area-variable);
```

The IN option must appear in a FREE statement if the based allocation was made within an explicitly specified area; otherwise, the option is omitted.

EXAMPLE:

```
DECLARE
STORE AREA(500)   BASED(P),
VALUE BASED(Q)   FIXED DECIMAL(5,2),
R   POINTER;
.
.
.
```

```
ALLOCATE STORE;
/* P ADDRESSES AREA STORE. */
.
.
.
ALLOCATE VALUE IN(STORE);
/* Q ADDRESSES ALLOCATION OF VALUE
   IN STORE. */
.
.
.
ALLOCATE VALUE IN(P->STORE) SET(R);
/* R ADDRESSES SECOND ALLOCATION
   OF VALUE IN STORE. */
.
.
.
FREE VALUE IN(STORE);
/* FREES ALLOCATION OF VALUE
   ADDRESSED BY Q. */
.
.
.
FREE R->VALUE IN(STORE);
/* FREES ALLOCATION OF VALUE
   ADDRESSED BY R. */
.
.
.
```

The first ALLOCATE statement allocates 500 bytes of storage for area STORE and assigns the location of the allocated storage to pointer P.

The second ALLOCATE statement causes storage for based variable VALUE to be allocated within P->STORE and assigns the location of the allocated storage to pointer Q.

The third ALLOCATE statement generates another allocation of VALUE (different from Q->VALUE) within area P->STORE and sets pointer R equal to the location of the allocated storage.

The FREE statements employ the IN option because allocations of VALUE were explicitly made in STORE. Although the allocations of VALUE become free, the storage for area STORE remains allocated.

#### *Emptying an Area*

When an area is allocated, it automatically receives the empty state; that is, it contains no allocations of based variables. An area that is not empty can be made empty by assigning an empty area to it or by assigning the value of the built-in function EMPTY:

```
AREA1 = EMPTY;
AREA2 = AREA1;
```

### The AREA ON-condition

An attempt to allocate based storage within an area that contains insufficient free storage for the allocation produces an AREA ON-condition. If no ON-unit appears in an ON statement for the AREA condition, the operating system issues a comment and raises the ERROR condition.

When an ON-unit is specified and normal return occurs from the ON-unit, the ALLOCATE statement that raised the AREA condition is executed again. If the ON-unit has changed the value of a pointer qualifying (explicitly or implicitly) the reference to the inadequate area so that the pointer value specifies another area, the allocation is reattempted within the new area. Failure of the ON-unit to provide a larger area may place the program in an error loop.

### Linking Allocations of Based Storage in an Area

Techniques for processing lists depend upon the ability to link collections of noncontiguous data items in any desired order by attached address elements. The PL/I structure organization provides a convenient way of attaching an address element to a data item. As an example, consider the declaration:

```

DECLARE
  1 COMPONENT BASED(P),
  2 DATA CHARACTER(80),
  2 LINK POINTER;

```

COMPONENT is a based structure that contains two elements: DATA, which is a character string of 80 positions, and LINK, which is a pointer variable. Placing LINK and DATA in the same structure effectively attaches LINK to DATA. The diagram in Figure 1.1 illustrates this attachment. Rectangles represent storage for DATA and LINK, and the protruding arrow indicates that LINK is a pointer variable whose value points to another allocation of storage.

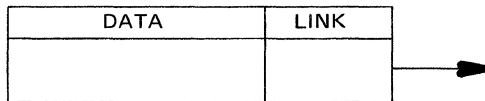


Figure 1.1. A data element with an attached pointer

Based structure COMPONENT may be used to allocate and link storage throughout an area, as indicated in Figure 1.2. The following statements show how such organization of storage may be performed.

```

DECLARE
  AREA1 AREA(1000),
  (AVAIL, TEMP, P) POINTER,
  1 COMPONENT BASED(P),
  2 DATA CHARACTER(80),
  2 LINK POINTER;
/* WHEN ALL STORAGE HAS BEEN ALLOCATED IN
  AREA1, SET LINK POINTER OF LAST COM-
  PONENT TO NULL AND GO TO LABEL NEXT. */
ON AREA
  BEGIN; P->LINK=NULL; GO TO NEXT; END;
/* ALLOCATE FIRST COMPONENT IN AREA1 AND
  ASSIGN THE COMPONENT ADDRESS TO
  POINTER AVAIL. */
ALLOCATE COMPONENT IN(AREA1) SET(P); AVAIL
= P;

/* CONTINUE ALLOCATING COMPONENTS IN
  AREA1 UNTIL ALL STORAGE HAS BEEN ALLO-
  CATED. LINK EACH COMPONENT TO THE PRE-
  VIOUSLY ALLOCATED COMPONENT. */
L: TEMP = P;
  ALLOCATE COMPONENT IN (AREA1) SET(P);
  TEMP->LINK = P;
  GO TO L;
NEXT: . . .
      . . .
      . . .
      . . .

```

Pointer AVAIL contains the address of the first allocation of COMPONENT in AREA1, and the LINK pointer of each allocation contains the address of the next allocation. A null address is assigned to the LINK pointer of the last allocation.

The organization shown in Figure 1.2 establishes AREA1 as a pool of available storage components from which free storage may be obtained when needed. Pointer AVAIL provides access to the first storage component, and successive components are reached by proceeding through the LINK pointers of the components. Chaining storage components in this way forms a special type of data organization called a *list*, which may be referred to by the name of the head pointer AVAIL.

Figure 1.3 illustrates how the first two components linked to AVAIL may be removed from AVAIL and linked to another head pointer to form a second list called LIST1. Similarly, when the components of LIST1 are no longer needed, they can be relinked to AVAIL, where they become available to other lists.

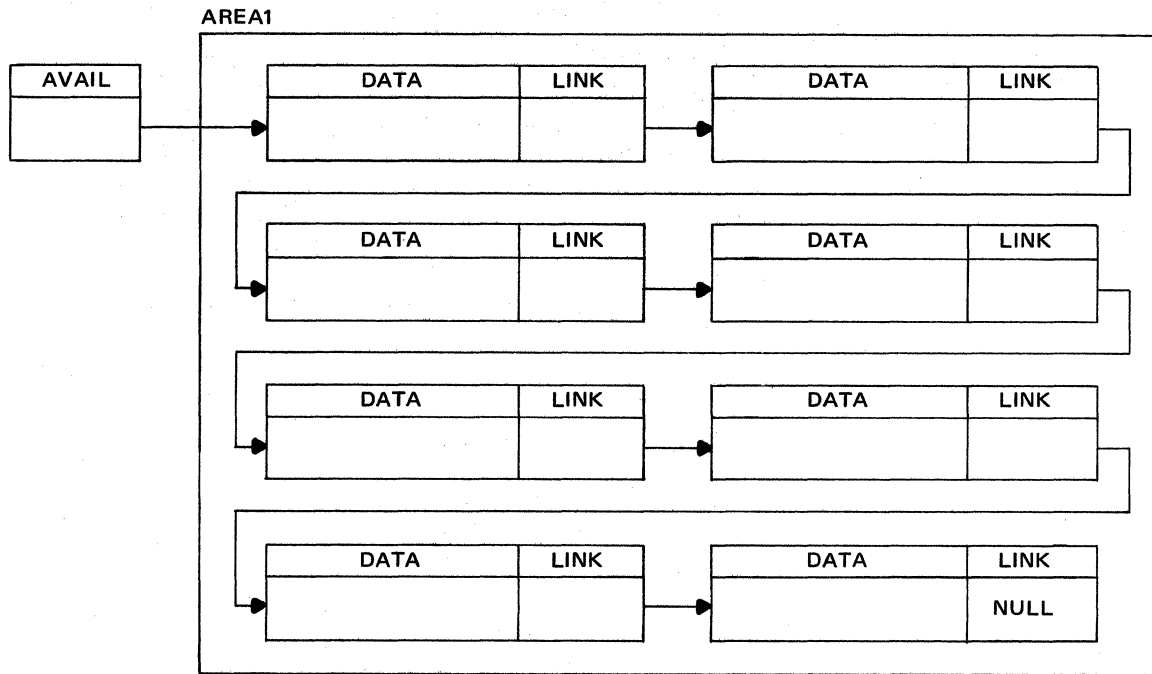


Figure 1.2. Allocations of based storage linked by pointer variables

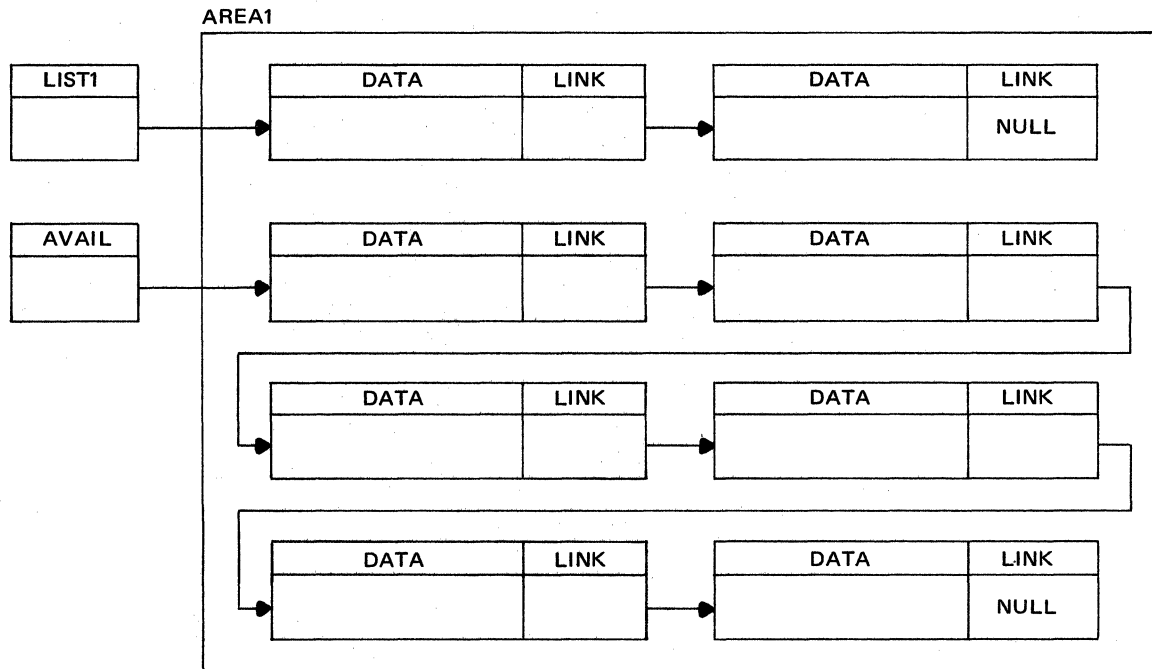


Figure 1.3. Linking available storage components to another list

This type of storage manipulation permits the storage requirements of a list to vary during the course of program execution. A list need reserve only the storage it is actually using at any given moment; storage need not lie dormant within each list in anticipation of maximum storage requirements, as it often does in conventional array and structure organizations. As a result, the programmer is freed from having to know exactly how much storage each list will require.

Linking storage by means of attached address elements can improve the execution time of a program by reducing the amount of data that must be moved. For example, if the components of a list are to be sorted on their data values, it is not necessary to change the physical positions of the data values in storage; their logical positions within the list can be changed by manipulating the attached address elements. Logically successive components need not occupy physically successive storage locations. Scattered components can be linked in any desired order.

## DATA LISTS

The type of list organization illustrated in Figures 1.2 and 1.3 is called a *data list* because it consists of linked data items. The lists in those illustrations possess a linear ordering: each component except the first has one predecessor, and each component except the last has one successor. As Figure 1.4 shows, this type of list consists of a head and a body. The head is a pointer variable that identifies the list and contains the address of the first component in the list. The body is a sequence of list components, each of which contains a data item and a pointer variable. Except for the last pointer, which has a null address, the pointer in a component contains the address of the next component in the list.

This type of list organization always requires a head and permits the body of a list to contain an arbitrary number of components, limited only by available storage. It is even possible for the body of a list to contain no components; in this case, the list is said to be null (see Figure 1.5).

Figures 1.4 and 1.5 do not show the area within which storage has been allocated for list components. These illustrations emphasize the main parts of a list and deemphasize the environmental aspects of list organization. They also stress the close resemblance between this type of list and a character string or a one-dimensional array, the major difference being that successive components of a list need not occupy contiguous storage locations.

Examples of subroutines used to allocate based variables throughout an area are provided in this manual.

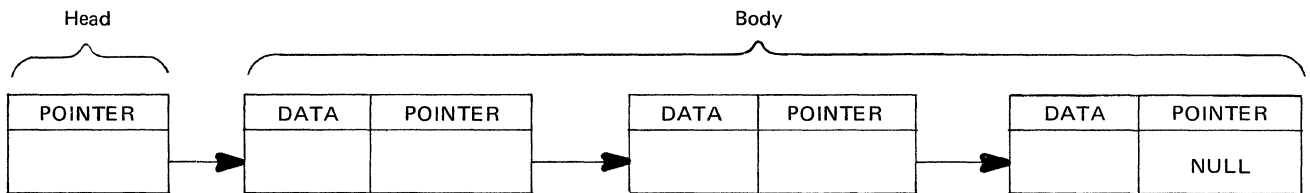


Figure 1.4. Data list

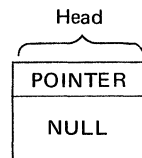


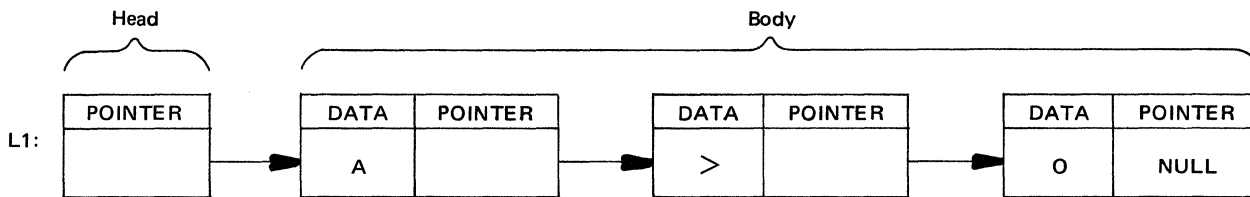
Figure 1.5. Null list

## Chapter 2: Simple Data Lists

This chapter develops methods for organizing and manipulating simple data lists and shows how these methods may be used in list-processing applications.

### ORGANIZING SIMPLE DATA LISTS

To avoid unnecessary complexity, this chapter uses a simple list organization in which each list component contains a single alphanumeric character for its data element. As an example, consider this illustration of a simple data list:

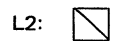


This list contains three characters: A, >, and O; each is a separate component of the list. The identifier L1, attached to the head of the list by a colon, specifies the name of the head pointer and, therefore, serves as the name of the data list.

Elimination of descriptive words from the elements of the list produces a more compact illustration:



Note that a diagonal rather than the keyword NULL indicates a null pointer. The following representation, therefore, specifies that L2 is a null list:



Also observe that a list of one or more blank characters is not a null list, since a blank is an encodable machine character even though it does not have a printable graphic. The following list, L3, contains two blank characters:



Restricting the data element of each list component to a single character simplifies the task of developing and illustrating list-processing techniques. It also permits analogies to be developed between simple data lists and character strings.

The above illustrations of data lists, however, are completely arbitrary and do not form a necessary feature of PL/I.

Chapter 3 discusses data lists with more complex organizations that require more elaborate illustrations.

### PROCESSING SIMPLE DATA LISTS

All data lists, despite the variety of their possible applications, undergo common basic operations, such as inserting, retrieving, and deleting list items. Because most operations performed on lists require several steps, which may be repeated many times during the course of program execution, it is often more convenient and efficient to provide these operations in subroutine or function form rather than as independent sets of PL/I statements that must be duplicated throughout a program. The following discussions show how a collection of subroutines and functions can be designed to simplify many list-processing operations. The routines cover five general categories:

1. Creating a list of available storage components
2. Manipulating the elements of list components
3. Manipulating list components (not just their elements)
4. Manipulating sublists and lists
5. Manipulating lists recursively



A hierarchical approach is taken in the development of routines for each of these categories. Procedures concerned with the primitive aspects of storage allocation are programmed first and are used in turn to create higher level procedures. This approach limits the number of procedures that deal with environmental factors and permits the complete collection of subroutines and functions to possess an application-oriented emphasis.

The sequential organization of the routines is maintained so that each routine uses only those list-processing procedures that have been developed earlier. No attempt is made at complete programming efficiency. Wherever possible, programming methods have been chosen to simplify the presentation for the reader rather than to produce efficient code.

### Creating a List of Available Storage Components

The amount of storage needed by a data list can vary during the course of program execution. As new data items are inserted into the list, additional storage is required, and when data items are deleted from the list, the associated storage becomes free.

The list-processing techniques developed in this chapter assume that any storage not needed by a data list is reserved in a special list of available storage components. This special list serves as a storage pool, which initially contains all the storage used to form the components of other lists.

As a data list grows, additional storage for the list is obtained from the list of available storage components. Similarly, when list components become free, their storage is returned to the list of available storage components. As a result, the same storage can be used by many different lists during the execution of a program. Sharing storage in this way reduces the amount of storage that might lie dormant within individual data lists in anticipation of maximum storage requirements for each list.

The creation of a list of available storage components is performed in this chapter with the `AREA_OPEN` subroutine, which is discussed next.

#### *AREA\_OPEN Subroutine*

Figures 2.1A, 2.1B, and 2.1C present the `AREA_OPEN` subroutine, which requires two arguments:

1. An area variable throughout which list components are allocated
2. A pointer variable that serves as the head of the list of available storage components

<b>AREA_OPEN Subroutine</b>	
<b>Purpose</b>	To create a list of available storage components
<b>Reference</b>	<code>AREA_OPEN(AREA, LIST)</code>
<b>Entry-Name Declaration</b>	<code>DECLARE AREA_OPEN ENTRY(AREA(*), POINTER);</code>
<b>Meaning of Arguments</b>	<p><code>AREA</code> --the area variable that is to contain the list of available storage components</p> <p><code>LIST</code> --the pointer variable that serves as the head of the list of available storage components</p>
<b>Remarks</b>	Storage must have been allocated for the <code>AREA</code> argument before <code>AREA_OPEN</code> is invoked. The <code>LIST</code> argument is assumed to be null upon entry to <code>AREA_OPEN</code> .
<b>Other Programmer-Defined Procedures Required</b>	None
<b>Method</b>	Storage for list components is allocated with the following based structure: <p style="text-align: center;"> <code>1 COMPONENT BASED(P),</code>  <code>2 DATA CHARACTER(1),</code>  <code>2 POINTER POINTER,</code> </p>
	Components are allocated throughout <code>AREA</code> until the <code>AREA ON</code> -condition occurs.
	The <code>LIST</code> argument contains the address of the first component. The <code>POINTER</code> element of each component contains the address of the next component. The <code>POINTER</code> element of the last component is null.

Figure 2.1A. Description of the `AREA_OPEN` subroutine for creating a list of available storage components

```

AREA_OPEN:
PROCEDURE
  (AREA,LIST);
  DECLARE
    AREA AREA(*),
    (LIST, P, T) POINTER,
    1 COMPONENT BASED(P),
    2 DATA CHARACTER(1),
    2 POINTER POINTER;
    /* WHEN ALL STORAGE HAS BEEN
    ALLOCATED IN AREA, SET POINTER OF
    LAST COMPONENT, IF ANY, TO NULL AND
    LEAVE SUBROUTINE. */
    ON AREA
  BEGIN;
    IF
      P->=NULL
    THEN
      P->POINTER = NULL;
      GO TO
        END_AREA_OPEN;
  END;
  /* ALLOCATE FIRST COMPONENT IN
  AREA, AND ASSIGN COMPONENT ADDRESS
  TO THE POINTER PARAMETER CALLED
  LIST. */
  P = NULL;
  ALLOCATE COMPONENT IN(AREA)
  SET(P);
  LIST = P;
  /* CONTINUE ALLOCATING COMPONENTS IN
  AREA UNTIL ALL STORAGE HAS BEEN
  ALLOCATED. LINK EACH COMPONENT
  TO THE PREVIOUSLY ALLOCATED
  COMPONENT. */
L: T = P;
  ALLOCATE COMPONENT IN(AREA) SET(P);
  T->POINTER = P;
  GO TO
    L;
  END_AREA_OPEN:
  END
  AREA_OPEN;

```

Figure 2.1B. Creating a list of available storage components for data lists

The area argument passed to AREA\_OPEN can be of any storage class and is not restricted to a particular size, but storage for the area must have been allocated before the subroutine is invoked. Although only one area is specified in an invocation of AREA\_OPEN, several invocations of the subroutine can allow the list of available storage components to occupy more than one area.

The AREA\_OPEN subroutine must be invoked at least once before another list-processing procedure is used; otherwise, no storage will be available for list formation. The sample procedures in this chapter use the identifier LIST as the head pointer of the data list being acted upon. The external identifier AVAIL is used as the head pointer of a list of storage components that can be inserted into the list named LIST. A list component in LIST that becomes superfluous can be deleted from LIST and inserted

into AVAIL. As later procedures demonstrate, declaring AVAIL to be an external identifier avoids having to pass the name of the list of available storage components as an argument each time it is used by a subroutine or function.

The AREA\_OPEN subroutine can be invoked to create a list of components named LIST and again to create a list of components named AVAIL:

```

CALL AREA_OPEN (AREA1,LIST);
CALL AREA_OPEN (AREA2,AVAIL);

```

Note that the allocation of a based variable into an area is optional; list components in scattered locations can be linked by their pointer elements. However, an area containing list components linked by offset variables can be moved about in main storage or transmitted to and from external storage. An offset variable is a storage address that is relative to the address of the beginning of an area. (See *Introduction to the List Processing Facilities of PL/I*, GF20-0015, for discussion and illustration of the use of offset variables in forming and transmitting relocatable data lists.)

### Manipulating the Elements of List Components

All list-processing operations involve at least one of these items:

1. Address of a list component
2. Data element of a list component
3. Pointer element of a list component

The following discussions develop subroutines and functions that:

1. Obtain the address of a list component
2. Assign values to the data and pointer elements of a list component
3. Obtain the values of the data and pointer elements of a list component

These procedures eliminate the syntactic details associated with PL/I pointer qualification and allow the programmer to view and process data lists in an application-oriented manner.

### Obtaining the Address of a List Component

Because a list component is formed by allocating a based variable, the address of the component must be obtained before it can be processed. The following discussions develop two function procedures that obtain the address of a specified list component:

1. ADDRESS\_N, which obtains the address of the nth component in a data list
2. ADDRESS\_NEXT, which obtains the address of the list component that follows a given component

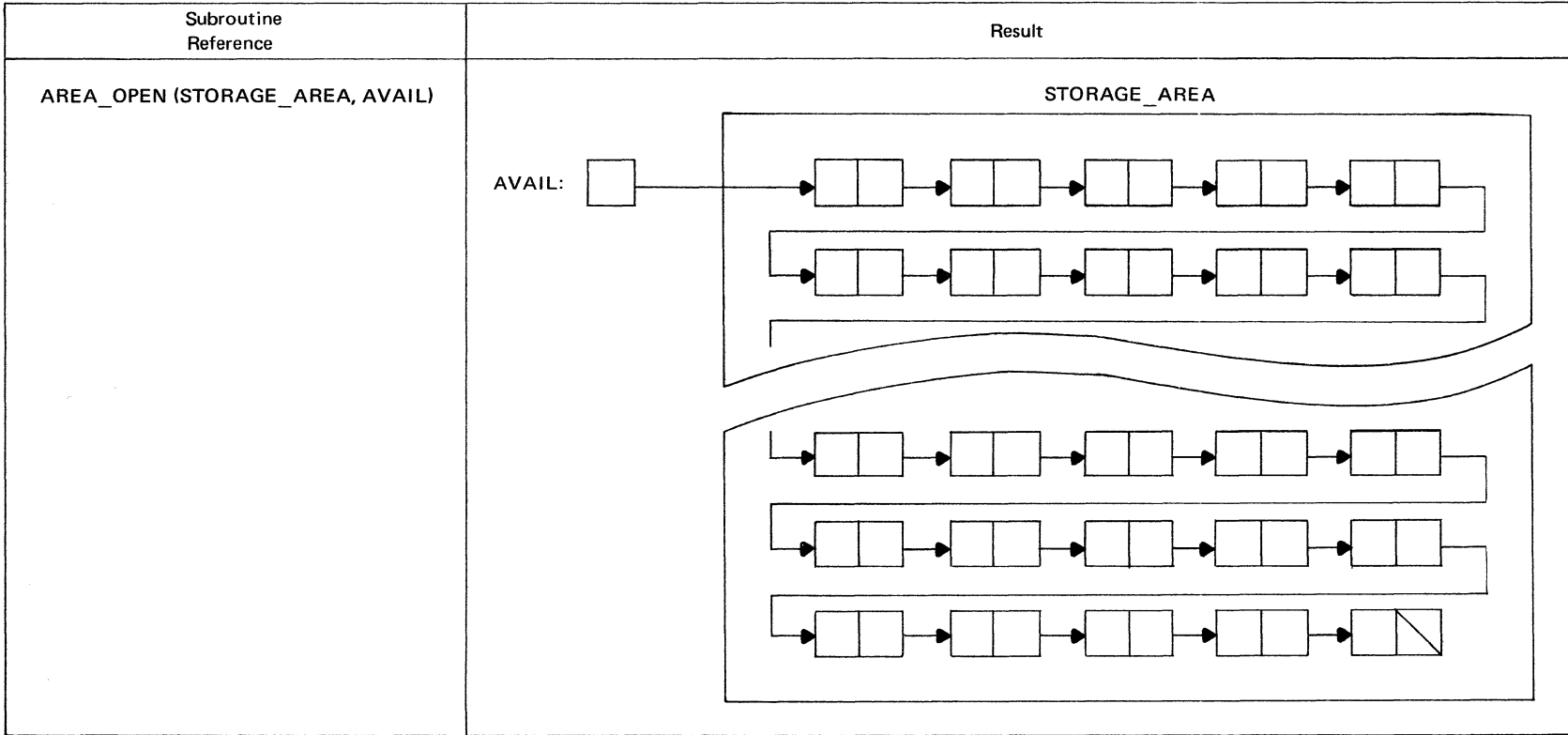


Figure 2.1C. An example of a reference to the AREA\_OPEN subroutine

**ADDRESS\_N Function** Figures 2.2A and 2.2B present the ADDRESS\_N function procedure. This function requires two arguments:

1. A pointer variable that forms the head of the data list in which the specified component appears
2. An integer that specifies the sequential list position (first, second, third, etc.) of the component whose address is desired

The function returns the address of the specified component.

<b>ADDRESS_N Function</b>	
<b>Purpose</b>	To obtain the address of the nth component of a data list
<b>Reference</b>	ADDRESS_N(LIST, N)
<b>Entry-Name Declaration</b>	DECLARE ADDRESS_N ENTRY(POINTER, FIXED DECIMAL(5)) RETURNS(POINTER);
<b>Meaning of Arguments</b>	LIST --the pointer variable that is the head of the list to be examined N --a fixed-point decimal integer value that specifies the component whose address is to be obtained; N has a maximum size of five digits
<b>Remarks</b>	A null pointer value is returned when LIST is null, N is less than one, or N is greater than the number of components in LIST.
<b>Other Programmer-Defined Procedures Required</b>	None
<b>Method</b>	The function proceeds through LIST until the (n-1)th component is reached. The pointer element of this component contains the address of the nth component.

Figure 2.2A. Description of the ADDRESS\_N function for obtaining the address of the nth component of a data list

```
ADDRESS_N:
  PROCEDURE(LIST, N)
  RETURNS (POINTER);
DECLARE
  (LIST, ADDRESS) POINTER,
  (N, I) FIXED DECIMAL(5),
  1 COMPONENT BASED(ADDRESS),
  2 DATA CHARACTER(1),
  2 POINTER POINTER;
  IF
    (LIST = NULL)|(N<1)
  THEN
    RETURN (NULL);
    ADDRESS = LIST;
DO
  I = 1 BY 1;
  IF
    (ADDRESS->POINTER = NULL) &
    (I=N)
  THEN
    RETURN (NULL);
  IF
    I = N
  THEN
    RETURN(ADDRESS);
    ADDRESS = ADDRESS->POINTER;
END;
END
ADDRESS_N;
```

Figure 2.2B. Obtaining the address of the nth component of a data list

**ADDRESS\_NEXT Function** When list components are processed in sequence, it is inefficient to use the ADDRESS\_N function to obtain successive list components, because the function always searches for a component from the beginning of the list. Figures 2.3A and 2.3B present the ADDRESS\_NEXT function, which causes the address of one list component to obtain the address of the next component in sequence. ADDRESS\_NEXT obtains the address of the next component directly from the pointer element of the component that is specified by the address argument in an invocation of the function.

**Assigning Values to the Elements of List Components**

When data items are inserted into or deleted from a data list, the data and pointer elements of list components must be changed. The following discussions develop two subroutines that perform such changes:

1. SET\_DATA, which assigns a value to the data element of a list component
2. SET\_POINTER, which assigns a value to the pointer element of a list component

**SET\_DATA Subroutine** Figures 2.4A and 2.4B present the SET\_DATA subroutine, which requires two arguments:

1. Address of a data list component
2. Character value to be assigned to the data element of the list component

### ADDRESS\_NEXT Function

#### Purpose

To obtain the address of the next component in a data list

#### Reference

ADDRESS\_NEXT(ADDRESS)

#### Entry-Name Declaration

```
DECLARE ADDRESS_NEXT ENTRY(POINTER)
  RETURNS(POINTER);
```

#### Meaning of Argument

ADDRESS --a pointer value that specifies the address of a data list component

#### Remarks

The function assumes that ADDRESS represents a valid address of a data list component. If ADDRESS is null, a null pointer value is returned.

#### Other Programmer-Defined Procedures Required

None

#### Method

The function returns the address contained in the pointer element of the component specified by ADDRESS.

Figure 2.3A. Description of the ADDRESS\_NEXT function for obtaining the address of the next component in a data list

```
ADDRESS_NEXT:
  PROCEDURE (ADDRESS)
  RETURNS (POINTER);
DECLARE
  ADDRESS POINTER,
  1 COMPONENT BASED(ADDRESS),
  2 DATA CHARACTER(1),
  2 POINTER POINTER;
  IF
    ADDRESS = NULL
  THEN
    RETURN (NULL);
  RETURN(ADDRESS->POINTER);
END
  ADDRESS_NEXT;
```

Figure 2.3B. Obtaining the address of the next component in a data list

### SET\_DATA Subroutine

#### Purpose

To assign a value to the data element of a data list component

#### Reference

SET\_DATA(ADDRESS, D)

#### Entry-Name Declaration

```
DECLARE SET_DATA ENTRY(POINTER,
  CHARACTER(1));
```

#### Meaning of Arguments

ADDRESS --a pointer that specifies the address of a data list component

D --the value to be assigned to the data element of the list component

#### Remarks

The subroutine assumes that ADDRESS represents a valid address of a data list component. If ADDRESS is null, no assignment is made.

#### Other Programmer-Defined Procedures Required

None

#### Method

The value of D is converted, if necessary, to a character string. The leftmost character of the string is assigned to the data element of the specified element.

Figure 2.4A. Description of the SET\_DATA subroutine for assigning a value to the data element of a component in a data list

```
SET_DATA:
  PROCEDURE (ADDRESS, D);
DECLARE
  ADDRESS POINTER,
  D CHARACTER(1),
  1 COMPONENT BASED(ADDRESS),
  2 DATA CHARACTER(1),
  2 POINTER POINTER;
  IF
    ADDRESS = NULL
  THEN
    RETURN;
  ADDRESS->DATA = D;
END
  SET_DATA;
```

Figure 2.4B. Assigning a value to the data element of a component in a data list

Because the subroutine uses the address of a list component, the character value is assigned directly to the data element, and no search is made for the appropriate component from the beginning of the associated list. Later discussions present other methods for inserting data items into lists.

*SET\_POINTER Subroutine* Figures 2.5A and 2.5B present the SET\_POINTER subroutine. This procedure is similar to SET\_DATA (Figures 2.4A and 2.4B) except that it assigns an address value to the pointer element of a specified list component.

```

SET_POINTER:
    PROCEDURE (ADDRESS, P);
    DECLARE
        P POINTER,
        ADDRESS POINTER,
        1 COMPONENT BASED(ADDRESS),
        2 DATA CHARACTER(1),
        2 POINTER POINTER;
    IF
        ADDRESS = NULL
    THEN
        RETURN;
        ADDRESS->POINTER = P;
    END
    SET_POINTER;

```

Figure 2.5B. Assigning a value to the pointer element of a component in a data list

<b>SET_POINTER Subroutine</b>	
<b>Purpose</b>	To assign a value to the pointer element of a data list component
<b>Reference</b>	SET_POINTER (ADDRESS, P)
<b>Entry-Name Declaration</b>	DECLARE SET_POINTER ENTRY (POINTER, POINTER);
<b>Meaning of Arguments</b>	ADDRESS --a pointer value that specifies the address of a data list component P --the value to be assigned to the pointer element of the list component
<b>Remarks</b>	The subroutine assumes that ADDRESS represents a valid address of a data list component. If ADDRESS is null, no assignment is made.
<b>Other Programmer-Defined Procedures Required</b>	None
<b>Method</b>	The pointer value represented by P is assigned to the pointer element of the specified component.

Figure 2.5A. Description of the SET\_POINTER subroutine for assigning a value to the pointer element of a component in a data list

### *Obtaining the Values of Elements in List Components*

Many list-processing operations examine the values of the data and pointer elements in list components. The following discussions develop two function procedures that obtain these values:

1. GET\_DATA, which obtains the value of the data element in a data list component
2. GET\_POINTER, which obtains the value of the pointer element in a data list component

*GET\_DATA Function* Figures 2.6A and 2.6B present the GET\_DATA function, which uses the address of a list component as its argument. The function returns the character value of the data element in the specified list component.

*GET\_POINTER Function* Figures 2.7A and 2.7B present the GET\_POINTER function. This procedure is similar to GET\_DATA (Figures 2.6A and 2.6B) except that it obtains the value of the pointer element in a specified list component.

Note that GET\_POINTER obtains the same value as ADDRESS\_NEXT (Figures 2.3A and 2.3B). The reason for having two different names for the same function is to provide an application-oriented emphasis in later procedures. When the pointer value in a list component is treated as the address of the next component, it is convenient to think in terms of the identifier ADDRESS\_NEXT. On other occasions, when the pointer value is used for chaining purposes and the emphasis is not on the next list component, the identifier GET\_POINTER conveys a more accurate description of the intended effect. In either case, the two functions are interchangeable, since they produce the same result.

**GET\_DATA Function**

**Purpose**  
To obtain the value of the data element in a data list component

**Reference**  
GET\_DATA(ADDRESS)

**Entry-Name Declaration**  
DECLARE GET\_DATA ENTRY(POINTER)  
RETURNS(Character(1));

**Meaning of Argument**  
ADDRESS --a pointer value that specifies the address of a data list component

**Remarks**  
The function assumes that ADDRESS represents a valid address of a data list component. If ADDRESS is null, a blank character is returned.

**Other Programmer-Defined Procedures Required**  
None

**Method**  
The function returns the value of the data element in the specified component.  
  
The value is a single alphameric character.

Figure 2.6A. Description of the GET\_DATA function for obtaining the value of the data element of a component in a data list

```

GET_DATA:
  PROCEDURE (ADDRESS)
  RETURNS (CHARACTER(1));
DECLARE
  ADDRESS POINTER,
  1 COMPONENT BASED(ADDRESS),
  2 DATA CHARACTER(1),
  2 POINTER POINTER;
  IF
  ADDRESS = NULL
  THEN
  RETURN(' ');
  RETURN(ADDRESS->DATA);
END
  GET_DATA;

```

Figure 2.6B. Obtaining the value of the data element in a component of a data list

**GET\_POINTER Function**

**Purpose**  
To obtain the value of the pointer element in a data list component

**Reference**  
GET\_POINTER(ADDRESS)

**Entry-Name Declaration**  
DECLARE GET\_POINTER ENTRY(POINTER)  
RETURNS(POINTER);

**Meaning of Argument**  
ADDRESS --a pointer value that specifies the address of a data list component

**Remarks**  
The function assumes that ADDRESS represents a valid address of a data list component. If ADDRESS is null, a null pointer value is returned.

**Other Programmer-Defined Procedures Required**  
None

**Method**  
The function returns the value of the pointer element in the specified component.

Figure 2.7A. Description of the GET\_POINTER function for obtaining the value of the pointer element of a component in a data list

```

GET_POINTER:
  PROCEDURE (ADDRESS)
  RETURNS (POINTER);
DECLARE
  ADDRESS POINTER,
  1 COMPONENT BASED(ADDRESS),
  2 DATA CHARACTER(1),
  2 POINTER POINTER;
  IF
  ADDRESS = NULL
  THEN
  RETURN (NULL);
  RETURN(ADDRESS->POINTER);
END
  GET_POINTER;

```

Figure 2.7B. Obtaining the value of the pointer element in a component of a data list

### Examples

Figure 2.8 contains examples of references to the ADDRESS\_N, ADDRESS\_NEXT, SET\_DATA, and GET\_DATA procedures. GET\_POINTER and SET\_POINTER are not illustrated, because GET\_POINTER is equivalent to ADDRESS\_NEXT, and later discussions present a more appropriate opportunity for demonstrating the effect of SET\_POINTER.

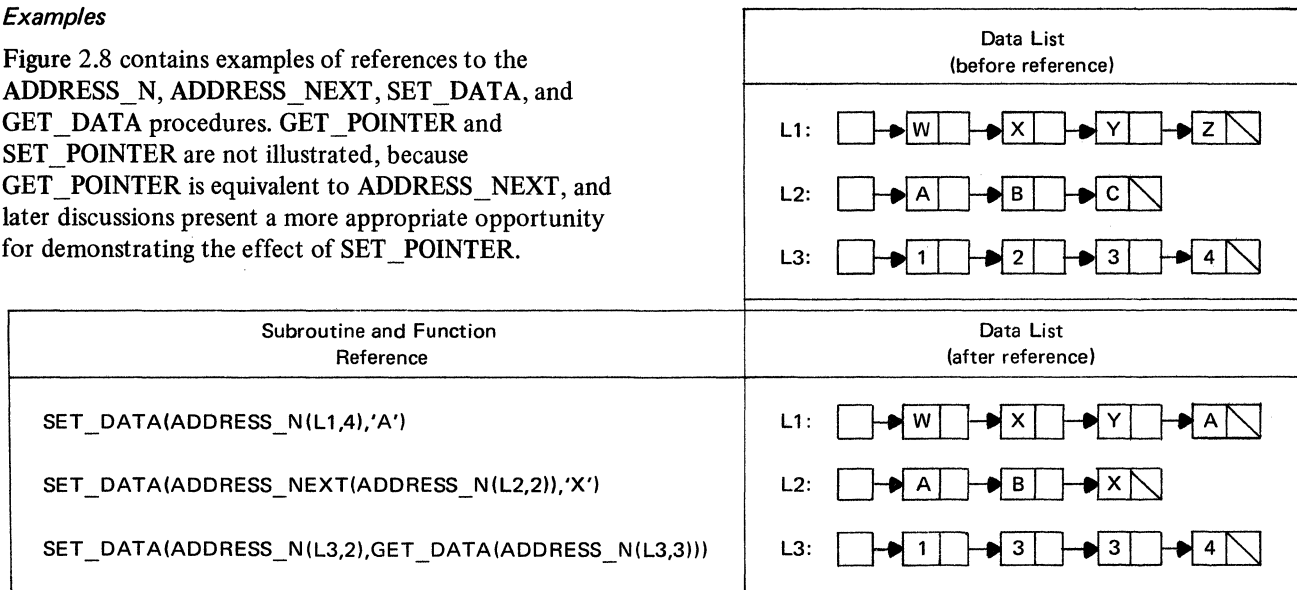


Figure 2.8. Examples of references to the ADDRESS\_N, ADDRESS\_NEXT, SET\_DATA, and GET\_DATA procedures

Note how the functions ADDRESS\_N, ADDRESS\_NEXT, and GET\_DATA appear as arguments of other procedures. Nesting function references in this manner is called *function composition* and is permitted to an arbitrary depth. Function composition forms an essential feature of the list-processing techniques developed in this manual; it produces compact references and reduces the need for intermediate variables in which to save function values.

### Manipulating List Components

The procedures in the previous section deal with list components individually; they manipulate the parts (elements) of list components but do not associate components with one another. The following discussions develop procedures for manipulating collections of list components. They also suppress much of the environmental detail underlying the structure of list organization and show how simple data lists may be viewed as sequences of data items rather than as sequences of storage components. As a result, the application-oriented aspects of list organization receive greater emphasis.

The operations performed by the procedures in this section fall into eight general categories:

1. Obtaining the size of a data list
2. Inserting data items into data lists
3. Getting the values of data items in data lists
4. Deleting data items from data lists
5. Removing data items from data lists (and at the same time getting the values of the items)
6. Replacing data items in data lists
7. Searching for data items in data lists
8. Interchanging data items in data lists

Most of the procedures in these categories are constructed from previously developed procedures. This method of construction shows how the procedures may be organized into an integrated collection.

#### Obtaining the Size of a Data List

A simple but common operation performed on data lists is counting the number of data items in a list. The following discussions develop two function procedures for obtaining the size of a list:

1. SIZE, which uses the ADDRESS\_NEXT subroutine (presented earlier in Figure 2.3B)
2. SIZE1, which does not use other list-processing procedures

The reason for presenting two versions of the same function is to show how a procedure may be treated either as a primitive routine that does not use other procedures or as a higher level routine that is constructed from one or more previously developed procedures.



*SIZE Function* Figures 2.9A, 2.9B, and 2.9C present the *SIZE* function. This function uses a pointer argument that forms the head of the list whose size is desired. The function returns a count of the data items in the specified list.

<b>SIZE Function</b>	
<b>Purpose</b>	To obtain the number of data items in a data list
<b>Reference</b>	SIZE(LIST)
<b>Entry-Name Declaration</b>	DECLARE SIZE ENTRY(POINTER) RETURNS(FIXED DECIMAL(5));
<b>Meaning of Argument</b>	LIST --the pointer variable that is the head of the list to be examined
<b>Remarks</b>	The maximum size is 99999. If LIST is null, a zero size is returned.
<b>Other Programmer-Defined Procedures Required</b>	ADDRESS_NEXT
<b>Method</b>	The function proceeds through LIST, counting the number of list components until a null pointer element is encountered.

Figure 2.9A. Description of the *SIZE* function for obtaining the number of data items in a data list

```

SIZE:
  PROCEDURE (LIST)
  RETURNS (FIXED DECIMAL(5));
  DECLARE
    (LIST,ADDRESS) POINTER,
    N FIXED DECIMAL(5);
    ADDRESS = LIST;
  DO
    N = 0 BY 1;
  IF
    ADDRESS = NULL
  THEN
    RETURN(N);
  ELSE
    ADDRESS = ADDRESS_NEXT(ADDRESS);
  END;
  END
  SIZE;

```

Figure 2.9B. Obtaining the size of a data list

The *ADDRESS\_NEXT* procedure is used to proceed through a list until a null pointer element is detected.

*SIZE1 Function* Figures 2.9D and 2.9E present the *SIZE1* function, which produces the same result as the *SIZE* function (Figures 2.9A through 2.9C). The main difference between the two functions is that *SIZE1* uses no other list-processing procedures; it is constructed as a primitive routine that uses pointer qualification of a based variable rather than a function reference to proceed through a specified list.

*Inserting Data Items into Data Lists*

So far, list-processing procedures have assumed the existence of data lists and have not shown how list components are attached to specified lists. When a data item is inserted into a list, a storage component must be obtained from the list of available storage components (*AVAIL*) and linked to the list that is to contain the inserted item. Without this storage component, no storage would be available for the new item.

	Data List	Function Reference	Function Value
L1:		SIZE(L1)	5
L2:		SIZE(L2)	3
L3:		SIZE(L3)	1
L4:		SIZE(L4)	0

Figure 2.9C. Examples of references to the *SIZE* function

## SIZE1 Function

### Purpose

To obtain the number of data items in a data list

### Reference

SIZE1(LIST)

### Entry-Name Declaration

```
DECLARE SIZE1 ENTRY(POINTER)
        RETURNS(FIXED DECIMAL(5));
```

### Meaning of Argument

LIST --the pointer variable that is the head of the list to be examined

### Remarks

The maximum size is 99999. If LIST is null, a zero size is returned.

### Other Programmer-Defined Procedures Required

None

### Method

The function proceeds through LIST, counting the number of list components until a null pointer element is encountered.

The following discussions develop four subroutine procedures that insert data items into specified list positions:

1. INSERT\_ND, which inserts a data item into the nth position of a data list
2. INSERT\_FD, which inserts a data item into the first position of a data list
3. INSERT\_LD, which inserts a data item into the last position of a data list
4. INSERT\_ND1, which produces the same results as INSERT\_ND but does not use other list-processing procedures

*INSERT\_ND Subroutine* Figures 2.10A and 2.10B present the INSERT\_ND subroutine procedure, and examples of the procedure appear in Figure 2.10G. INSERT\_ND uses three arguments: a data list, a position within the list, and a data item to be inserted at that position in the list.

Insertion of a data item into a list increases the size of the list by one and at the same time decreases the size of the list of available storage components (AVAIL) by one. The size of AVAIL can also be zero when INSERT\_ND is invoked; in this case, no data is inserted into the specified list, but a message is printed, and control is returned to the point of invocation. In a more elaborate system of procedures, attempted use of AVAIL when it is null can transfer control to an error procedure where special action can be taken, such as allocating additional storage for AVAIL.

*INSERT\_FD Subroutine* Figures 2.10C and 2.10D present the INSERT\_FD subroutine procedure, which inserts a data item into the first position of a data list. This procedure is similar to INSERT\_ND (Figures 2.10A and 2.10B) but does not require an argument for the insertion position, since the first position is always implied.

Although INSERT\_FD is not essential when INSERT\_ND is available, it is convenient to use when insertions frequently occur at the front of lists.

Figure 2.10G contains examples of the procedure.

*INSERT\_LD Subroutine* Figures 2.10E and 2.10F present the INSERT\_LD subroutine procedure, which inserts a data item into the last position of a data list. This procedure is similar to INSERT\_FD (Figure 2.10C and 2.10D) but deals with the last rather than the first position of a list.

Figure 2.10G contains examples of the procedure.

*INSERT\_ND1 Subroutine* Figures 2.10H and 2.10I present the INSERT\_ND1 procedure, which produces the same results as INSERT\_ND (Figures 2.10A and 2.10B). The main difference between the two procedures is that INSERT\_ND1 does not use other list-processing procedures.

Figure 2.9D. Description of the alternate function SIZE1 for obtaining the number of data items in a data list

```
SIZE1:
  PROCEDURE (LIST)
  RETURNS (FIXED DECIMAL(5));
DECLARE
  LIST POINTER,
  ADDRESS POINTER,
  N FIXED DECIMAL(5),
  1 COMPONENT BASED(ADDRESS),
  2 DATA CHARACTER(1),
  2 POINTER POINTER;
  ADDRESS = LIST;
DO
  N = 0 BY 1;
  IF
  ADDRESS = NULL
  THEN
  RETURN(N);
  ELSE
  ADDRESS = ADDRESS->POINTER;
END;
END
  SIZE1;
```

Figure 2.9E. An alternative function for obtaining the size of a data list

**INSERT\_ND Subroutine**

**Purpose**  
To insert a data item into the nth position of a data list

**Reference**  
INSERT\_ND(LIST, N, D)

**Entry-Name Declaration**  
DECLARE INSERT\_ND  
ENTRY(P POINTER, FIXED DECIMAL(5), CHARACTER(1));

**Meaning of Arguments**  
LIST --the pointer variable that is the head of the list to be processed  
N --the position in the list where the data item is to be inserted  
D --the data item to be inserted

**Remarks**  
When the list is null or N is less than two, the data item is inserted into the first position of the list. When N exceeds the size of the list, the data item is inserted into the last position. N cannot have a value greater than 99999.

**Other Programmer-Defined Procedures Required**  
ADDRESS\_N, SET\_DATA, SET\_POINTER, and SIZE

**Method**  
This subroutine does not destroy data previously in the list. When an item is inserted, the size of the list increases by one. The item previously in the nth position becomes the (n+1)th item. The value of D is converted, if necessary, to a character string, and the leftmost character of the string is inserted into the list.

Figure 2.10A. Description of the INSERT\_ND subroutine for inserting a data item into the nth position of a data list

```

INSERT_ND:
  PROCEDURE(LIST,N,D);
  DECLARE
    N FIXED DECIMAL(5),
    D CHARACTER(1),
    (P,Q) POINTER,
    (LIST, ADDRESS1, ADDRESS2, AVAIL
     EXTERNAL) POINTER;
    /* IF LIST OF AVAILABLE STORAGE
     COMPONENTS IS EMPTY THEN PRINT
     MESSAGE AND RETURN. */
  IF
    AVAIL = NULL
  THEN
  DO;
    PUT
      LIST('LIST OF AVAILABLE STORAGE IS
        EMPTY');
    RETURN;
  END;
  /* ASSIGN DATA ITEM TO FIRST
  COMPONENT IN LIST OF AVAILABLE
  STORAGE. */
  CALL SET_DATA(AVAIL, D);
  /* IF LIST IS NULL OR N<2, INSERT
  FIRST COMPONENT OF AVAIL INTO FIRST
  POSITION OF LIST AND RETURN. */
  IF
    (LIST = NULL)|(N<2)
  THEN
  DO;
    ADDRESS1 = LIST; LIST = AVAIL;
    AVAIL = ADDRESS_N(AVAIL,2);
    CALL SET_POINTER(LIST,ADDRESS1);
    RETURN;
  END;
  IF
    N > SIZE(LIST)
  THEN
  DO;
    P = LIST;
  DO
    WHILE(P ≠ NULL);
    Q = P; P = Q->POINTER;
  END;
  P, Q->POINTER = AVAIL;
  AVAIL = ADDRESS_N(AVAIL, 2);
  P -> POINTER = NULL;
  RETURN;
  END;
  /* OTHERWISE OBTAIN THE ADDRESS OF
  THE N-TH COMPONENT OF LIST. */
  ADDRESS2 = ADDRESS_N(LIST,N);
  ADDRESS1 = ADDRESS_N(LIST,N-1);
  /* INSERT FIRST COMPONENT OF AVAIL
  INTO THE N-TH POSITION OF LIST. */
  CALL SET_POINTER(ADDRESS1, AVAIL);
  ADDRESS1 = AVAIL;
  AVAIL = ADDRESS_N(AVAIL,2);
  CALL SET_POINTER(ADDRESS1,ADDRESS2);
  END
  INSERT_ND;

```

Figure 2.10B. Inserting a data item into the nth position of a data list

**INSERT\_FD Subroutine**

**Purpose**  
To insert a data item into the first position of a data list

**Reference**  
INSERT\_FD(LIST, D)

**Entry-Name Declaration**  
DECLARE INSERT\_FD  
ENTRY(POINTER, CHARACTER(1));

**Meaning of Arguments**  
LIST --the pointer variable that is the head of the list to be processed  
D --the data item to be inserted

**Remarks**  
When the list is null, the data item is inserted into the first position of the list.

**Other Programmer-Defined Procedures Required**  
INSERT\_ND

**Method**  
This subroutine uses the INSERT\_ND procedure with N equal to one:  
CALL INSERT\_ND(LIST, 1, D);

Figure 2.10C. Description of the INSERT\_FD subroutine for inserting a data item into the first position of a data list

**INSERT\_LD Subroutine**

**Purpose**  
To insert a data item into the last position of a data list

**Reference**  
INSERT\_LD(LIST, D)

**Entry-Name Declaration**  
DECLARE INSERT\_LD  
ENTRY(POINTER, CHARACTER(1));

**Meaning of Arguments**  
LIST --the pointer variable that is the head of the list to be processed  
D --the data item to be inserted

**Remarks**  
When the list is null, the data item is inserted into the first position of the list.

**Other Programmer-Defined Procedures Required**  
INSERT\_ND

**Method**  
This subroutine uses the INSERT\_ND procedure with N equal to 99999, which forces the item to be inserted into the last position of the list:  
  
CALL INSERT\_ND(LIST, 99999, D);

Figure 2.10E. Description of the INSERT\_LD subroutine for inserting a data item into the last position of a data list

```

INSERT_FD:
  PROCEDURE (LIST,D);
  DECLARE
    LIST POINTER,
    D CHARACTER(1);
    CALL INSERT_ND(LIST,1,D);
  END
  INSERT_FD;

```

Figure 2.10D. Inserting a data item into the first position of a data list

```

INSERT_LD:
  PROCEDURE (LIST,D);
  DECLARE
    LIST POINTER,
    D CHARACTER(1);
    CALL INSERT_ND(LIST,99999,D);
  END
  INSERT_LD;

```

Figure 2.10F. Inserting a data item into the last position of a data list

Data List (before reference)	Subroutine Reference	Data List (after reference)
L1:	INSERT_ND(L1,1,'A')	L1:
L1:	INSERT_ND(L1,2,'A')	L1:
L1:	INSERT_ND(L1,3,'A')	L1:
L1:	INSERT_ND(L1,9,'A')	L1:
L1:	INSERT_ND(L1,-3,'A')	L1:
L1:	INSERT_FD(L1,'A')	L1:
L1:	INSERT_LD(L1,'A')	L1:
L2:	INSERT_ND(L2,1,'6')	L2:
L2:	INSERT_ND(L2,2,'6')	L2:
L2:	INSERT_FD(L2,'*')	L2:
L2:	INSERT_LD(L2,'*')	L2:

Figure 2.10G. Examples of references to the INSERT\_ND, INSERT\_FD, and INSERT\_LD subroutines

## INSERT\_ND1 Subroutine

### Purpose

To insert a data item into the nth position of a data list

### Reference

INSERT\_ND1(LIST, N, D)

### Entry-Name Declaration

```
DECLARE INSERT_ND1
  ENTRY(POINTER, FIXED DECIMAL(5),
  CHARACTER(1));
```

### Meaning of Arguments

LIST --the pointer variable that is the head of the list to be processed

N --the position in the list where the data item is to be inserted

D --the data item to be inserted

### Remarks

When the list is null or N is less than two, the data item is inserted into the first position of the list. When N exceeds the size of the list, the data item is inserted into the last position. N cannot have a value greater than 99999.

### Other Programmer-Defined Procedures Required

None

### Method

This subroutine does not destroy data previously in the list. When an item is inserted, the size of the list increases by one. The item previously in the nth position becomes the (n+1)th item. The value of D is converted, if necessary, to a character string, and the leftmost character of the string is inserted into the list.

```
INSERT_ND1:
  PROCFDURF(LIST,N,D);
  DECLARE
    (N,I) FIXED DECIMAL(5),
    D CHARACTER(1),
    (LIST, ADDRESS1, AVAIL EXTERNAL)
    POINTER,
    ADDRESS2 POINTER,
    1 COMPONENT BASED(ADDRESS2),
    2 DATA CHARACTER(1),
    2 POINTER POINTER;
    /* IF LIST OF AVAIL2 STORAGE
    COMPONENTS IS EMPTY THEN PRINT
    MESSAGE AND RETURN. */
  IF
    AVAIL = NULL
  THEN
  DO;
    PUT
      LIST('AVAIL STORAGE IS EMPTY');
    RETURN;
  END;
  /* ASSIGN DATA ITEM TO FIRST
  COMPONENT IN LIST OF AVAIL
  STORAGE. */
  AVAIL->DATA = D;
  /* IF LIST IS NULL OR N<2, INSERT
  FIRST COMPONENT OF AVAIL INTO
  FIRST POSITION OF LIST AND
  RETURN. */
  IF
    (LIST = NULL)|(N<2)
  THEN
  DO;
    ADDRESS1 = LIST; LIST = AVAIL;
    AVAIL = AVAIL->POINTER;
    LIST->POINTER = ADDRESS1;
    RETURN;
  END;
  /* OTHERWISE, OBTAIN THE ADDRESS OF
  THE N-TH COMPONENT OF LIST. */
  ADDRESS2 = LIST; I = 2;
  DO
    WHILE((ADDRESS2->POINTER<=>NULL) &
    (I<N));
    ADDRESS2 = ADDRESS2->POINTER;
    I = I + 1;
  END;
  /* INSERT FIRST COMPONENT OF AVAIL
  INTO THE N-TH POSITION OF LIST. */
  ADDRESS1 = ADDRESS2->POINTER;
  ADDRESS2->POINTER = AVAIL;
  AVAIL = AVAIL->POINTER;
  ADDRESS2 = ADDRESS2->POINTER;
  ADDRESS2->POINTER = ADDRESS1;
  END
  INSERT_ND1;
```

Figure 2.10H. Description of the alternative subroutine INSERT\_ND1

Figure 2.10I. An alternative subroutine for inserting a data item into the nth position of a data list

### Getting the Values of Data Items in Data Lists

Once a data item has been inserted into a list, a common operation is to retrieve the item from the list. The following discussions develop four function procedures for obtaining the values of data items at specified list positions:

1. **GET\_ND**, which gets the value of the data item in the *n*th position of a data list
2. **GET\_FD**, which gets the value of the data item in the first position of a data list
3. **GET\_LD**, which gets the value of the data item in the last position of a data list
4. **GET\_ND1**, which produces the same results as **GET\_ND** but does not use other list-processing procedures

**GET\_ND Function** Figures 2.11A and 2.11B present the **GET\_ND** function procedure, which gets the value of the data item in the *n*th position of a data list. Examples of references to the procedure appear in Figure 2.11G.

**GET\_ND** uses two arguments: a data list and the position of a data item in the list. The data item remains in the list after the value of the item is returned by **GET\_ND**.

**GET\_FD Function** Figures 2.11C and 2.11D present the **GET\_FD** function procedure, which gets the value of the data item in the first position of a data list. Figure 2.11G contains examples of the procedure.

**GET\_FD** is similar to **GET\_ND** (Figures 2.11A and 2.11B) but does not require an argument to specify the item position, because the first position is always implied.

**GET\_LD Function** Figures 2.11E and 2.11F present the **GET\_LD** function procedure, which gets the value of the data item in the last position of a data list. Examples of the procedure appear in Figure 2.11G.

**GET\_LD** is similar to **GET\_FD** (Figures 2.11C and 2.11D) but deals with the last rather than the first position of a list.

**GET\_ND1 Function** Figures 2.11H and 2.11I present the **GET\_ND1** function procedure, which produces the same results as **GET\_ND** (Figure 2.11A and 2.11B). The main difference between the two procedures is that **GET\_ND1** does not use other list-processing procedures.

#### GET\_ND Function

##### Purpose

To get the value of the data item in the *n*th position of a data list

##### Reference

**GET\_ND**(LIST, N)

##### Entry-Name Declaration

```
DECLARE GET_ND ENTRY(POINTER,  
FIXED DECIMAL(5))  
RETURNS(Character(1));
```

##### Meaning of Arguments

**LIST** --the pointer variable that is the head of the list to be processed

**N** --the position of the data item whose value is to be obtained

##### Remarks

A value of *N* less than one or greater than the number of data items in the list causes a blank character to be returned.

##### Other Programmer-Defined Procedures Required

**ADDRESS\_N** and **GET\_DATA**

##### Method

The following reference obtains the value of the *n*th item:

```
GET_DATA(ADDRESS_N(LIST, N))
```

The *n*th item remains in the list after its value is returned.

Figure 2.11A. Description of the **GET\_ND** function for getting the data item in the *n*th position of a data list

```

GET_ND:
  PROCEDURE (LIST, N)
  RETURNS (CHARACTER(1));
DECLARE
  LIST POINTER,
  N FIXED DECIMAL(5);
  RETURN(GET_DATA(ADDRESS_N(LIST,N)));
END
  GET_ND;

```

Figure 2.11B. Getting the data item in the nth position of a data list

```

GET_FD:
  PROCEDURE (LIST)
  RETURNS (CHARACTER(1));
DECLARE
  LIST POINTER;
  RETURN(GET_DATA(LIST));
END
  GET_FD;

```

Figure 2.11D. Getting the data item in the first position of a data list

**GET\_FD Function**

**Purpose**  
To get the value of the data item in the first position of a data list

**Reference**  
GET\_FD(LIST)

**Entry-Name Declaration**  
DECLARE GET\_FD ENTRY(POINTER)  
RETURNS(CHARACTER(1));

**Meaning of Argument**  
LIST --the pointer variable that is the head of the list to be processed

**Remarks**  
A null value for LIST causes a blank character to be returned.

**Other Programmer-Defined Procedures Required**  
GET\_DATA

**Method**  
The following reference obtains the value of the first item:

GET\_DATA(LIST)

The first item remains in the list after its value is returned.

Figure 2.11C. Description of the GET\_FD function for getting the data item in the first position of a data list

**GET\_LD Function**

**Purpose**  
To get the value of the data item in the last position of a data list

**Reference**  
GET\_LD(LIST)

**Entry-Name Declaration**  
DECLARE GET\_LD ENTRY(POINTER)  
RETURNS(CHARACTER(1));

**Meaning of Argument**  
LIST --the pointer variable that is the head of the list to be processed

**Remarks**  
A null value for LIST causes a blank character to be returned.

**Other Programmer-Defined Procedures Required**  
GET\_DATA and ADDRESS\_NEXT

**Method**  
The ADDRESS\_NEXT function is used to progress through the list to the last component. The GET\_DATA function then obtains the data value in the last component. The last item remains in the list after its value is returned.

Figure 2.11E. Description of the GET\_LD function for getting the data item in the last position of a data list



```

GET_LD:
  PROCEDURE (LIST)
  RETURNS (CHARACTER(1));
  DECLARE
    (LIST, ADDRESS1, ADDRESS2) POINTER;
    ADDRESS1, ADDRESS2 = LIST;
  DO
    WHILE(ADDRESS2≠NULL);
    ADDRESS1 = ADDRESS2;
    ADDRESS2 = ADDRESS_NEXT(ADDRESS2);
  END;
  RETURN(GET_DATA(ADDRESS1));
END
  GET_LD;

```

Figure 2.11F. Getting the data item in the last position of a data list

Data List	Function Reference	Function Value
L1:	GET_ND(L1,1)	'A'
L1:	GET_ND(L1,5)	','
L1:	GET_ND(L1,3)	'B'
L1:	GET_ND(L1,0)	'␣'
L1:	GET_ND(L1,9)	'␣'
L1:	GET_FD(L1)	'A'
L1:	GET_LD(L1)	','
L2:	GET_ND(L2,1)	'␣'
L2:	GET_ND(L2,0)	'␣'
L2:	GET_FD(L2)	'␣'
L2:	GET_LD(L2)	'␣'

Figure 2.11G. Examples of references to the GET\_ND, GET\_FD, and GET\_LD functions

<b>GET_ND1 Function</b>	
<b>Purpose</b>	To get the value of the data item in the nth position of a data list
<b>Reference</b>	GET_ND1(LIST, N)
<b>Entry-Name Declaration</b>	<pre> DECLARE GET_ND1 ENTRY(POINTER, FIXED DECIMAL(5)) RETURNS(CHARACTER(1)); </pre>
<b>Meaning of Arguments</b>	<p>LIST --the pointer variable that is the head of the list to be processed</p> <p>N --the position of the data item whose value is to be obtained</p>
<b>Remarks</b>	A value of N less than one or greater than the number of data items in the list causes a blank character to be returned.
<b>Other Programmer-Defined Procedures Required</b>	None
<b>Method</b>	The nth data item remains in the list after its value is returned.

Figure 2.11H. Description of the alternative function GET\_ND1 for getting the data item in the nth position of a data list

```

GET_ND1:
    PROCEDURE (LIST, N)
    RETURNS (CHARACTER(1));
DECLARE
    LIST POINTER,
    ADDRESS POINTER,
    (N,1) FIXED DECIMAL(5),
    1 COMPONENT BASED(ADDRESS),
    2 DATA CHARACTER(1),
    2 POINTER POINTER;
    IF
        (LIST = NULL) | (N < 1)
    THEN
        RETURN(' ');
        ADDRESS = LIST;
DO
    I = 1 BY 1;
    IF
        (ADDRESS->POINTER = NULL) &
        (I=N)
    THEN
        RETURN(' ');
    IF
        I = N
    THEN
        RETURN(ADDRESS->DATA);
        ADDRESS = ADDRESS->POINTER;
END;
END
    GET_ND1;

```

Figure 2.11I. An alternative method for obtaining the value of the nth data item in a data list

#### *Deleting Data Items from Data Lists*

After a data item has been inserted into a list, it may be necessary to delete the item from the list. The following discussions develop four subroutine procedures for deleting data items from specified list positions:

1. DELETE\_ND, which deletes the data item in the nth position of a data list
2. DELETE\_FD, which deletes the data item in the first position of a data list
3. DELETE\_LD, which deletes the data item in the last position of a data list
4. DELETE\_ND1, which is equivalent to DELETE\_ND but does not use other list-processing procedures

These procedures return the deleted storage to the list of available storage components, AVAIL.

*DELETE\_ND Subroutine* Figures 2.12A and 2.12B present the DELETE\_ND subroutine procedure, which deletes the data item in the nth position of a data list. Examples of the procedure appear in Figure 2.12G.

### DELETE\_ND Subroutine

#### Purpose

To delete the data item in the nth position of a data list

#### Reference

DELETE\_ND(LIST, N)

#### Entry-Name Declaration

```
DECLARE DELETE_ND ENTRY(POINTER,  
FIXED DECIMAL(5));
```

#### Meaning of Arguments

LIST --the pointer variable that is the head of the list to be processed  
N --the position of the data item to be deleted

#### Remarks

No data item is deleted when the value of N is less than one or greater than the number of items in the list.

#### Other Programmer-Defined Procedures Required

ADDRESS\_N and SET\_POINTER

#### Method

The nth component of LIST is deleted from LIST and inserted in the list of available storage components, AVAIL. The size of LIST is decreased by one, and that of AVAIL is increased by one.

Figure 2.12A. Description of the DELETE\_ND subroutine for deleting the data item in the nth position of a data list

```
DELETE_ND:  
  PROCEDURE(LIST,N);  
  DECLARE  
    N FIXED DECIMAL(5),  
    (LIST,ADDRESS1,ADDRESS2,ADDRESS3,  
     AVAIL EXTERNAL) POINTER;  
    /* IF LIST IS EMPTY OR N IS LESS  
     THAN 1, THEN RETURN. */  
  IF  
    (LIST = NULL)|(N<1)  
  THEN  
    RETURN;  
    /* DELETE FIRST COMPONENT IF N  
    EQUALS 1. */  
  IF  
    N = 1  
  THEN  
    DO;  
      ADDRESS2 = LIST;  
      LIST = ADDRESS_N(LIST,2);  
      GO TO  
      L;  
    END;  
    /* OBTAIN N-TH COMPONENT. */  
    ADDRESS2 = ADDRESS_N(LIST,N);  
    IF  
      ADDRESS2 = NULL  
    THEN  
      RETURN;  
      ADDRESS1 = ADDRESS_N(LIST,N-1);  
      ADDRESS3 = ADDRESS_N(LIST,N+1);  
      /* DELETE N-TH COMPONENT. */  
      CALL SET_POINTER(ADDRESS1,ADDRESS3);  
      /* INSERT DELETED COMPONENT INTO  
      LIST OF AVAILABLE STORAGE  
      COMPONENTS. */  
    L:  
      ADDRESS1 = AVAIL;  
      AVAIL = ADDRESS2;  
      CALL SET_POINTER(AVAIL, ADDRESS1);  
    END  
    DELETE_ND;
```

Figure 2.12B. Deleting the data item in the nth position of a data list

DELETE\_ND uses two arguments: a data list and the position of a data item in the list. The size of the list decreases by one after deletion occurs, and the list of available storage components, AVAIL, acquires the storage component used by the deleted item.

*DELETE\_FD Subroutine* Figures 2.12C and 2.12D present the DELETE\_FD subroutine procedure, which deletes the data item in the first position of a data list. Figures 2.12G contains examples of the procedure.

DELETE\_FD is similar to DELETE\_ND (Figures 2.12A and 2.12B) but does not use an argument to specify the item position, since the first position is always implied.

<b>DELETE_FD Subroutine</b>	
<b>Purpose</b>	To delete the data item in the first position of a data list
<b>Reference</b>	DELETE_FD(LIST)
<b>Entry-Name Declaration</b>	DECLARE DELETE_FD ENTRY(POINTER);
<b>Meaning of Argument</b>	LIST --the pointer variable that is the head of the list to be processed
<b>Remarks</b>	No data item is deleted when LIST is null.
<b>Other Programmer-Defined Procedures Required</b>	DELETE_ND
<b>Method</b>	The following reference is used:  DELETE_ND(LIST, 1)

Figure 2.12C. Description of the DELETE\_FD subroutine for deleting the data item in the first position of a data list

*DELETE\_LD Subroutine* Figures 2.12E and 2.12F present the DELETE\_LD subroutine procedure, which deletes the data item in the last position of a data list. This procedure is similar to DELETE\_FD (Figures 2.12C and 2.12D) but deals with the last rather than the first position of a list.

Figure 2.12G contains an example of reference to the procedure.

<b>DELETE_LD Subroutine</b>	
<b>Purpose</b>	To delete the data item in the last position of a data list
<b>Reference</b>	DELETE_LD(LIST)
<b>Entry-Name Declaration</b>	DECLARE DELETE_LD ENTRY(POINTER);
<b>Meaning of Argument</b>	LIST --the pointer variable that is the head of the list to be processed
<b>Remarks</b>	No data item is deleted when LIST is null.
<b>Other Programmer-Defined Procedures Required</b>	DELETE_ND and SIZE
<b>Method</b>	The following reference is used:  DELETE_ND(LIST, SIZE(LIST))

Figure 2.12E. Description of the DELETE\_LD subroutine for deleting the data item in the last position of a data list

```
DELETE_FD:
  PROCEDURE(LIST);
DECLARE
  LIST POINTER;
  CALL DELETE_ND(LIST,1);
END
  DELETE_FD;
```

Figure 2.12D. Deleting the data item in the first position of a data list

```
DELETE_LD:
  PROCEDURE(LIST);
DECLARE
  LIST POINTER;
  CALL DELETE_ND(LIST, SIZE(LIST));
END
  DELETE_LD;
```

Figure 2.12F. Deleting the data item in the last position of a data list


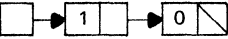
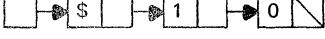
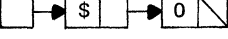
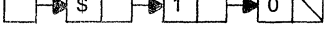
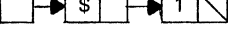


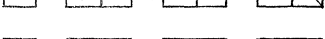
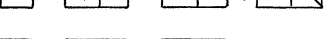
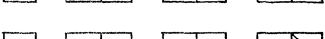
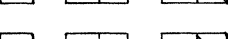










Data List (before reference)	Subroutine Reference	Data List (after reference)
L1: 	DELETE_ND(L1,1)	L1: 
L1: 	DELETE_ND(L1,2)	L1: 
L1: 	DELETE_ND(L1,3)	L1: 
L1: 	DELETE_ND(L1,5)	L1: 
L1: 	DELETE_ND(L1,-2)	L1: 
L1: 	DELETE_FD(L1)	L1: 
L1: 	DELETE_LD(L1)	L1: 
L2: 	DELETE_ND(L2,1)	L2: 
L2: 	DELETE_ND(L2,0)	L2: 
L2: 	DELETE_FD(L2)	L2: 
L2: 	DELETE_LD(L2)	L2: 

Figure 2.12G. Examples of references to the DELETE\_ND, DELETE\_FD, and DELETE\_LD subroutines

*DELETE\_ND1 Subroutine* Figures 2.12H and 2.12I present the DELETE\_ND1 subroutine procedure, which produces the same results as DELETE\_ND (Figures 2.12A and 2.12B). The main difference between the two procedures is that DELETE\_ND1 does not use other list-processing procedures.

<b>DELETE_ND1 Subroutine</b>	
<b>Purpose</b>	To delete the data item in the nth position of a data list
<b>Reference</b>	DELETE_ND1(LIST, N)
<b>Entry-Name Declaration</b>	DECLARE DELETE_ND1 ENTRY(POINTER, FIXED DECIMAL(5));
<b>Meaning of Arguments</b>	LIST --the pointer variable that is the head of the list to be processed N --the position of the data item to be deleted
<b>Remarks</b>	No data item is deleted when the value of N is less than one or greater than the number of items in the list.
<b>Other Programmer-Defined Procedures Required</b>	None
<b>Method</b>	The nth component of LIST is deleted from LIST and inserted in the list of available storage components, AVAIL. The size of LIST is decreased by one, and size of AVAIL is increased by one.

Figure 2.12H. Description of the alternative subroutine DELETE\_ND1 for deleting the data item in the nth position of a data list

```

DELETE_ND1:
  PROCEDURE(LIST,N);
  DECLARE
    (N,I) FIXED DECIMAL(5),
    (LIST,ADDRESS1,AVAIL EXTERNAL)
    POINTER,
    ADDRESS2 POINTER,
    1 COMPONENT BASED(ADDRESS2),
    2 DATA CHARACTER(1),
    2 POINTER POINTER;
    /* IF LIST IS EMPTY OR N IS LESS
    THAN 1, THEN RETURN. */
  IF
    (LIST = NULL)|(N<1)
  THEN
    RETURN;
    /* DELETE FIRST COMPONENT IF N
    EQUALS 1. */
    ADDRESS2 = LIST;
  IF
    N = 1
  THEN
    DO;
      LIST = ADDRESS2->POINTER;
    GO TO
    L;
  END;
    /* OBTAIN N-TH COMPONENT. */
    ADDRESS1 = LIST; I = 1;
    DO WHILE!((ADDRESS2->POINTER~=NULL)
    &(I<N));
      ADDRESS1 = ADDRESS2;
      ADDRESS2 = ADDRESS2->POINTER;
      I = I + 1;
  END;
  IF
    (ADDRESS2->POINTER = NULL) &
    (I~=N)
  THEN
    RETURN;
    /* DELETE N-TH COMPONENT. */
    ADDRESS1->POINTER =
    ADDRESS2->POINTER;
    /* INSERT DELETED COMPONENT INTO
    LIST OF AVAILABLE STORAGE
    COMPONENTS. */
  L:
    ADDRESS1 = AVAIL;
    AVAIL = ADDRESS2;
    ADDRESS2->POINTER = ADDRESS1;
  END
  DELETE_ND1;

```

Figure 2.12I. An alternative subroutine for deleting the data item in the nth position of a data list

### Removing Data Items from Data Lists

A frequent combination of operations on a list involves getting the value of a data item and then deleting the item from the list. This combination is referred to as *removing* a data item from a list and may be implemented as a single function procedure. The following discussions develop three functions for removing data items from specified list positions:

1. REMOVE\_ND, which gets and deletes the data item in the nth position of a data list
2. REMOVE\_FD, which gets and deletes the data item in the first position of a data list
3. REMOVE\_LD, which gets and deletes the data item in the last position of a data list

**REMOVE\_ND Function** Figures 2.13A and 2.13B present the REMOVE\_ND function procedure, which gets and deletes the data item in the nth position of a data list. Examples of the procedure appear in Figure 2.13G.

<b>REMOVE_ND Function</b>
<b>Purpose</b> To get and delete the data item in the nth position of a data list
<b>Reference</b> REMOVE_ND(LIST, N)
<b>Entry-Name Declaration</b> DECLARE REMOVE_ND ENTRY(POINTER, FIXED DECIMAL(5)) RETURNS(CHARACTER(1));
<b>Meaning of Arguments</b> LIST --the pointer variable that is the head of the list to be processed N --the position of the data item to be removed
<b>Remarks</b> When the value of N is less than one or greater than the number of data items in the list, no data item is deleted from the list, and a blank character is returned.
<b>Other Programmer-Defined Procedures Required</b> GET_ND and DELETE_ND
<b>Method</b> This function combines the operations of the GET_ND and DELETE_ND procedures.

Figure 2.13A. Description of the REMOVE\_ND function for removing the data item in the nth position of a data list

```
REMOVE_ND:
  PROCEDURE (LIST, N)
  RETURNS (CHARACTER(1));
DECLARE
  LIST POINTER,
  N FIXED DECIMAL(5),
  D CHARACTER(1);
  D = GET_ND(LIST,N);
  CALL DELETE_ND(LIST,N);
  RETURN(D);
END
REMOVE_ND;
```

Figure 2.13B. Removing the data item in the nth position of a data list

REMOVE\_ND uses two arguments: a data list and the position of a data item in the list. The size of the list decreases by one after the item is deleted and its value returned to the point of invocation. The list of available storage components, AVAIL, acquires the storage component used by the removed item.

**REMOVE\_FD Function** Figures 2.13C and 2.13D present the REMOVE\_FD function procedure, which gets and deletes the data item in the first position of a data list. Figure 2.13G contains examples of the procedure.

REMOVE\_FD is similar to REMOVE\_ND (Figures 2.13A and 2.13B) but does not require an argument to specify the item position, since the first position is always implied.

<b>REMOVE_FD Function</b>
<b>Purpose</b> To get and delete the data item in the first position of a data list
<b>Reference</b> REMOVE_FD(LIST)
<b>Entry-Name Declaration</b> DECLARE REMOVE_FD ENTRY(POINTER) RETURNS(CHARACTER(1));
<b>Meaning of Argument</b> LIST --the pointer variable that is the head of the list to be processed
<b>Remarks</b> When LIST is null, no data item is deleted from the list, and a blank character is returned.
<b>Other Programmer-Defined Procedures Required</b> REMOVE_ND

Figure 2.13C. Description of the REMOVE\_FD function for removing the data item in the first position of a data list

```

REMOVE_FD:
  PROCEDURE (LIST)
  RETURNS (CHARACTER(1));
DECLARE
  LIST POINTER;
  RETURN(REMOVE_ND(LIST,1));
END
  REMOVE_FD;

```

Figure 2.13D. Removing the data item in the first position of a data list

*REMOVE\_LD Function* Figures 2.13E and 2.13F present the REMOVE\_LD function procedure, which gets and deletes the data item in the last position of a data list. This procedure is similar to REMOVE\_FD (Figures 2.13C and 2.13D) but deals with the last rather than the first position of a list.

Examples of the procedure appear in Figure 2.13G.

#### *Replacing Data Items in Data Lists*

Replacing a data item in a data list involves changing an item already in the list. The following discussions develop three subroutine procedures for replacing data items in specified list positions:

1. REPLACE\_ND, which replaces the data item in the nth position of a data list
2. REPLACE\_FD, which replaces the data item in the first position of a data list
3. REPLACE\_LD, which replaces the data item in the last position of a data list

*REPLACE\_ND Subroutine* Figures 2.14A and 2.14B present the REPLACE\_ND subroutine procedure, which replaces the data item in the nth position of a data list. Examples of the procedure appear in Figure 2.14G.

REPLACE\_ND uses three arguments: a data list, the position of the data item to be replaced, and the new data item. The procedure combines a deletion and an insertion involving the nth position in the list.

*REPLACE\_FD Subroutine* Figures 2.14C and 2.14D present the REPLACE\_FD subroutine procedure, which replaces the data item in the first position of a data list. Figure 2.14G contains examples of the procedure.

#### REMOVE\_LD Function

##### Purpose

To get and delete the data item in the last position of a data list

##### Reference

REMOVE\_LD(LIST)

##### Entry-Name Declaration

```

DECLARE REMOVE_LD ENTRY(POINTER)
  RETURNS(CHARACTER(1));

```

##### Meaning of Argument

LIST --the pointer variable that is the head of the list to be processed

##### Remarks

When LIST is null, no data item is deleted from the list, and a blank character is returned.

##### Other Programmer-Defined Procedures Required

REMOVE\_ND and SIZE

##### Method

The following reference is used:

```
REMOVE_ND(LIST, SIZE(LIST))
```

Figure 2.13E. Description of the REMOVE\_LD function for removing the data item in the last position of a data list

```

REMOVE_LD:
  PROCEDURE (LIST)
  RETURNS (CHARACTER(1));
DECLARE
  LIST POINTER;
  RETURN(REMOVE_ND(LIST,SIZE(LIST)));
END
  REMOVE_LD;

```

Figure 2.13F. Removing the data item in the last position of a data list



Data List (before reference)	Function Reference	Function Value	Data List (after reference)
L1:	REMOVE_ND(L1,1)	'*'	L1:
L1:	REMOVE_ND(L1,2)	'\$'	L1:
L1:	REMOVE_FD(L1)	'*'	L1:
L1:	REMOVE_LD(L1)	'2'	L1:
L1:	REMOVE_ND(L1,5)	'b'	L1:
L2:	REMOVE_LD(L2)	'b'	L2:

Figure 2.13G. Examples of references to the REMOVE\_ND, REMOVE\_FD, and REMOVE\_LD functions

## REPLACE\_ND Subroutine

### Purpose

To replace the data item in the nth position of a data list

### Reference

REPLACE\_ND(LIST, N, D)

### Entry-Name Declaration

```
DECLARE REPLACE_ND
  ENTRY(POINTER, FIXED DECIMAL(5),
  CHARACTER(1));
```

### Meaning of Arguments

LIST --the pointer variable that is the head of the list to be processed  
N --the position of the data item to be replaced  
D --the data item that replaces the nth data item

### Remarks

When N is less than one, the value of D becomes the first data item in the list. When N exceeds the size of the list, the value of D becomes the last item in the list. In both of these cases, the size of the list increases by one. In other cases, the size of the list remains unchanged.

### Other Programmer-Defined Procedures Required

DELETE\_ND and INSERT\_ND

### Method

The nth data item in the list is deleted. Then the value of D is inserted into the nth position of the list.

Figure 2.14A. Description of the REPLACE\_ND subroutine for replacing the data item in the nth position of a data list

```
REPLACE_ND:PROCEDURE(LIST, N, D);
  DECLARE
    1 COMPONENT BASED(LIST),
    2 DATA CHARACTER(1),
    2 POINTER POINTER,
    N FIXED DECIMAL(5),
    D CHARACTER(1),
    LIST POINTER;
    CALL DELETE_ND(LIST,N);
    CALL INSERT_ND(LIST,N,D);
  END
  REPLACE_ND;
```

Figure 2.14B. Replacing the data item in the nth position of a data list

## REPLACE\_FD Subroutine

### Purpose

To replace the data item in the first position of a data list

### Reference

REPLACE\_FD(LIST, D)

### Entry-Name Declaration

```
DECLARE REPLACE_FD
  ENTRY(POINTER, CHARACTER(1));
```

### Meaning of Arguments

LIST --the pointer variable that is the head of the list to be processed  
D --the data item that replaces the first data item

### Remarks

When LIST is null, D is inserted into the list.

### Other Programmer-Defined Procedures Required

REPLACE\_ND

### Method

The following reference is used:

REPLACE\_ND(LIST, 1, D)

Figure 2.14C. Description of the REPLACE\_FD subroutine for replacing the data item in the first position of a data list

```
REPLACE_FD:
  PROCEDURE(LIST,D);
  DECLARE
    LIST POINTER,
    D CHARACTER(1);
    CALL REPLACE_ND(LIST,1,D);
  END
  REPLACE_FD;
```

Figure 2.14D. Replacing the data item in the first position of a data list

**REPLACE\_FD** is similar to **REPLACE\_ND** (Figures 2.14A and 2.14B) but does not require an argument for the item position, because the first position is always implied.

*REPLACE\_LD Subroutine* Figures 2.14E and 2.14F present the **REPLACE\_LD** subroutine procedure, which replaces the data item in the last position of a data list. This procedure is similar to **REPLACE\_FD** (Figures 2.14C and 2.14D) but deals with the last rather than the first position of a list.

Examples of the procedure appear in Figure 2.14G.

#### *Searching for Data Items in Data Lists*

Table lookup operations are possible on data lists by performing serial searches through the lists for specified items. The following discussion develops the **FIND\_D** function procedure, which performs such a search.

*FIND\_D Function* Figures 2.15A, 2.15B, and 2.15C present the **FIND\_D** function procedure, which searches a list for the first occurrence of a specified data item. **FIND\_D** uses two arguments: the data list to be searched and the data item to be searched for in the list. The value returned by the function is the position where the item first occurs in the list. A zero value indicates that the list does not contain the data item.

Examples of the function procedure appear in Figure 2.15C.

#### REPLACE\_LD Subroutine

##### Purpose

To replace the data item in the last position of a data list

##### Reference

**REPLACE\_LD**(LIST, D)

##### Entry-Name Declaration

```
DECLARE REPLACE_LD
  ENTRY(POINTER, CHARACTER(1));
```

##### Meaning of Arguments

**LIST** --the pointer variable that is the head of the list to be processed  
**D** --the data item that replaces the last data item

##### Remarks

When **LIST** is null, **D** is inserted into the list.

##### Other Programmer-Defined Procedures Required

**REPLACE\_ND** and **SIZE**

##### Method

The following reference is used:

```
REPLACE_ND(LIST, SIZE(LIST), D)
```

Figure 2.14E. Description of the **REPLACE\_LD** subroutine for replacing the data item in the last position of a data list

```
REPLACE_LD:
  PROCEDURE (LIST, D);
  DECLARE
    LIST POINTER,
    D CHARACTER(1);
    CALL REPLACE_ND(LIST, SIZE(LIST), D);
  END
  REPLACE_LD;
```

Figure 2.14F. Replacing the data item in the last position of a data list

Data List (before reference)	Subroutine Reference	Data List (after reference)
L1:	REPLACE_ND(L1,1,'\$')	L1:
L1:	REPLACE_ND(L1,3,'b')	L1:
L2:	REPLACE_ND(L2,0,'*')	L2:
L2:	REPLACE_ND(L2,3,'*')	L2:
L3:	REPLACE_FD(L3,'A')	L3:
L3:	REPLACE_LD(L3,'A')	L3:

Figure 2.14G. Examples of references to the REPLACE\_ND, REPLACE\_FD, and REPLACE\_LD subroutines

### FIND\_D Function

#### Purpose

To find the position of the first occurrence of a specified data item in a data list

#### Reference

FIND\_D(LIST, D)

#### Entry-Name Declaration

```
DECLARE FIND_D ENTRY(POINTER,
CHARACTER(1))
RETURNS(FIXED DECIMAL(5));
```

#### Meaning of Arguments

LIST --the pointer variable that is the head of the list to be searched  
D --the data item that is to be searched for in the list

#### Remarks

When the list does not contain D, the function returns a zero value.

#### Other Programmer-Defined Procedures Required

GET\_DATA and ADDRESS\_NEXT

#### Method

The ADDRESS\_NEXT function obtains successive addresses of list components. The GET\_DATA function obtains the value of the data element in successive list components.

```
FIND_D:
PROCEDURE (LIST, D)
RETURNS (FIXED DECIMAL(5));
DECLARE
(LIST, ADDRESS) POINTER,
D CHARACTER(1),
I FIXED DECIMAL(5);
ADDRESS = LIST;
I = 0;
DO
WHILE(ADDRESS~=NULL);
I = I + 1;
IF
D = GET_DATA(ADDRESS)
THEN
RETURN(I);
ADDRESS = ADDRESS_NEXT(ADDRESS);
END;
RETURN(0);
END
FIND_D;
```

Figure 2.15B. Finding the position of a data item in a data list

Figure 2.15A. Description of the FIND\_D function for finding the position of a data item in a data list


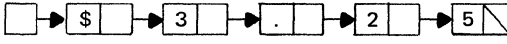


Data List		Function Reference	Function Value
L1:		FIND_D(L1,'\$')	1
L1:		FIND_D(L1, '.')	3
L1:		FIND_D(L1, '**')	0
L2:		FIND_D(L2,'\$')	0

Figure 2.15C. Examples of references to the FIND\_D function

### Interchanging Data Items in Data Lists

Reordering data items in a list is a common list-processing operation. The following discussion develops the SWAP subroutine procedure, which interchanges the data items at two specified positions in a list.

**SWAP Subroutine** Figures 2.16A, 2.16B, and 2.16C present the SWAP subroutine procedure, which interchanges the positions of two data items in a list. The procedure uses three arguments: a data list, the position of one item, and the position of a second item. If an argument specifies a nonexistent list position, no interchange occurs.

Examples of the subroutine procedure appear in Figure 2.16C.

### Manipulating Sublists and Lists

Most of the foregoing procedures apply to single data items and not to collections of items in a list. For example, the DELETE\_ND subroutine (Figures 2.12A and 2.12B) deletes only one data item from a list. If all items are to be deleted, DELETE\_ND must be invoked repeatedly until the list becomes null.

The following discussions develop procedures for manipulating entire lists or portions of lists called sublists. The operations performed by these procedures fall into twelve general categories:

1. Inserting sublists and lists into data lists
2. Deleting sublists and lists from data lists
3. Assigning sublists and lists to data lists
4. Linking data lists
5. Splitting data lists
6. Catenating data lists
7. Searching data lists for sublists
8. Testing data lists for equality
9. Comparing data lists (greater than, equal to, or less than)
10. Reversing data lists
11. Sorting data lists
12. Converting character strings to and from data lists

### SWAP Subroutine

#### Procedure

To interchange the positions of the two data items in a data list

#### Reference

SWAP(LIST, N1, N2)

#### Entry-Name Declaration

```
DECLARE SWAP
ENTRY(POINTER, FIXED DECIMAL(5),
FIXED DECIMAL(5));
```

#### Meaning of Arguments

- LIST --the pointer variable that is the head of the list to be processed
- N1 --the position of a data item in the list
- N2 --the position of another data item in the list

#### Remarks

N1 does not have to be less than N2. The list remains unchanged, however, when N1 or N2 specifies a position outside the list or N1 equals N2.

#### Other Programmer-Defined Procedures Required

ADDRESS\_N, GET\_DATA, and SET\_DATA

#### Method

ADDRESS\_N obtains the addresses of the list components that contain the specified data items. GET\_DATA obtains the values of the data items. SET\_DATA assigns each data item to the list component of the other.

Figure 2.16A. Description of the SWAP subroutine for interchanging data items in a data list

More compact programs can be written with these types of procedures. They eliminate the additional statements needed to control the repeated invocation of procedures that process single data items.

*Inserting Sublists and Lists into Data Lists*

Data lists are frequently created or extended by copying all or part of another list. The following discussions develop two procedures for obtaining data items from one list and inserting them into another list:

1. INSERT\_SUB, which inserts a portion of one list into another list
2. INSERT\_LIST, which inserts a complete list of items into another list

Both procedures obtain storage from the list of available storage components, AVAIL.

```

SWAP:
  PROCEDURE(LIST, N1, N2);
  DECLARE
    D CHARACTER(1),
    (N1, N2) FIXED DECIMAL(5),
    (LIST, ADDRESS1, ADDRESS2) POINTER;
    ADDRESS1 = ADDRESS_N (LIST,N1);
  IF
    ADDRESS1 = NULL
  THEN
    RETURN;
    ADDRESS2 = ADDRESS_N(LIST,N2);
    IF
      ADDRESS2 = NULL
    THEN
      RETURN;
      D = GET_DATA (ADDRESS1);
      CALL SET_DATA (ADDRESS1,
        GET_DATA(ADDRESS2));
      CALL SET_DATA(ADDRESS2, D);
  END
  SWAP;
  
```

Figure 2.16B. Interchanging data items in a data list

Data List (before reference)	Subroutine Reference	Data List (after reference)
L1: [ ] → A → B → C → D	SWAP(L1,1,4)	L1: [ ] → D → B → C → A
L1: [ ] → A → B → C → D	SWAP(L1,4,1)	L1: [ ] → D → B → C → A
L1: [ ] → A → B → C → D	SWAP(L1,2,3)	L1: [ ] → A → C → B → D
L1: [ ] → A → B → C → D	SWAP(L1,1,1)	L1: [ ] → A → B → C → D
L1: [ ] → A → B → C → D	SWAP(L1,0,1)	L1: [ ] → A → B → C → D
L1: [ ] → A → B → C → D	SWAP(L1,1,5)	L1: [ ] → A → B → C → D
L2: [ ]	SWAP(L2,1,1)	L2: [ ]

Figure 2.16C. Examples of references to the SWAP subroutine

*INSERT\_SUB Subroutine* Figures 2.17A and 2.17B present the `INSERT_SUB` subroutine procedure, which inserts a portion of the data items in one list into another list. Examples of the procedure appear in Figure 2.17E.

<b>INSERT_SUB Subroutine</b>	
<b>Purpose</b>	To insert a sublist into a list
<b>Reference</b>	<code>INSERT_SUB(LIST1, N1, LIST2, N2, L)</code>
<b>Entry-Name Declaration</b>	<pre> DEclare INSERT_SUB   ENTRY(Pointer, Fixed Decimal(5),         Pointer, Fixed Decimal(5),         Fixed Decimal(5)); </pre>
<b>Meaning of Arguments</b>	<p><code>LIST1</code> --the list into which the sublist is to be inserted</p> <p><code>N1</code> --the position in <code>LIST1</code> at which the sublist is to be inserted</p> <p><code>LIST2</code> --the list that contains the sublist</p> <p><code>N2</code> --the position in <code>LIST2</code> at which the sublist begins</p> <p><code>L</code> --the number of data items in the sublist</p>
<b>Remarks</b>	<p>If <code>N1</code> is less than one, the sublist is inserted at the front of <code>LIST1</code>. If <code>N1</code> exceeds the size of <code>LIST1</code>, the sublist is inserted at the end of <code>LIST1</code>. If <code>L</code> exceeds the size of <code>LIST2</code>, the sublist is extended with blank characters to give it a size of <code>L</code>; however, <code>LIST2</code> is not changed. When <code>N2</code> is less than one, the sublist contains <code>1-N2</code> leading blanks followed by the first <code>(N2+L-1)</code> characters of <code>LIST2</code>. When <code>(N2+L-1)</code> is greater than the size of <code>LIST2</code>, the sublist contains the last <code>(SIZE(LIST2)-N2+1)</code> characters of <code>LIST2</code> followed by <code>(L-(SIZE(LIST2)-N2+1))</code> blank characters. In all cases, the sublist contains <code>L</code> data items, and <code>LIST2</code> is not changed.</p>
<b>Other Programmer-Defined Procedures Required</b>	<code>INSERT_ND</code> and <code>GET_ND</code>
<b>Method</b>	<p><code>GET_ND</code> obtains each sublist data item from <code>LIST2</code>, and <code>INSERT_ND</code> inserts each sublist data item into <code>LIST1</code>.</p>

Figure 2.17A. Description of the `INSERT_SUB` subroutine for inserting a sublist into a list

```

INSERT_SUB:
  PROCEDURE(LIST1,N1,LIST2,N2,L);
  DECLARE
    (LIST1,LIST2) POINTER,
    (N,N1,N2,L,I) FIXED DECIMAL(5);
  IF
    N1<1
  THEN
    N = 1;
  ELSE
    N = N1;
  DO
    I = 0 TO L-1;
    CALL INSERT_ND(LIST1, N + I,
                  GET_ND(LIST2, N2 + I));
  END;
  END
  INSERT_SUB;

```

Figure 2.17B. Inserting a sublist into a list

<b>INSERT_LIST Subroutine</b>	
<b>Purpose</b>	To insert a list into another list
<b>Reference</b>	<code>INSERT_LIST(LIST1, N, LIST2)</code>
<b>Entry-Name Declaration</b>	<pre> DEclare INSERT_LIST   ENTRY(Pointer, Fixed Decimal(5),         Pointer); </pre>
<b>Meaning of Arguments</b>	<p><code>LIST1</code> --the list into which the second list is to be inserted</p> <p><code>N</code> --the position in <code>LIST1</code> at which the list is to be inserted</p> <p><code>LIST2</code> --the list to be inserted into <code>LIST1</code></p>
<b>Remarks</b>	<p>If <code>N</code> is less than one, <code>LIST2</code> is inserted at the front of <code>LIST1</code>. If <code>N</code> is greater than the size of <code>LIST1</code>, <code>LIST2</code> is inserted at the end of <code>LIST1</code>.</p>
<b>Other Programmer-Defined Procedures Required</b>	<code>INSERT_SUB</code> and <code>SIZE</code>
<b>Method</b>	<p>The following reference is used:</p> <p><code>INSERT_SUB(LIST1, N, LIST2, 1, SIZE(LIST2))</code></p>

Figure 2.17C. Description of the `INSERT_LIST` subroutine for inserting a list into another list

INSERT\_SUB uses five arguments: two data lists, the insertion position in the first list, the retrieval position in the second list, and the number of items to be inserted.

The size of the first list increases by the number of items inserted. The size of the second list remains unchanged.

*INSERT\_LIST Subroutine* Figures 2.17C and 2.17D present the INSERT\_LIST subroutine procedure, which inserts all the data items in one list into another list. Figure 2.17E contains an illustration of the procedure.

```

INSERT_LIST:
  PROCEDURE(LIST1, N, LIST2);
  DECLARE
    (LIST1, LIST2) POINTER,
    N FIXED DECIMAL(5);
  CALL INSERT_SUB(LIST1, N, LIST2, 1,
    SIZE(LIST2));
  END
  INSERT_LIST;

```

Figure 2.17D. Inserting a list into another list

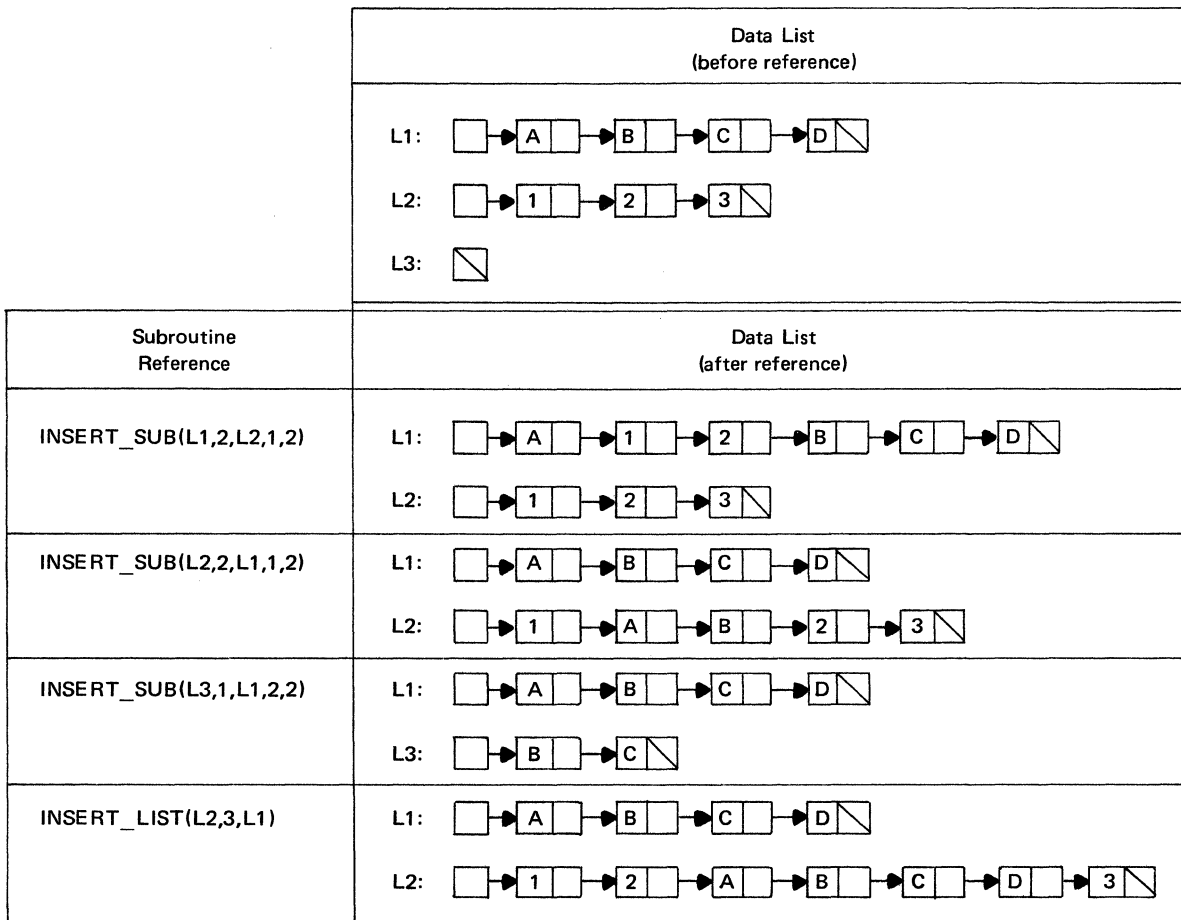


Figure 2.17E. Examples of references to the INSERT\_SUB and INSERT\_LIST subroutines



INSERT\_LIST is similar to INSERT\_SUB (Figures 2.17A and 2.17B) except that an entire list is inserted. As a result of this difference, INSERT\_LIST requires only three arguments: two lists and the insertion position in the first list.

*Deleting Sublists and Lists from Data Lists*

The following discussions develop two subroutine procedures for deleting all or some of the items in a data list:

1. DELETE\_SUB, which deletes a portion of a list
2. DELETE\_LIST, which deletes a complete list

The storage used by the deleted items is returned to the list of available storage components, AVAIL.

*DELETE\_SUB Subroutine* Figures 2.18A and 2.18B present the DELETE\_SUB subroutine procedure, which deletes a portion of a list. Examples of the procedure appear in Figure 2.18E.

DELETE\_SUB uses three arguments: a data list, the list position at which deletion starts, and the number of data items being deleted.

*DELETE\_LIST Subroutine* Figures 2.18C and 2.18D present the DELETE\_LIST subroutine procedure, which deletes all items from a list. Figure 2.18E contains an illustration of the procedure.

DELETE\_LIST is similar to DELETE\_SUB (Figures 2.18A and 2.18B) but uses only one argument: the list being deleted.

*Assigning Sublists and Lists to Data Lists*

The following discussions develop two subroutine procedures for setting a data list equal to all or part of another list:

1. ASSIGN\_SUB, which assigns a sublist to a list
2. ASSIGN\_LIST, which assigns an entire list to another list

The effect of these procedures is equivalent to deletion of the receiving list followed by insertion of a list.

<b>DELETE_SUB Subroutine</b>	
<b>Purpose</b>	To delete a portion of a list
<b>Reference</b>	DELETE_SUB(LIST, N, L)
<b>Entry-Name Declaration</b>	DECLARE DELETE_SUB ENTRY(POINTER, FIXED DECIMAL(5), FIXED DECIMAL(5));
<b>Meaning of Arguments</b>	LIST --the list in which deletion is to occur N --the position in LIST at which deletion is to start L --the number of data items to be deleted
<b>Remarks</b>	Data items are deleted from position N to position (N+L-1) in LIST. When (N+L-1) exceeds the size of LIST, all data items from position N to the end of LIST are deleted. If N exceeds the size of LIST, no data items are deleted. If N is less than one, the first (N+L-1) data items are deleted provided (N+L-1) is not negative; if it is negative, no data items are deleted.
<b>Other Programmer-Defined Procedures Required</b>	DELETE_ND
<b>Method</b>	The following reference is used:  DELETE_ND(LIST, N)

Figure 2.18A. Description of the DELETE\_SUB subroutine for deleting a sublist from a data list

```

DELETE_SUB:
    PROCEDURE(LIST, N, L);
DECLARE
    (N, L, I) FIXED DECIMAL(5),
    LIST POINTER;
DO
    I = N TO (N + L - 1);
    CALL DELETE_ND(LIST, N);
END;
END
    DELETE_SUB;

```

Figure 2.18B. Deleting a sublist

**DELETE\_LIST Subroutine**

**Purpose**

To delete an entire list

**Reference**

DELETE\_LIST(LIST)

**Entry-Name Declaration**

DECLARE DELETE\_LIST ENTRY(POINTER);

**Meaning of Argument**

LIST --the list to be deleted

**Remarks**

LIST is null after deletion.

**Other Programmer-Defined Procedures Required**

DELETE\_SUB and SIZE

**Method**

The following reference is used:

DELETE\_SUB(LIST, 1, SIZE(LIST))

```
DELETE_LIST:
    PROCEDURE(LIST);
    DECLARE
        LIST POINTER;
        CALL DELETE_SUB(LIST, 1,
            SIZE(LIST));
    END
        DELETE_LIST;
```

Figure 2.18D. Deleting a list

Figure 2.18C. Description of the DELETE\_LIST subroutine for deleting a list

Data List (before reference)	Subroutine Reference	Data List (after reference)
L1: □ → X → Y → Z ↘	DELETE_SUB(L1,1,2)	L1: □ → Z ↘
L1: □ → X → Y → Z ↘	DELETE_SUB(L1,3,1)	L1: □ → X → Y ↘
L1: □ → X → Y → Z ↘	DELETE_SUB(L1,0,3)	L1: □ → Z ↘
L1: □ → X → Y → Z ↘	DELETE_SUB(L1,2,3)	L1: □ → X ↘
L1: □ → X → Y → Z ↘	DELETE_SUB(L1,1,3)	L1: □ ↘
L1: □ → X → Y → Z ↘	DELETE_LIST(L1)	L1: □ ↘

Figure 2.18E. Examples of references to the DELETE\_SUB and DELETE\_LIST subroutines

*ASSIGN\_SUB Subroutine* Figures 2.19A and 2.19B present the ASSIGN\_SUB subroutine procedure, which assigns a sublist to a list. Examples of the procedure appear in Figure 2.19E.

<p><b>ASSIGN_SUB Subroutine</b></p> <p><b>Purpose</b> To assign a sublist to a list</p> <p><b>Reference</b> ASSIGN_SUB(LIST1, LIST2, N, L)</p> <p><b>Entry-Name Declaration</b> DECLARE ASSIGN_SUB ENTRY(POINTER, POINTER, FIXED DECIMAL (5), FIXED DECIMAL(5));</p> <p><b>Meaning of Arguments</b> LIST1 --the list to which the sublist is to be assigned LIST2 --the list that contains the sublist N --the position in LIST2 at which the sublist begins L --the number of data items in the sublist</p> <p><b>Remarks</b> LIST1 is deleted before it receives the sublist. The sublist is then inserted into LIST1 according to the conventions of the INSERT_SUB subroutine.</p> <p><b>Other Programmer-Defined Procedures Required</b> DELETE_LIST and INSERT_SUB</p> <p><b>Method</b> The following references are used:  DELETE_LIST(LIST1) and INSERT_SUB(LIST1, 1, LIST2, N, L)</p>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 2.19A. Description of the ASSIGN\_SUB subroutine for assigning a sublist to a list

```

ASSIGN_SUB:
  PROCEDURE(LIST1, LIST2, N, L);
  DECLARE
    (LIST1,LIST2) POINTER,
    (N,L) FIXED DECIMAL(5);
    CALL DELETE_LIST(LIST1);
    CALL INSERT_SUB (LIST1, 1, LIST2,
    N, L);
  END
  ASSIGN_SUB;

```

Figure 2.19B. Assigning a sublist to a list

ASSIGN\_SUB uses four arguments: a receiving list, a source list, the position in the source list at which the sublist begins, and the number of items in the sublist. The source list is not changed by the procedure.

*ASSIGN\_LIST Subroutine* Figures 2.19C and 2.19D present the ASSIGN\_LIST subroutine procedure, which sets one list equal to another. An illustration of the procedure appears in Figure 2.19E.

ASSIGN\_LIST is similar to ASSIGN\_SUB (Figures 2.19A and 2.19B) but requires only two arguments: the receiving and source lists.

<p><b>ASSIGN_LIST Subroutine</b></p> <p><b>Purpose</b> To assign one list to another list</p> <p><b>Reference</b> ASSIGN_LIST(LIST1, LIST2)</p> <p><b>Entry-Name Declaration</b> DECLARE ASSIGN_LIST ENTRY(POINTER, POINTER);</p> <p><b>Meaning of Arguments</b> LIST1 --the list to which the second list is to be assigned LIST2 --the list to be assigned to LIST1</p> <p><b>Remarks</b> LIST1 is deleted before it is assigned the data items of LIST2.</p> <p><b>Other Programmer-Defined Procedures Required</b> ASSIGN_SUB and SIZE</p> <p><b>Method</b> The following reference is used:  ASSIGN_SUB(LIST1, LIST2, 1, SIZE(LIST2))</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 2.19C. Description of the ASSIGN\_LIST subroutine for assigning a list to another list

```

ASSIGN_LIST:
  PROCEDURE(LIST1, LIST2);
  DECLARE
    (LIST1, LIST2) POINTER;
    CALL ASSIGN_SUB(LIST1, LIST2, 1,
    SIZE(LIST2));
  END
  ASSIGN_LIST;

```

Figure 2.19D. Assigning a list to another list

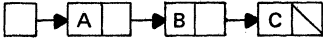
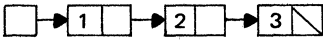
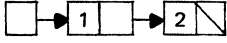
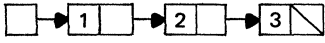

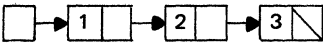

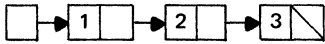
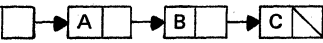
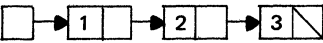

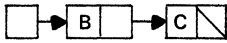
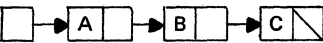
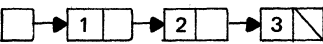
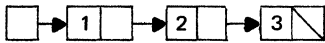
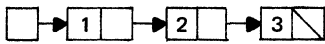
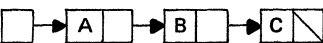
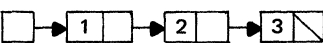
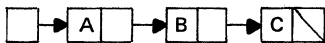
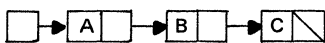
Data List (before reference)	Subroutine Reference	Data List (after reference)
L1:  L2: 	ASSIGN_SUB(L1,L2,1,2)	L1:  L2: 
L3:  L2: 	ASSIGN_SUB(L3,L2,2,2)	L3:  L2: 
L1:  L2: 	ASSIGN_SUB(L2,L1,2,3)	L1:  L2: 
L1:  L2: 	ASSIGN_SUB(L1,L2,1,3)	L1:  L2: 
L1:  L2: 	ASSIGN_LIST(L2,L1)	L1:  L2: 

Figure 2.19E. Examples of references to the ASSIGN\_SUB and ASSIGN\_LIST subroutines

### *Linking Data Lists*

Two lists may be combined to form a single list. One way of performing this operation is to insert the items of one list at the end of another list and to delete the contributing list. A more efficient method involves direct linkage of one list behind another by address manipulation rather than data movement. This method is used in the following discussion to develop the LINK subroutine procedure.

*LINK Subroutine* Figures 2.20A, 2.20B, and 2.20C present the LINK subroutine procedure, which appends the data items of one list behind the items of another list. The subroutine uses two arguments: the data lists being linked. The appended list becomes null after LINK has been executed.

### *Splitting Data Lists*

The reverse operation of linking two lists is to split a single list into two separate lists. The following discussion develops the SPLIT subroutine procedure for such an operation.

*SPLIT Subroutine* Figures 2.21A, 2.21B, and 2.21C present the SPLIT subroutine procedure, which divides a list into two lists. The subroutine uses three arguments: the list to be split, the position where the split is to occur, and the list that receives the split items.

### *Catenating Data Lists*

When the LINK subroutine (Figures 2.20A through 2.20C) is used to link two data lists, both lists are modified by the subroutine. If the lists are to remain unchanged, LINK cannot be used.

The following discussion develops the CATENATE subroutine procedure for creating a third list from the data items of two other lists.

*CATENATE Subroutine* Figures 2.22A, 2.22B, and 2.22C present the CATENATE subroutine procedure, which links the data items of two lists to form a third list. The subroutine uses three arguments: the two lists to be linked and the list that receives the linked items.

#### LINK Subroutine

##### Purpose

To append the data items of one list at the end of another list

##### Reference

LINK(LIST1, LIST2)

##### Entry-Name Declaration

```
DECLARE LINK ENTRY(POINTER, POINTER);
```

##### Meaning of Arguments

LIST1 --the list to which the data items of the second list are appended  
LIST2 --the list whose data items are appended to LIST1

##### Remarks

LIST2 is null after its data items are appended to LIST1.

##### Other Programmer-Defined Procedures Required

ADDRESS\_NEXT and SET\_POINTER

##### Method

The ADDRESS\_NEXT function is used to progress to the last component of LIST1.

The SET\_POINTER subroutine links the last component of LIST1 to the first component of LIST2. LIST2 is then set to null.

Figure 2.20A. Description of the LINK subroutine for linking two lists

```

LINK:
  PROCEDURE(LIST1, LIST2);
  DECLARE
    (LIST1, LIST2, ADDRESS1, ADDRESS2)
    POINTER;
    IF
      LIST1 = NULL
    THEN
      DO;
        LIST1 = LIST2; LIST2 = NULL;
        RETURN;
      END;
      DO
        ADDRESS2 = LIST1;
        WHILE(ADDRESS2≠NULL);
        ADDRESS1 = ADDRESS2;
        ADDRESS2 = ADDRESS_NEXT(ADDRESS2);
      END;
      CALL SET_POINTER(ADDRESS1, LIST2);
      LIST2 = NULL;
    END
  LINK;

```

Figure 2.20B. Linking two lists

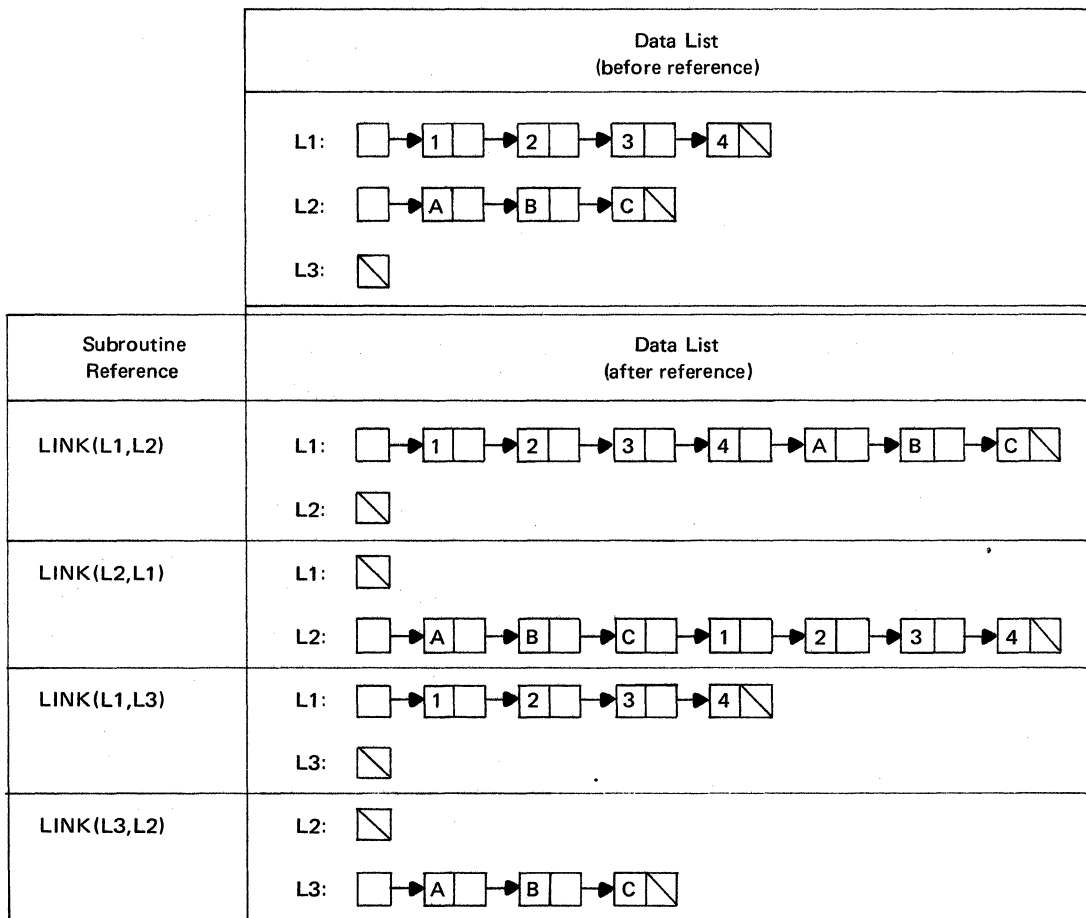


Figure 2.20C. Examples of references to the LINK subroutine

## SPLIT Subroutine

### Purpose

To divide a list into two lists

### Reference

SPLIT(LIST1, N, LIST2)

### Entry-Name Declaration

```
DECLARE SPLIT  
  ENTRY(POINTER, FIXED DECIMAL(5),  
  POINTER);
```

### Meaning of Arguments

LIST1 --the list that is to be split  
N --the position at which LIST1 is to be split  
LIST2 --the list to which the second portion of  
LIST1 is to be assigned

### Remarks

Before LIST1 is split, LIST2 is deleted. Then the data items from position N to the end of LIST1 are removed from LIST1 and linked to LIST2. If N exceeds the size of LIST1, no splitting occurs, but LIST2 is deleted. If N is less than or equal to one, all data items in LIST1 are linked to LIST2, and LIST1 is set to null.

### Other Programmer-Defined Procedures Required

DELETE\_LIST, ADDRESS\_N, GET\_POINTER,  
and SET\_POINTER

### Method

DELETE\_LIST deletes LIST2. ADDRESS\_N obtains the address of the (N-1)th component of LIST1. This pointer element is then assigned to LIST2, causing the second portion of LIST1 to be linked to LIST2. Finally, the pointer element in the (N-1)th component of LIST1 is set to null.

```
SPLIT:  PROCEDURE(LIST1, N, LIST2);  
  DECLARE  
    (LIST1, LIST2, P) POINTER,  
    N FIXED DECIMAL(5);  
  CALL DELETE_LIST(LIST2);  
  IF  
    (N<=1)  
  THEN  
  DO;  
    LIST2 = LIST1; LIST1 = NULL;  
    RETURN;  
  END;  
  P = ADDRESS_N(LIST1, N - 1);  
  IF  
    (P = NULL)  
  THEN  
  RETURN;  
  LIST2 = GET_POINTER(P);  
  CALL SET_POINTER(P, NULL);  
  END  
  SPLIT;
```

Figure 2.21B. Splitting a list

Figure 2.21A. Description of the SPLIT subroutine for splitting a list

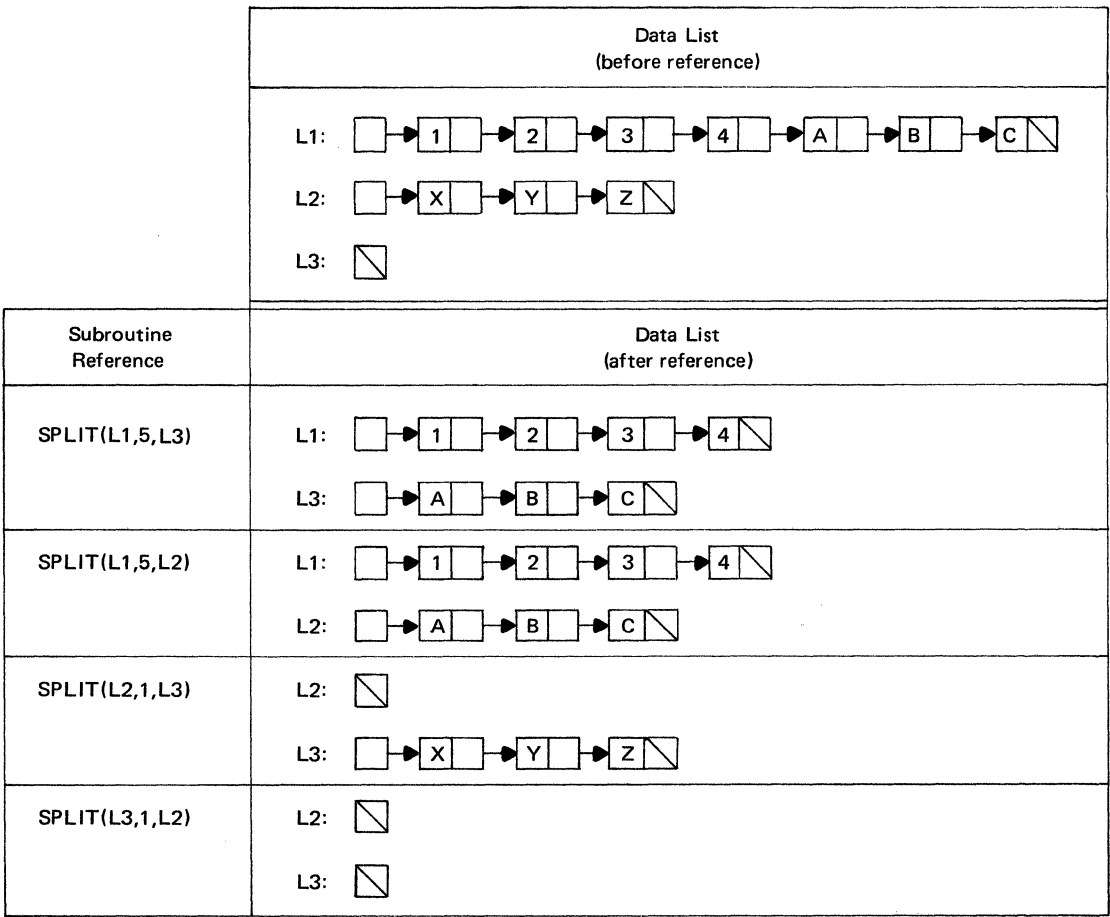


Figure 2.21C. Examples of references to the SPLIT subroutine



## CATENATE Subroutine

### Purpose

To form a third list by linking the data items of two other lists

### Reference

CATENATE(LIST1, LIST2, LIST3)

### Entry-Name Declaration

```
DECLARE CATENATE  
  ENTRY(POINTER, POINTER, POINTER);
```

### Meaning of Arguments

LIST1 --the first list to be linked  
LIST2 --the list to be linked behind LIST1  
LIST3 --the list to which the linked data items of LIST1 and LIST2 are assigned

### Remarks

Any two or all three list arguments may be the same list. LIST1 and LIST2 are not changed when LIST3 is different from LIST1 and LIST2.

### Other Programmer-Defined Procedures Required

INSERT\_LIST, ASSIGN\_LIST, and DELETE\_LIST

### Method

INSERT\_LIST is used to form a temporary list that contains the linked data items of LIST1 and LIST2. ASSIGN\_LIST assigns the temporary list to LIST3. DELETE\_LIST deletes the temporary list.

```
CATENATE:  
  PROCEDURE(LIST1, LIST2, LIST3):  
  DECLARE  
    (LIST1, LIST2, LIST3, T) POINTER;  
    T = NULL;  
    CALL INSERT_LIST(T, 1, LIST2);  
    CALL INSERT_LIST(T, 1, LIST1);  
    CALL ASSIGN_LIST(LIST3, T);  
    CALL DELETE_LIST(T);  
  END  
CATENATE;
```

Figure 2.22B. Catenating two lists and assigning the result to a third list

Figure 2.22A. Description of the CATENATE subroutine for catenating two lists and assigning the result to a third list

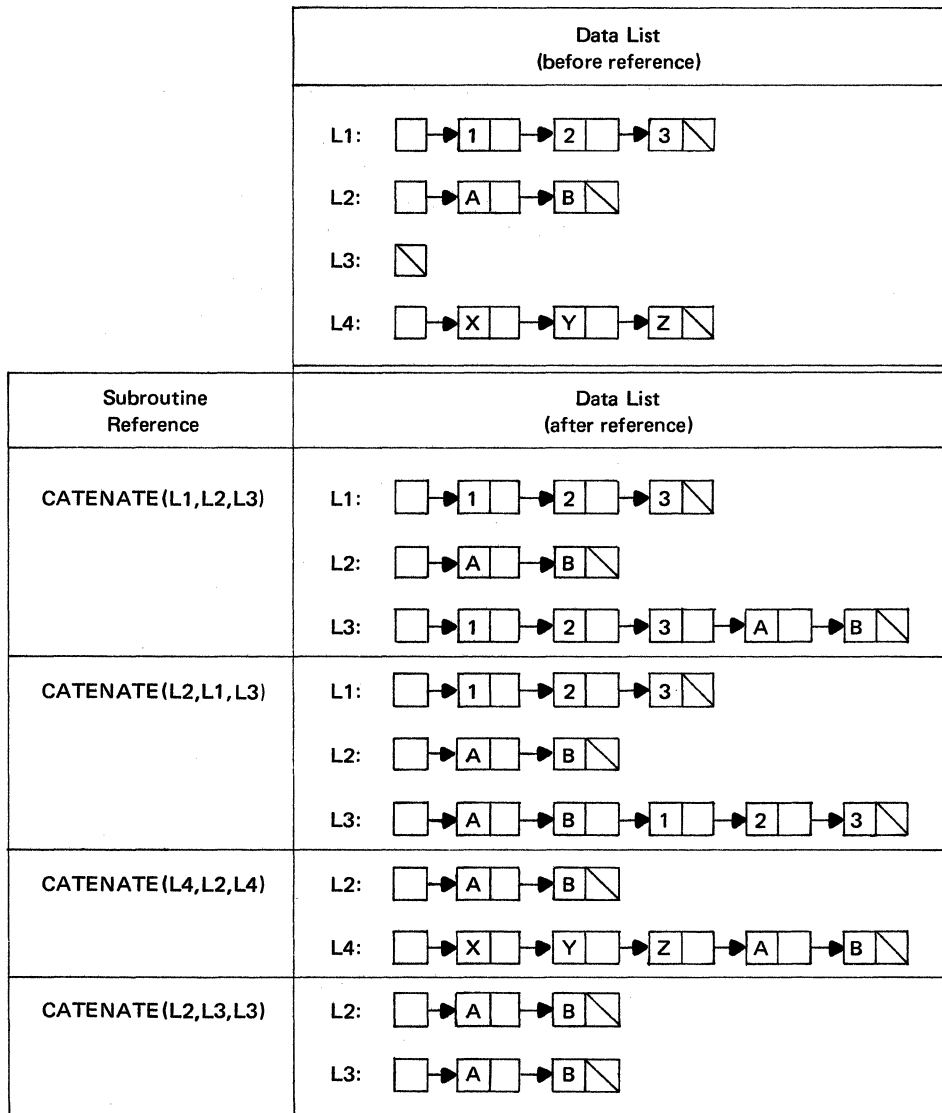


Figure 2.22C. Examples of references to the CATENATE subroutine

### *Searching Data Lists for Sublists*

Retrieval of information from a list may involve a pattern search through the list, such as scanning for a repeating decimal. The following discussion develops the FIND\_LIST function procedure, which searches a list for the appearance of another list. The search can be restricted to a portion of the list and need not involve the entire list.

*FIND\_LIST Function* Figures 2.23A, 2.23B, and 2.23C present the FIND\_LIST function procedure, which searches a list between two positions in the list for the first appearance of another list. The function uses four arguments: the list to be searched, the limiting positions of the search, and the list to be searched for.

When the search is successful, FIND\_LIST returns the first position where the matching list appears. The function returns a zero value when the search is unsuccessful.

### *Testing Data Lists for Equality*

Two data lists may be tested for equality. The following discussion develops the EQUAL function procedure, which determines whether two lists contain the same data items arranged in the same order.

*EQUAL Function* Figures 2.24A, 2.24B, and 2.24C present the EQUAL function procedure, which tests two lists for equality both in number and order of data items. The function returns a one-bit when the lists are equal and a zero-bit when they are not equal.

#### **FIND\_LIST Function**

##### **Purpose**

To search a list between two data positions for the first appearance of a second list

##### **Reference**

FIND\_LIST(LIST1, N1, N2, LIST2)

##### **Entry-Name Declaration**

```
DECLARE FIND_LIST  
ENTRY(POINTER, FIXED DECIMAL(5),  
FIXED DECIMAL(5), POINTER);
```

##### **Meaning of Arguments**

LIST1 --the list to be searched  
N1 --the position in LIST1 where searching begins  
N2 --the position in LIST1 where searching ends  
LIST2 --the list to be searched for

##### **Remarks**

The function returns the first position between N1 and N2 where LIST2 appears in LIST1. If LIST2 is not found, the function returns a zero value. A zero value is also returned when LIST1 or LIST2 is null or N1 is greater than N2.

##### **Other Programmer-Defined Procedures Required**

GET\_ND and SIZE

##### **Method**

GET\_ND obtains data items from each list for comparison.

Figure 2.23A. Description of the FIND\_LIST function for finding a list in another list

```

FIND_LIST:
  PROCEDURE(LIST1, N1, N2, LIST2)
  RETURNS (FIXED DECIMAL(5));
DECLARE
  (LIST1, LIST2) POINTER,
  (N1,N2,LL,UL,S1,S2,I,J)
  FIXED DECIMAL(5);
  /* IF LIST1 OR LIST2 IS NULL,
  RETURN ZERO. */
  IF
  (LIST1 = NULL)|(LIST2 = NULL)
  THEN
  RETURN(0);
  /* IF LIST1 IS SHORTER THAN LIST2,
  RETURN ZERO. */
  S1 = SIZE(LIST1);
  S2 = SIZE(LIST2);
  IF
  S1<S2
  THEN
  RETURN(0);
  /* INITIALIZE LOWER AND UPPER
  SEARCH LIMITS. */
  LL = N1; UL = N2;
  /* IF LOWER SEARCH LIMIT EXCEEDS
  SEARCH LIMIT, RETURN ZERO. */
  IF
  LL>UL
  THEN
  RETURN(0);
  /* IF LOWER SEARCH LIMIT EXCEEDS
  SIZE OF LIST1, RETURN ZERO. */
  IF
  LL>S1
  THEN
  RETURN(0);
  /* IF LOWER SEARCH LIMIT IS LESS
  THAN ONE, ADJUST LIMIT. */
  IF
  LL<1
  THEN
  LL = 1;
  /* IF UPPER SEARCH LIMIT EXCEEDS
  SIZE OF LIST1, ADJUST LIMIT. */
  IF
  UL>S1
  THEN
  UL = S1;
  /* IF SIZE OF LIST2 EXCEEDS EXTENT
  OF SEARCH, RETURN ZERO. */
  IF
  S2>(UL - LL + 1)
  THEN
  RETURN(0);
  /* FIND FIRST POSITION OF LIST2
  BETWEEN LOWER AND UPPER SEARCH
  LIMITS OF LIST1.*/
  DO
  I = LL TO (UL - S2 + 1);
  DO
  J = 1 TO S2;
  IF
  GET_ND(LIST1, I + J - 1)≠GET_ND
  (LIST2,J)
  THEN
  GO TO
  L1;
  /* LIST2 FOUND AT I-TH POSITION OF
  LIST1. */
  RETURN(I);
  L1:
  END;
  /* LIST2 NOT FOUND. */
  RETURN(0);
END
FIND_LIST;

```

Figure 2.23B. Finding a list in another list

Data List	Function Reference	Function Value
L1: [ ] → W → * → X → + → Y → / → Z → [ ] L2: [ ] → X → + → Y → [ ]	FIND_LIST(L1,1,7,L2)	3
L3: [ ] → G → O → T → O → [ ] → L → ; → [ ] L4: [ ] → T → O → [ ]	FIND_LIST(L3,3,6,L4)	3
L5: [ ] → - → 2 → 7 → 3 → . → 6 → [ ] L6: [ ] → . → [ ]	FIND_LIST(L5,2,4,L6)	0
L7: [ ] → E → N → D → [ ] L8: [ ] → E → N → D → ; → [ ]	FIND_LIST(L7,1,3,L8)	0
L7: [ ] → E → N → D → [ ] L8: [ ] → E → N → D → ; → [ ]	FIND_LIST(L8,1,4,L7)	1

Figure 2.23C. Examples of references to the FIND\_LIST function

## EQUAL Function

### Purpose

To test two data lists for equality

### Reference

EQUAL(LIST1, LIST2)

### Entry-Name Declaration

```
DECLARE EQUAL ENTRY(POINTER, POINTER)
  RETURNS (BIT(1));
```

### Meaning of Arguments

LIST1 --the first list  
LIST2 --the second list

### Remarks

The function returns a one-bit when the lists are equal; otherwise, a zero-bit. If both lists are null, they are considered to be equal.

### Other Programmer-Defined Procedures Required

ADDRESS\_NEXT and GET\_DATA

### Method

ADDRESS\_NEXT obtains successive list components. GET\_DATA obtains the data element of each list component. Testing stops when corresponding list positions do not contain equal data items. Lists with different sizes are always unequal.

```
EQUAL:
  PROCEDURE (LIST1, LIST2)
  RETURNS (BIT(1));
  DECLARE
    (LIST1, LIST2, ADDRESS1, ADDRESS2)
    POINTER;
    ADDRESS1 = LIST1;
    ADDRESS2 = LIST2;
  L:
    IF
      ADDRESS1 = ADDRESS2
    THEN
      RETURN('1'B);
    IF
      (ADDRESS1 = NULL) || (ADDRESS2 = NULL)
    THEN
      RETURN('0'B);
    IF
      GET_DATA(ADDRESS1) ≠ GET_DATA
      (ADDRESS2)
    THEN
      RETURN('0'B);
      ADDRESS1 = ADDRESS_NEXT(ADDRESS1);
      ADDRESS2 = ADDRESS_NEXT(ADDRESS2);
    GO TO
      L;
  END
  EQUAL;
```

Figure 2.24B. Testing two data lists for equality

Figure 2.24A. Description of the EQUAL function for testing the equality of two data lists

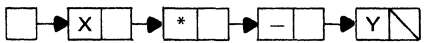
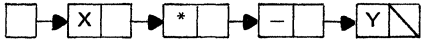
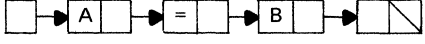
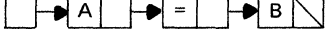




Data List	Function Reference	Function Value
L1:  L2: 	EQUAL(L1,L2)	'1'B
L3:  L4: 	EQUAL(L3,L4)	'0'B
L5:  L6: 	EQUAL(L5,L6)	'1'B
L7:  L8: 	EQUAL(L7,L8)	'0'B

Figure 2.24C. Examples of references to the EQUAL function

## Comparing Data Lists

The conventional string comparisons (less than, equal to, and greater than) may also be applied to data lists. The following discussion develops the COMPARE function procedure for comparing two lists.

<p><b>COMPARE Function</b></p> <p><b>Purpose</b> To determine whether a list is less than, equal to, or greater than another list</p> <p><b>Reference</b> COMPARE(LIST1, LIST2)</p> <p><b>Entry-Name Declaration</b> DECLARE COMPARE ENTRY(POINTER, POINTER) RETURNS(BIT(2));</p> <p><b>Meaning of Arguments</b> LIST1 --the first list LIST2 --the second list</p> <p><b>Remarks</b> When LIST1 equals LIST2, the function returns '11'B. When LIST1 is less than LIST2, the function returns '01'B. When LIST1 is greater than LIST2, the function returns '10'B.</p> <p><b>Other Programmer-Defined Procedures Required</b> ADDRESS_NEXT and GET_DATA</p> <p><b>Method</b> ADDRESS_NEXT obtains the address of successive list components. GET_DATA obtains the data element of each component. Comparison begins with the first data item in each list and matches successive items in corresponding positions. Comparison stops when the first unequal match occurs. If the end of one list is reached before inequality is established, the shorter list is considered to be less than the longer. Comparison of two null lists produces an equal match.</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```
COMPARE:
PROCEDURE (LIST1, LIST2)
RETURNS (BIT (2));
DECLARE
(D1,D2) CHARACTER(1),
(LIST1,LIST2,ADDRESS1,ADDRESS2)
POINTER;
ADDRESS1 = LIST1;
ADDRESS2 = LIST2;
L:
    IF
ADDRESS1 = ADDRESS2
THEN
RETURN('11'B);
    IF
ADDRESS1 = NULL
THEN
RETURN('01'B);
    IF
ADDRESS2 = NULL
THEN
RETURN('10'B);
D1 = GET_DATA(ADDRESS1);
D2 = GET_DATA(ADDRESS2);
    IF
D1<D2
THEN
RETURN('01'B);
    IF
D1>D2
THEN
RETURN('10'B);
ADDRESS1 = ADDRESS_NEXT(ADDRESS1);
ADDRESS2 = ADDRESS_NEXT(ADDRESS2);
GO TO
L;
END
COMPARE;
```

Figure 2.25B. Comparing two data lists

*COMPARE Function* Figures 2.25A, 2.25B and 2.25C present the COMPARE function procedure, which determines whether one list is less than, equal to, or greater than another list. The function returns a two-position bit string. Both positions contain one-bits when the lists are equal. A zero-bit in the left position indicates a less-than comparison, and a zero-bit in the right position indicates a greater-than comparison.

Figure 2.25A. Description of the COMPARE function for comparing two data lists

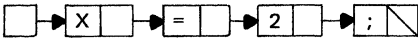
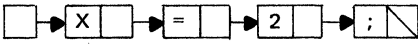
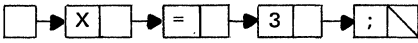
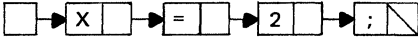
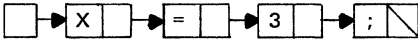
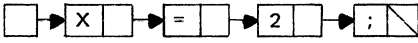
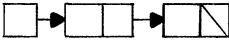
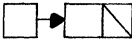


Data List	Function Reference	Function Value
L1: 	COMPARE(L1,L2)	'11'B (=)
L2: 		
L3: 	COMPARE(L4,L3)	'01'B (<)
L4: 		
L3: 	COMPARE(L3,L4)	'10'B (>)
L4: 		
L5: 	COMPARE(L5,L6)	'10'B (>)
L6: 		
L7: 	COMPARE(L7,L8)	'11'B (=)
L8: 		

Figure 2.25C. Examples of references to the COMPARE function

### Reversing Data Lists

Frequent manipulation of data items at the end of a list may become time-consuming, because the addresses of the items must be obtained by proceeding serially through the list. One way of improving the access time for data items at the end of a list is to reverse the order of the items in the list. The desired items will then lie at the front of the list, and fewer items will have to be passed over.

The following discussion develops the REVERSE subroutine procedure for reversing the order of the data items in a list.

**REVERSE Subroutine** Figures 2.26A, 2.26B, and 2.26C present the REVERSE subroutine procedure, which reverses the order of the data items in a list. The procedure

removes successive data items from the front of the list and inserts them successively at the front of a temporary list. When the original list becomes null, it receives the items in the temporary list.

### Sorting Data Lists

The following discussion develops the SORT subroutine for arranging the data items of a list in ascending sequence.

**SORT Subroutine** Figures 2.27A, 2.27B, and 2.27C present the SORT subroutine procedure. The procedure removes all items from the list and inserts them in a temporary list. Each item is then reinserted into the original list in sort sequence.

**REVERSE Subroutine**

**Purpose**  
To reverse the order of the data items in a list

**Reference**  
REVERSE(LIST)

**Entry-Name Declaration**  
DECLARE REVERSE ENTRY(POINTER);

**Meaning of Argument**  
LIST --the list to be reversed

**Remarks**  
LIST can be null.

**Other Programmer-Defined Procedures Required**  
INSERT\_FD and REMOVE\_FD

**Method**  
REMOVE\_FD obtains and removes successive data items from the first position of LIST. INSERT\_FD stores each item (as it is obtained) in the first position of a temporary list, T. The following reference is used:

INSERT\_FD(T, REMOVE\_FD(LIST))

When LIST becomes null, it is assigned the value of pointer T. Then DELETE\_LD(LIST) removes whatever was in the data item in AVAIL.

```

REVERSE: PROCEDURE(LIST);
  DECLARE (LIST, T) POINTER,
  AVAIL EXTERNAL POINTER;
  T = AVAIL;
  AVAIL->POINTER = NULL;
  DO WHILE (LIST /= NULL);
  CALL INSERT_FD(T,REMOVE_FD(LIST));
  END;
  LIST = T;
  CALL DELETE_LD(LIST);
END REVERSE;

```

Figure 2.26B. Reversing a data list

Figure 2.26A. Description of the REVERSE subroutine for reversing a data list

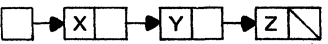
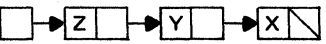


Data List (before reference)	Subroutine Reference	Data List (after reference)
L1: 	REVERSE(L1)	L1: 
L2: 	REVERSE(L2)	L2: 

Figure 2.26C. Examples of references to the REVERSE subroutine



**SORT Subroutine**

**Purpose**  
To arrange the data items in a list into ascending sequence

**Reference**  
SORT(LIST)

**Entry-Name Declaration**  
DECLARE SORT ENTRY(POINTER);

**Meaning of Argument**  
LIST --the list to be sorted

**Remarks**  
LIST can be null.

**Other Programmer-Defined Procedures Required**  
REMOVE\_FD, GET\_DATA, ADDRESS\_NEXT, INSERT\_ND, and INSERT\_LD

**Method**  
All data items are removed from LIST and assigned to a temporary list, T. Each item is then reinserted into LIST in ascending sequence.

```

SORT:
  PROCEDURE(LIST);
  DECLARE
    (LIST, T, ADDRESS) POINTER,
    D CHARACTER(1),
    N FIXED DECIMAL(5);
    IF
      LIST = NULL
    THEN
      RETURN;
      T = LIST;
      LIST = NULL;
    DO
      WHILE(T≠NULL);
      D = REMOVE_FD(T);
      N = 0;
      ADDRESS = LIST;
    DO
      WHILE(ADDRESS≠NULL);
      N = N + 1;
      IF
        GET_DATA(ADDRESS) >= D
      THEN
        DO;
          CALL INSERT_ND(LIST,N,D);
          GO TO
            L;
        END;
        ADDRESS = ADDRESS_NEXT(ADDRESS);
      CALL INSERT_LD(LIST,D);
    L:
  END;
END;
END
  SORT;

```

Figure 2.27B. Sorting a data list

Figure 2.27A. Description of the SORT subroutine for sorting a data list

Data List (before reference)	Subroutine Reference	Data List (after reference)
L1: [ ] → E → N → D → [ ] L2: [ ]	SORT(L1)  SORT(L2)	L1: [ ] → D → E → N → [ ] L2: [ ]

Figure 2.27C. Examples of references to the SORT subroutine

### Converting Character Strings to and from Data Lists

The similarities between character strings and data lists (as developed in this chapter) permit the conversion of character strings to and from data lists. The following discussions develop two subroutine procedures for such conversions:

1. `STRING_TO_LIST`, which converts a character string to a data list
2. `LIST_TO_STRING`, which converts a data list to a character string

*STRING\_TO\_LIST Subroutine* Figures 2.28A, 2.28B, and 2.28C present the `STRING_TO_LIST` subroutine procedure, which obtains successive characters from the character string and inserts them into the data list. The character string is not changed by the procedure.

*LIST\_TO\_STRING Subroutine* Figures 2.29A, 2.29B, and 2.29C present the `LIST_TO_STRING` subroutine procedure, which obtains successive items from the list and inserts them into the string. The data list remains unchanged.

### Manipulating Lists Recursively

The structure of a simple data list always satisfies one of these conditions:

1. The data list is null.
2. The data list contains one data item.
3. Each data item in the list is followed by a data list (which may be null).

These conditions provide an elementary example of recursive organization (see Figure 2.30). Insertion of a data item into a data list does not change the organization of the list; although the size of the list increases, it still retains the structure of a list. Similarly, deletion of a data item from a data list also results in a list organization.

<p><b>STRING_TO_LIST Subroutine</b></p> <p><b>Purpose</b> To convert a character string to a data list</p> <p><b>Reference</b> <code>STRING_TO_LIST(STRING, LIST)</code></p> <p><b>Entry-Name Declaration</b> <code>DECLARE STRING_TO_LIST ENTRY(Character(*), POINTER);</code></p> <p><b>Meaning of Arguments</b> <code>STRING</code> --the character string to be converted to a data list <code>LIST</code> --the list to which the converted character string is to be assigned</p> <p><b>Remarks</b> <code>STRING</code> is a fixed-length character string that can be of any storage class. If <code>STRING</code> has a zero length, <code>LIST</code> becomes null. In all cases, <code>STRING</code> remains unchanged.</p> <p><b>Other Programmer-Defined Procedures Required</b> <code>DELETE_LIST</code> and <code>INSERT_LD</code></p> <p><b>Method</b> <code>DELETE_LIST</code> deletes <code>LIST</code> before conversion begins. The built-in function <code>SUBSTR</code> obtains successive characters from <code>STRING</code>. <code>INSERT_LD</code> inserts successive characters at the end of <code>LIST</code>.</p>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 2.28A. Description of the `STRING_TO_LIST` subroutine for converting a character string to a data list

```

STRING_TO_LIST:
  PROCEDURE (STRING, LIST);
  DECLARE
    LIST POINTER,
    STRING CHARACTER (*),
    I FIXED DECIMAL (5);
  CALL DELETE_LIST (LIST);
  DO
    I = 1 TO LENGTH (STRING);
    CALL INSERT_LD (LIST, SUBSTR
      (STRING, I, 1));
  END;
END
  STRING_TO_LIST;

```

Figure 2.28B. Converting a character string to a data list

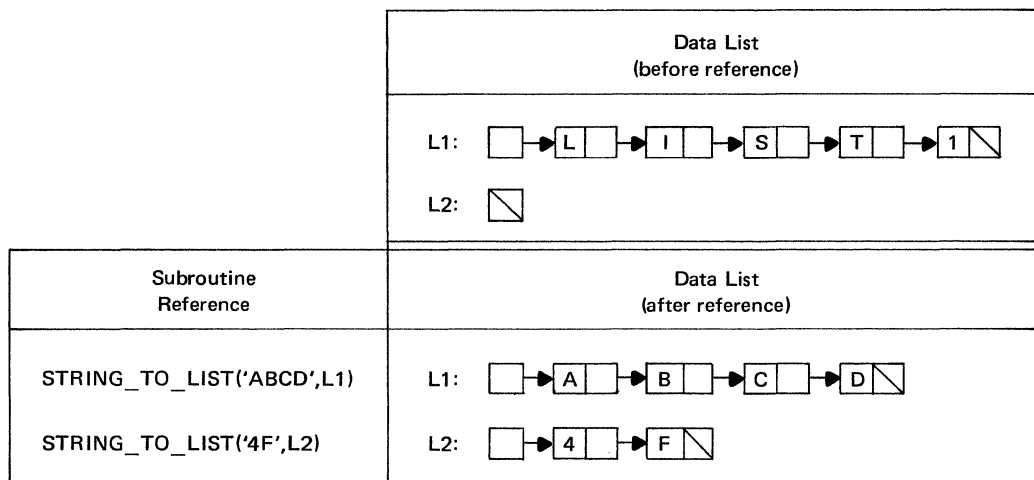


Figure 2.28C. Examples of references to the STRING\_TO\_LIST subroutine

## LIST\_TO\_STRING Subroutine

### Purpose

To convert a data list to a character string

### Reference

LIST\_TO\_STRING(LIST, STRING)

### Entry-Name Declaration

```
DECLARE LIST_TO_STRING  
ENTRY(POINTER, CHARACTER(*));
```

### Meaning of Arguments

LIST --the list to be converted to a character string  
STRING --the character-string variable to which the converted list is to be assigned

### Remarks

STRING is a fixed-length character string that can be of any storage class, but storage must have been allocated for it before LIST\_TO\_STRING is invoked. When the length of STRING is less than the size of LIST, excess data items in LIST are not inserted into STRING. When the length of STRING is greater than the size of LIST, excess positions in STRING become blank. When LIST is null, all positions in STRING become blank. In all cases, LIST remains unchanged.

### Other Programmer-Defined Procedures Required

GET\_ND

### Method

GET\_ND obtains successive data items from LIST. The first data item is assigned to STRING. Remaining data items are inserted into successive positions of STRING through the pseudo variable SUBSTR.

```
LIST_TO_STRING:  
  PROCEDURE(LIST, STRING);  
  DECLARE  
    LIST POINTER,  
    STRING CHARACTER(*),  
    I FIXED DECIMAL(5);  
    STRING = GET_ND(LIST,1);  
  DO  
    I = 2 TO LENGTH(STRING);  
    SUBSTR(STRING,I,1) = GET_ND(LIST,I);  
  END;  
END  
LIST_TO_STRING;
```

Figure 2.29B. Converting a data list to a character string

Figure 2.29A. Description of the LIST\_TO\_STRING subroutine


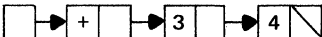

Data List and Character String (before reference)	Subroutine Reference	String (after reference)
L1:  S1: 'FOUR'	LIST_TO_STRING(L1,S1)	S1: 'A1B2'
L2:  S2: '4F'	LIST_TO_STRING(L2,S2)	S2: '+3'
L3:  S3: 'XYZ'	LIST_TO_STRING(L3,S3)	S3: 'bbb'

Figure 2.29C. Examples of references to the LIST\_TO\_STRING subroutine

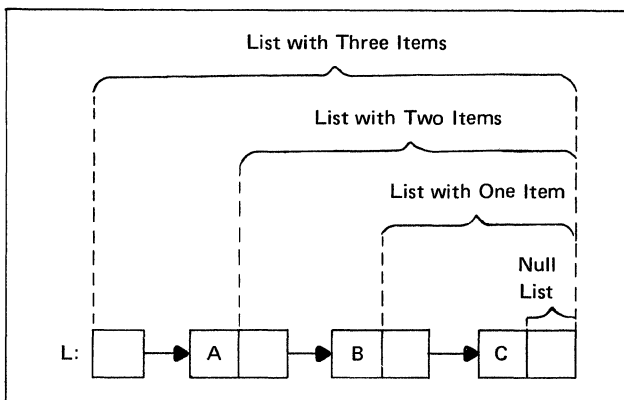


Figure 2.30. Recursive structure of a data list

A PL/I procedure may be declared to have the RECURSIVE option. An active recursive procedure can be activated from within itself. This successive invocation of itself continues until terminated by itself.

The recursive facilities of PL/I procedures allow list-processing operations to be performed recursively. This section develops six recursive procedures for the following operations:

1. Recursive deletion of data lists
2. Recursive computation of data list sizes
3. Recursive searching for data items in data lists
4. Recursive linking of data lists
5. Recursive testing for equality of data lists
6. Recursive comparison of data lists

Note that earlier discussions developed nonrecursive (that is, iterative) procedures for each of these operations. The relative merits of recursive methods compared to iterative techniques is a controversial aspect of computer programming. Generally, recursive methods produce more compact program statements at the expense of increased execution time and storage space. This expense may be justified, however, when iterative techniques distort the natural organization of a recursive application and produce programming complexities that can be avoided with recursive methods.

Appendix 1 contains a summary of the recursive facilities for PL/I procedures.

#### *Recursive Deletion of Data Lists*

Figures 2.30A through 2.30D present the DELETER subroutine procedure, which deletes a data list recursively.

The procedure performs three basic operation:

1. Tests whether the list is null
2. Deletes the first data item in the list
3. Invokes itself recursively

Each recursive invocation of DELETER deletes a data item from the list. When the list becomes null, control returns to each higher level invocation until the original point of invocation is reached.

Figure 2.30C illustrates successive stages in the recursive deletion of a data list.

**DELETER Subroutine**

**Purpose**  
To delete a data list

**Reference**  
DELETER(LIST)

**Entry-Name Declaration**  
DECLARE DELETER ENTRY(POINTER);

**Meaning of Argument**  
LIST --the list to be deleted

**Remarks**  
LIST is null after deletion.

**Other Programmer-Defined Procedures Required**  
DELETE\_FD

**Method**  
DELETER is a recursive subroutine. Each invocation (initial and recursive) causes DELETE\_FD to delete the first data item from LIST. When LIST becomes null, recursion stops, and control returns to the initial point of invocation.

Figure 2.30A. Description of the DELETER subroutine for recursive deletion of a data list

```

DELETER:
  PROCEDURE(LIST) RECURSIVE;
  DECLARE
    LIST POINTER;
    IF
      LIST = NULL
    THEN
      RETURN;
      CALL DELETE_FD(LIST);
      CALL DELETER(LIST);
  END
  DELETER;

```

Figure 2.30B. A recursive subroutine for deleting a data list

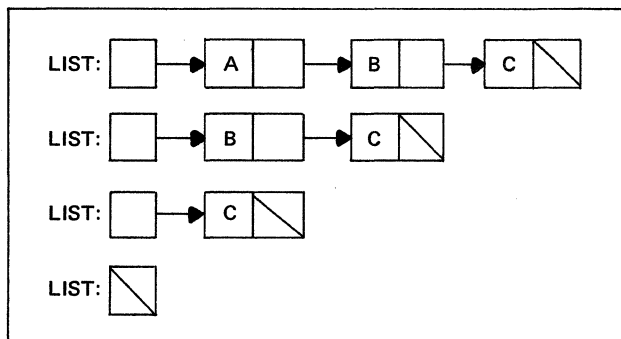


Figure 2.30C. Successive stages in the recursive deletion of a data list

Figure 2.30D shows the flow of control through recursive invocations of the DELETER subroutine. To simplify the presentation, the diagram duplicates the subroutine at each stage of recursion. It also shows the state of the list each time control enters the subroutine.

The diagram begins at the top of Figure 2.30D with execution of the statement:

```
CALL DELETER (LIST);
```

LIST contains two data items, A and B, whose list components are assumed to be at locations 10 and 75. Since LIST is not null the first time control enters DELETER, item A is deleted by the subroutine DELETE\_FD, and DELETER is invoked a second time with LIST as the argument. Again, LIST is not null, and item B is deleted by DELETE\_FD. When DELETER is invoked for the third time, LIST is null, and no further invocations are required.

The RETURN statement in the third copy of DELETER returns control to the END statement in the second copy of DELETER. This END statement then returns control to the END statement in the first copy of DELETER. Finally, control returns to the statement that follows the original invocation of DELETER.

In the diagram, solid lines denote flow of control from an invoking reference to the invoked procedure, and beaded lines represent flow of control from the invoked procedure back to the statement that follows the invoking reference.

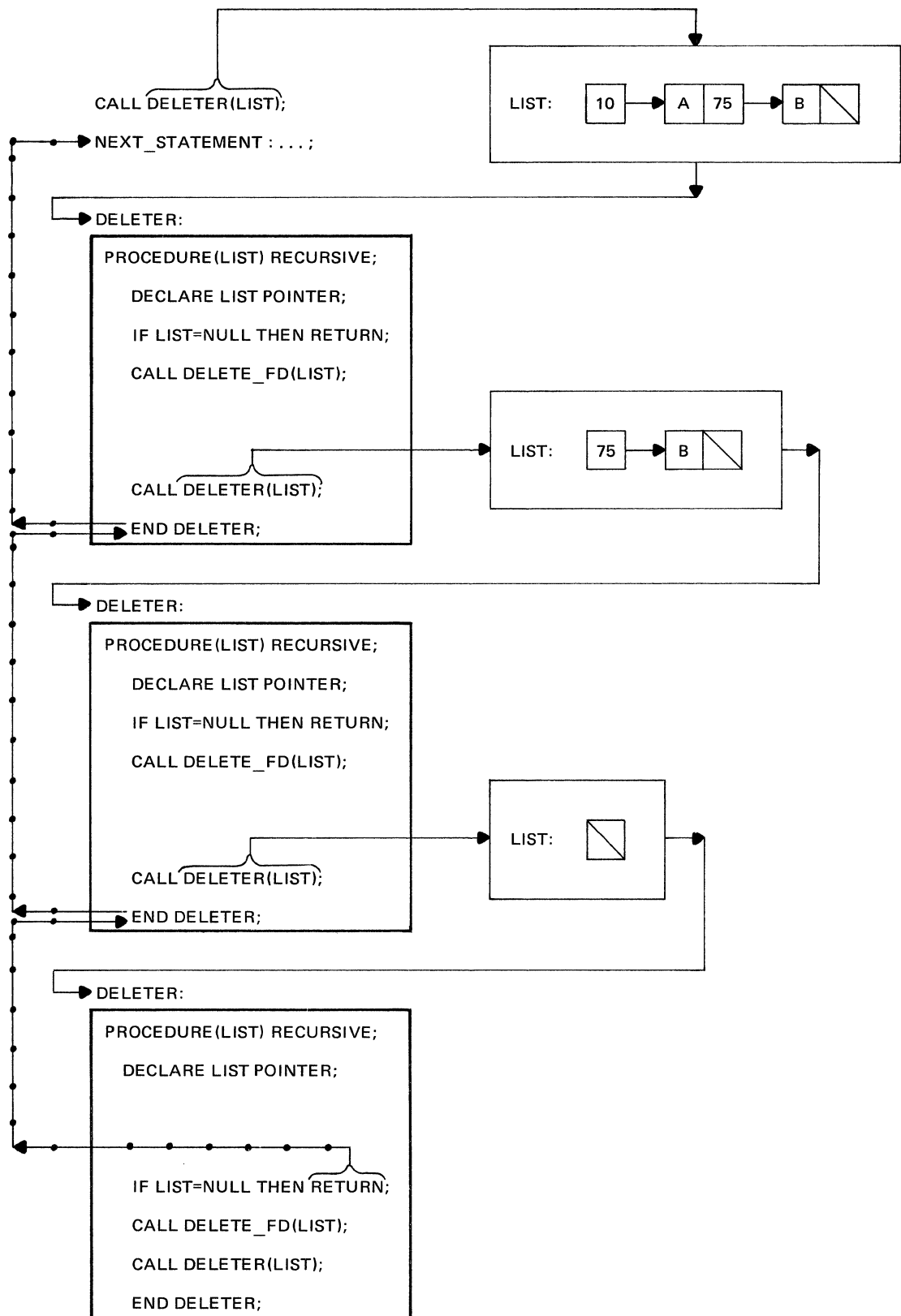


Figure 2.30D. Deleting a list recursively

### Recursive Computation of Data List Sizes

Figures 2.31A through 2.31D present the SIZER function, which computes the size of a data list recursively. When the list is null, SIZER returns a zero value. If the list is not null, the procedure returns the value of the expression:

$$1 + \text{SIZER}(\text{ADDRESS\_NEXT}(\text{LIST}))$$

Evaluation of this expression requires recursive invocation of SIZER. The addresses of successive data items in the list serve as the arguments of successive invocations of SIZER.

<b>SIZER Function</b>
<b>Purpose</b> To obtain the size of a data list
<b>Reference</b> SIZER(LIST)
<b>Entry-Name Declaration</b> DECLARE SIZER ENTRY(POINTER) RETURNS(FIXED DECIMAL(5));
<b>Meaning of Argument</b> LIST --the list whose size is to be computed
<b>Remarks</b> The maximum possible size is 99999. If LIST is null, the function returns a zero size.
<b>Other Programmer-Defined Procedures Required</b> ADDRESS_NEXT
<b>Method</b> SIZER is a recursive function. Each invocation (initial or recursive) causes the value of the following expression to be returned:  $1 + \text{SIZER}(\text{ADDRESS\_NEXT}(\text{LIST}))$  When the argument becomes null, recursion stops, and control returns to the initial point of invocation.

Figure 2.31A. Description of the SIZER function for recursive computation of the size of a data list

```
SIZER:PROCEDURE (LIST) RETURNS
(FIXED DECIMAL(5)) RECURSIVE;
DECLARE
LIST POINTER;
IF LIST = NULL THEN RETURN (0);
RETURN
(1 + SIZER (ADDRESS_NEXT(LIST)));
END SIZER;
```

Figure 2.31B. A recursive function for obtaining the size of a data list

The above expression becomes equivalent to an arithmetic series of ones, which is terminated by the zero value that is returned when the end of the list is reached. The number of ones in the equivalent series equals the number of data items in the list. Figure 2.31C illustrates the successive stages performed by SIZER for a data list of three items.

Figure 2.31D shows how program control flows through recursive invocations of SIZER. The diagram duplicates SIZER at each stage of recursion. It also shows the argument value passed by each invocation, the flow of control into and out of the function, and the value returned to each invoking reference. Since pointer parameter LIST has the automatic storage class, storage is automatically allocated for parameter LIST at each level of recursion, and the address of the next component in the list is assigned as the value of the new generation of parameter LIST. Solid lines in the diagram denote flow of control from an invoking reference to the invoked procedure.

The diagram begins with the execution of the assignment statement:

$$\text{LENGTH} = \text{SIZER}(\text{LIST});$$

As illustrated, LIST is the head pointer of a two-component data list whose components have been given the arbitrary addresses 100 and 250. For a value to be assigned to variable LENGTH on the left of the above assignment statement, the expression on the right must be evaluated first. This expression consists of a reference to the SIZER function.

The head pointer LIST serves as the assignment of the reference and has an address value of 100, which is passed to SIZER. When control enters the function, storage is allocated for parameter LIST, which is internal to the procedure, and the value 100 is assigned to this storage. Because parameter LIST is not null, the following statement is executed:

$$\text{RETURN} (1 + \text{SIZER} (\text{ADDRESS\_NEXT} (\text{LIST})));$$



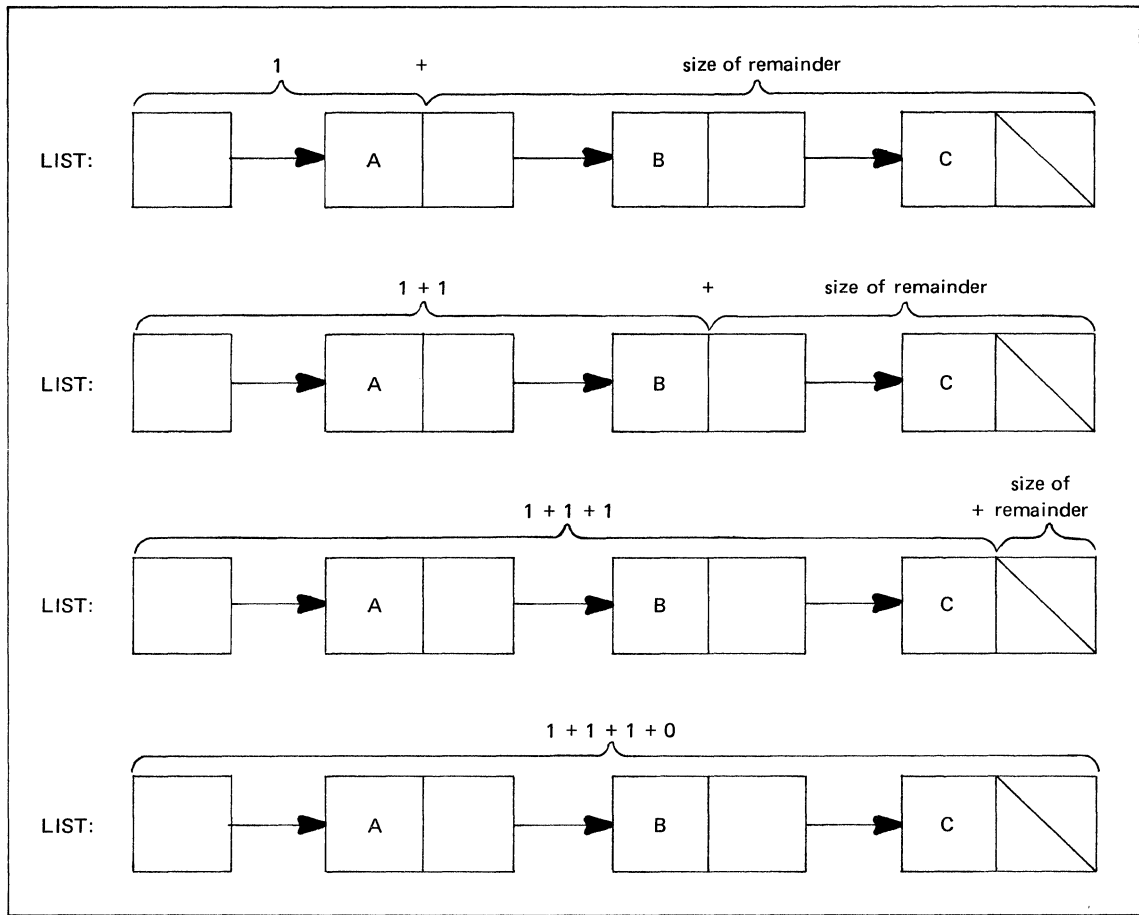


Figure 2.31C. Successive stages in the recursive computation of the size of a data list

This statement cannot return control until its expression is evaluated. However, the expression itself contains reference to `SIZER` that causes recursive invocation of the function with an argument value of 250. The value 250 is obtained from the reference `ADDRESS_NEXT (LIST)`.

This second invocation of `SIZER`, which is represented by the second copy of the function in Figure 2.31D, causes new storage to be allocated for parameter `LIST` with the address value of 250. Again, parameter `LIST` does not have a null value; therefore, the `RETURN` statement mentioned above is reexecuted in the second copy of the function. Once more, return of control is suspended until the expression within the `RETURN` statement is evaluated. The evaluation causes a third invocation of `SIZER` with a null argument value.

When control enters the function for the third time (represented by the third copy of `SIZER` in Figure 2.31D), new storage is allocated for parameter `LIST`, to which a null value is assigned. At this stage of recursion, the null value of parameter `LIST` causes the statement:

```
RETURN (1 + SIZER (ADDRESS_NEXT));
```

In this statement, `SIZER` has a value of zero, and the value returned by the statement, therefore, is one ( $1 = 1 + 0$ ). This value is returned to the still previous point of invocation, which occurred within the first copy of the function and is also associated with the statement:

```
RETURN (1 + SIZER (ADDRESS_NEXT (LIST)));
```

This time `SIZER` has a value of one, and the statement returns a value of two ( $2 = 1 + 1$ ) to the previous point of invocation, which occurred on the right side of the assignment statement:

```
LENGTH = SIZER (LIST);
```

At this point, the value of two for the function reference is assigned to variable `LENGTH`.

Throughout the entire evaluation, the data list remains unchanged. Each recursive allocation of parameter `LIST` serves as a temporary head pointer for the list and leaves the original head pointer undisturbed.

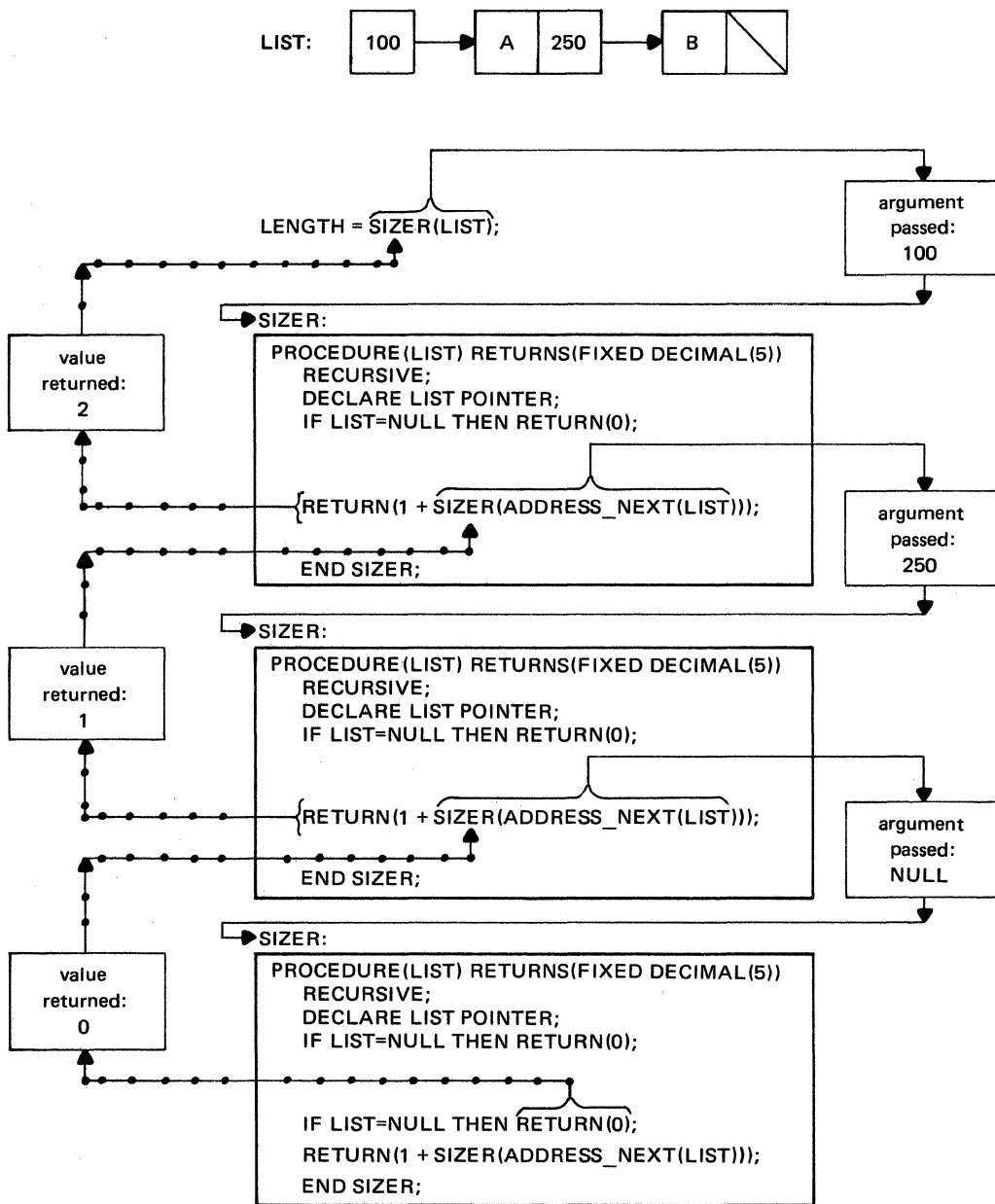


Figure 2.31D. Computing the size of a data list recursively

## FINDR Function

### Purpose

To find the position of the first occurrence of a data item in a data list

### Reference

FINDR(LIST, D)

### Entry-Name Declaration

```
DECLARE FINDR ENTRY(POINTER,  
CHARACTER(1))  
  RETURNS(FIXED DECIMAL(5));
```

### Meaning of Arguments

LIST --the list to be searched  
D --the data item to be found

### Remarks

The function returns a zero value when LIST does not contain D.

### Other Programmer-Defined Procedures Required

GET\_FD and ADDRESS\_NEXT

### Method

FINDR is a recursive function. A null list argument causes a zero value to be returned. When D equals GET\_FD(LIST), the value one is returned; otherwise, the value of the following expression is computed:

$$1 + \text{FINDR}(\text{ADDRESS\_NEXT}(\text{LIST}), D)$$

This expression produces recursive invocation of FINDR while the list argument is not null. The value of the expression is assigned to variable N, which specifies the position of D in the list. When D is not in the list, the value of N equals the size of the list and must be set to zero before FINDR is terminated. Multiplication of N by variable T, which equals one when D is in the list, assures proper control over the value of N. Declaring N and T to be static variables causes all levels of recursion to deal with the same storage for the variable.

```
FINDR:  
  PROCEDURE (LIST, D) RETURNS  
    (FIXED DECIMAL(5)) RECURSIVE;  
  DECLARE  
    LIST POINTER,  
    D CHARACTER(1),  
    (T,N) FIXED DECIMAL(5) STATIC;  
    T,N = 1;  
    IF  
      LIST = NULL  
    THEN  
      DO;  
        T = 0; RETURN(0);  
      END;  
    IF  
      D=GET_FD(LIST)  
    THEN  
      N = 1 + FINDR(ADDRESS_NEXT  
        (LIST),D);  
      RETURN(T*N);  
    END  
  FINDR;
```

Figure 2.32B. A recursive function for finding the position of a data item in a data list

### *Recursive Searching for Data Items in Data Lists*

Figures 2.32A and 2.32B present the FINDR function procedure, which performs a recursive search for the first occurrence of a data item in a data list. When the data item is found, the function returns its position in the list. Absence of the item in the list is indicated by a zero value.

FINDR computes the position of the data item by performing a recursive count similar to the size calculation of SIZER (Figures 2.31A, 2.31B, and 2.31C). When the data item is not in the list, the position count is changed to zero by a zero multiplication factor.

### *Recursive Linking of Data Lists*

Figures 2.33A and 2.33B present the LINKR subroutine procedure, which links two data lists recursively. Each invocation of the subroutine causes the data item at the front of one list to be removed and to be inserted at the end of a second list. When the first list becomes null, recursion stops.

Figure 2.32A. Description of the FINDR function for recursive searching of a data list

### LINKR Subroutine

#### Purpose

To link two data lists

#### Reference

LINKR(LIST1, LIST2)

#### Entry-Name Declaration

```
DECLARE LINKR ENTRY(POINTER, POINTER);
```

#### Meaning of Arguments

LIST1 --the list to which the second list is to be linked

LIST2 --the second list

#### Remarks

LIST2 is null after its data items are linked to LIST1.

#### Other Programmer-Defined Procedures Required

REMOVE\_FD and INSERT\_LD

#### Method

LINKR is a recursive subroutine. Each invocation (initial or recursive) removes the first data item from LIST2 and inserts the data item at the end of LIST1. When LIST2 becomes null, recursion stops, and control returns to the initial point of invocation.

Figure 2.33A. Description of the LINKR subroutine for recursive linking of two data lists

```
LINKR:
  PROCEDURE(LIST1,LIST2) RECURSIVE;
  DECLARE
    (LIST1,LIST2) POINTER;
    IF
      LIST2 = NULL
    THEN
      RETURN;
      CALL INSERT_LD(LIST1,REMOVE_FD
        (LIST2));
      CALL LINKR(LIST1,LIST2);
  END
  LINKR;
```

Figure 2.33B. A recursive subroutine for linking two data lists

### Recursive Testing for Equality of Data Lists

Figures 2.34A and 2.34B present the EQUALR function procedure, which performs a recursive test for equality of two data lists. Recursion occurs when data items in corresponding list positions are equal.

The function returns a one-bit when the lists are equal and a zero-bit when they are not equal.

### EQUALR Function

#### Purpose

To test two data lists for equality

#### Reference

EQUALR(LIST1, LIST2)

#### Entry-Name Declaration

```
DECLARE EQUALR ENTRY(POINTER,
  POINTER)
  RETURNS(BIT(1));
```

#### Meaning of Arguments

LIST1 --the first list

LIST2 --the second list

#### Remarks

The function returns a one-bit when the lists are equal; otherwise, it returns a zero-bit. Null lists are considered equal.

#### Other Programmer-Defined Procedures Required

ADDRESS\_NEXT

#### Method

EQUALR is a recursive function. When the data items in the first position of each list are not equal, the function returns '0'B; otherwise, it returns the value of the following expression:

```
EQUALR(ADDRESS_NEXT(LIST1),
  ADDRESS_NEXT(LIST2))
```

This expression causes recursive invocation of EQUALR as long as corresponding data items are equal. When one (but not both) of the arguments becomes null, recursion stops, and the function returns '0'B. When both arguments become null together, recursion stops, and the function returns '1'B.

Figure 2.34A. Description of the EQUALR function for recursively testing the equality of two data lists

```

EQUALR:  PROCEDURE (LIST1, LIST2) RETURNS
         (BIT(1)) RECURSIVE;
DECLARE
         (LIST1,LIST2) POINTER;
         IF
         LIST1 = LIST2
         THEN
         RETURN('1'B);
         IF
         LIST1 = NULL
         THEN
         RETURN('0'B);
         IF
         LIST2 = NULL
         THEN
         RETURN('0'B);
         IF
         GET_FD(LIST1)≠GET_FD(LIST2)
         THEN
         RETURN('0'B);
         RETURN(EQUALR (ADDRESS_NEXT(LIST1),
         ADDRESS_NEXT(LIST2)));
END
         EQUALR;

```

Figure 2.34B. A recursive function for testing the equality of two data lists

#### *Recursive Comparison of Data Lists*

Figures 2.35A and 2.35B present the COMPARER function procedure, which performs a recursive comparison of two data lists to determine whether one is less than, equal to, or greater than the other. Recursion occurs while data items in corresponding list positions are equal.

For equal lists the function returns the value '11'B. When the first list is less than the second, the function returns '01'B. When the first list is greater than the second, the returned value is '10'B.

#### COMPARER Function

##### Purpose

To determine whether a list is less than, equal to, or greater than another list

##### Reference

COMPARER(LIST1, LIST2)

##### Entry-Name Declaration

```

DECLARE COMPARER ENTRY(POINTER,
POINTER)
RETURNS(BIT(2));

```

##### Meaning of Arguments

LIST1 --the first list  
LIST2 --the second list

##### Remarks

When LIST1 equals LIST2, the function returns '11'B.  
When LIST1 is less than LIST2, the function returns '01'B.  
When LIST1 is greater than LIST2, the function returns '10'B'

##### Other Programmer-Defined Procedures Required

GET\_FD and ADDRESS\_NEXT

##### Method

COMPARER is a recursive function. When the data item in the first position of LIST1 is less than the data item in the first position of LIST2, the function returns '01'B. When it is greater, the function returns '10'B. When the data items are equal, the function returns the value of the following expression:

```

COMPARER(ADDRESS_NEXT(LIST1),
ADDRESS_NEXT(LIST2))

```

This expression causes recursive invocation of COMPARER as long as corresponding data items are equal. Recursion stops when one or both of the arguments becomes null. When both arguments become null together, the function returns '11'B. When only the first argument becomes null, the function returns '01'B. When only the second argument becomes null, the function returns '10'B.

Figure 2.35A. Description of the COMPARER function for the recursive comparison of two data lists

```

COMPARER:
  PROCEDURE (LIST1, LIST2) RETURNS
  (BIT(2)) RECURSIVE;
DECLARE
  (LIST1,LIST2) POINTER,
  (D1,D2) CHARACTER(1);
  IF
  LIST1 = LIST2
  THEN
  RETURN('11'B);
  IF
  LIST1 = NULL
  THEN
  RETURN('01'B);
  IF
  LIST2 = NULL
  THEN
  RETURN('10'B);
  D1 = GET_FD(LIST1);
  D2 = GET_FD(LIST2);
  IF
  D1<D2
  THEN
  RETURN('01'B);
  IF
  D1>D2
  THEN
  RETURN('10'B);
  RETURN(COMPARER(ADDRESS_NEXT(LIST1),
  ADDRESS_NEXT(LIST2)));
END
  COMPARER;

```

Figure 2.35B. A recursive function for comparing two data lists

### Using Simple Data Lists

The similarities between the simple data lists of this chapter and character strings in general provide a variety of list-processing applications concerned with text editing, pattern searching, and symbol manipulation. The following discussions develop programs for eight applications in these areas:

1. Editing cash values
2. Removing edit symbols from cash values
3. Expanding a multiple assignment statement
4. Expanding a picture specification
5. Contracting a picture specification
6. Adding variable-length integers
7. Subtracting variable-length integers
8. Gathering DECLARE statements in a procedure

Each application uses the procedures developed earlier in this chapter, and, although list-processing techniques are not essential for these applications, they improve programming flexibility and provide greater control over storage than is available with more conventional programming methods.

### Editing Cash Values

Figures 2.36A and 2.36B present the \$EDIT subroutine procedure, which accepts an unedited cash value in list form and inserts appropriate edit symbols into the list.

The procedure assumes that the argument list contains no characters other than digits. The number of digits is arbitrary and is restricted only by the size of the list of available storage components, AVAIL.

```

$EDIT:
  PROCEDURE (CASH);
DECLARE
  CASH POINTER,
  (DA, DB) CHARACTER(1),
  S FIXED DECIMAL(5);
ZEROS:
  DO
  WHILE(CASH≠NULL);
  IF
  GET_FD(CASH) ≠ '0'
  THEN
  GO TO
  PERIOD;
  CALL DELETE_FD(CASH);
END ZEROS;
PERIOD:
  S = SIZE(CASH);
  IF S = 0 THEN DO;
  CALL STRING_TO_LIST('$0.00',CASH);
  RETURN;
END;
  IF S = 1 THEN DO;
  CALL STRING_TO_LIST('$0.0' || REMOVE_FD
  (CASH),CASH); RETURN;
END;
  IF S = 2 THEN DO;
  DA = REMOVE_FD(CASH);
  DB = REMOVE_FD(CASH);
  CALL STRING_TO_LIST('$0.' || DA || DB,CASH);
  RETURN;
END;
  CALL INSERT_ND(CASH, S-1, '.');
  S = S-2;
COMMAS:
  S = S-3;
  IF
  S>0
  THEN
  DO;
  CALL INSERT_ND(CASH,S+1,',');
  GO TO
  COMMAS;
END;
$_SIGN:
  CALL INSERT_FD(CASH,'$');
END
  $EDIT;

```

Figure 2.36A. Editing cash values

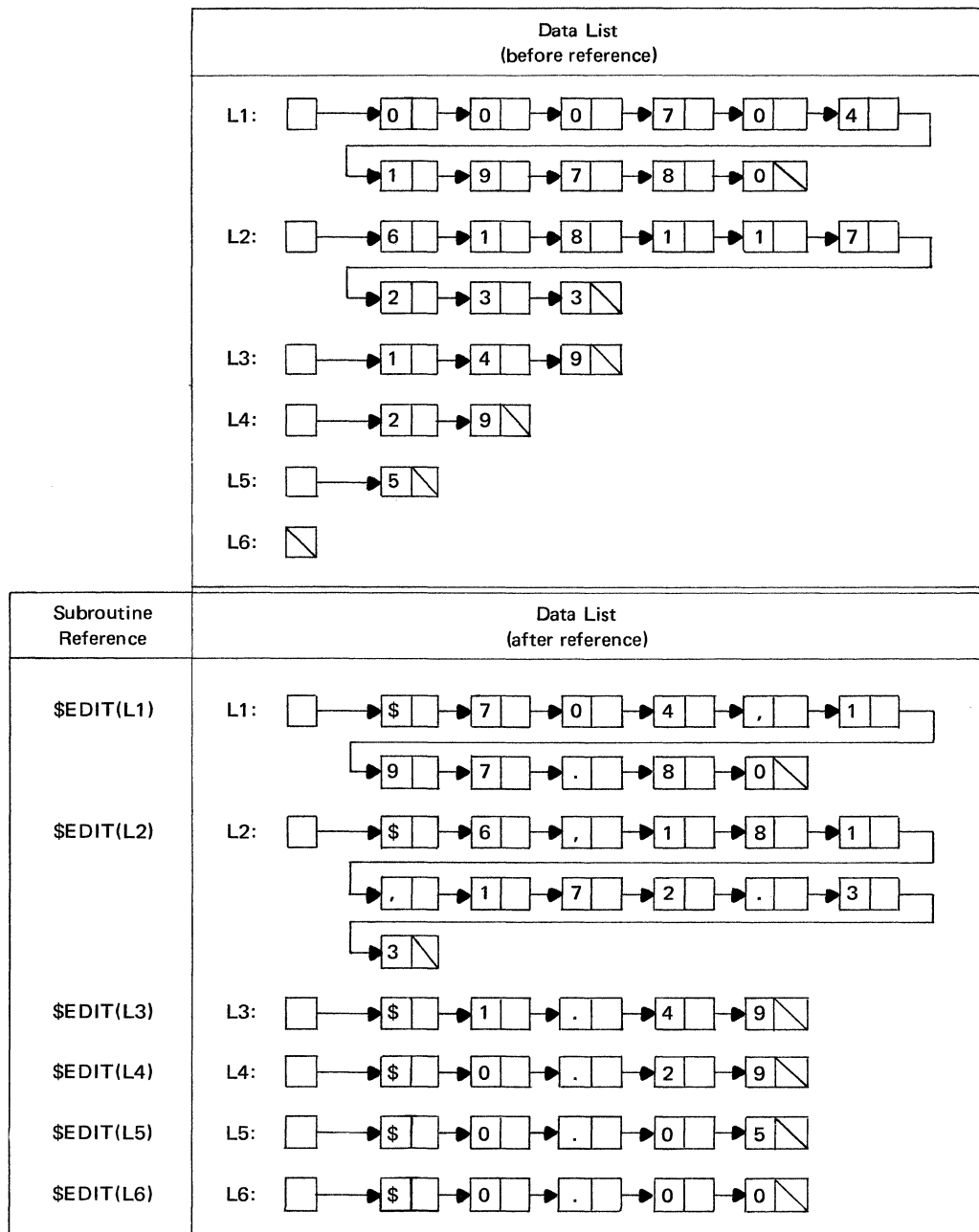


Figure 2.36B. Examples of references to the \$EDIT subroutine

\$EDIT deletes leading zeros in the argument list and inserts a decimal point, commas, and a dollar sign, as shown in Figure 2.36B. Values less than a dollar always receive a zero before the decimal point. The size of the argument list can either decrease or increase, depending on the number of leading zeros that are deleted and the number of edit symbols that are inserted.

Although PL/I provides extensive editing facilities through the PICTURE attribute and the P format item for edit-directed input and output, these facilities may not be

sufficient in particular applications. The \$EDIT procedure shows how list-processing techniques may be used to construct editing procedures for special needs.

#### Removing Edit Symbols from Cash Values

Figures 2.37A and 2.37B present the DE\_EDIT subroutine procedure, which accepts an edited cash value in list form and removes the edit symbols from the list. This procedure may be considered to be the inverse of the \$EDIT procedure (Figures 2.36A and 2.36B).

```

DE_EDIT:
PROCEDURE(CASH);
DECLARE
  CASH POINTER,
  C CHARACTER(1),
  I FIXED DECIMAL(5);
DO
  I = 1 TO SIZE(CASH);
  C = REMOVE_FD(CASH);
  IF
  C = '$'
  THEN
    IF
    C = ','
    THEN
      IF
      C = '.'
      THEN
        CALL INSERT_LD(CASH,C);
      END;
    END;
  END;
DE_EDIT;

```

Figure 2.37A. Removing edit symbols from cash values

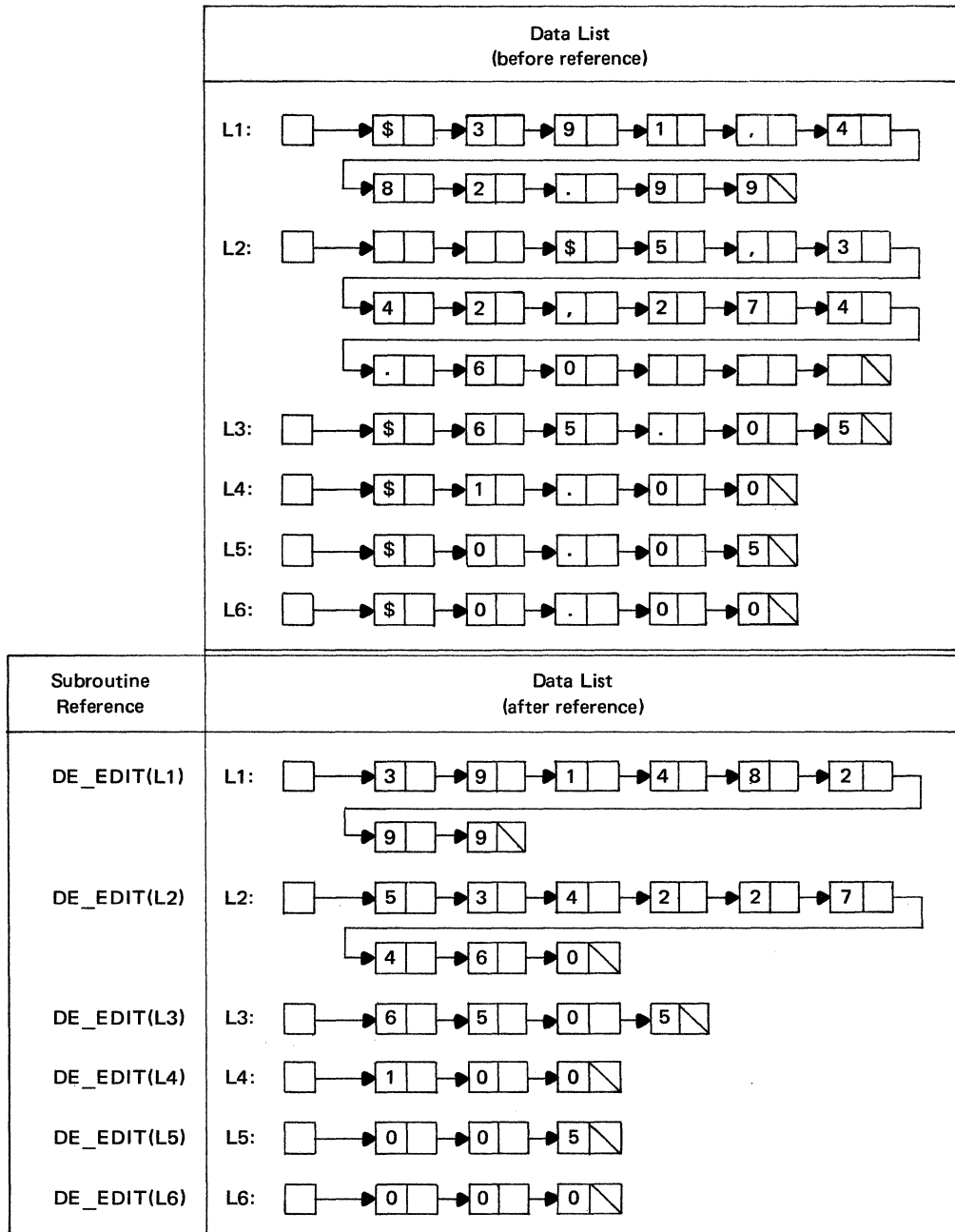


Figure 2.37B. Examples of references to the DE\_EDIT subroutine



### *Expanding a Multiple Assignment Statement*

Figures 2.38A and 2.38B present the A\_EXPAND subroutine procedure, which accepts a multiple assignment statement in list form and expands the list to a series of simple assignment statements.

This procedure provides an elementary example of how list-processing techniques may be used to construct a PL/I compiler. Source statements can be treated as character strings, which are converted to list form and then analyzed by list-processing procedures. Using successive levels of analysis will eventually convert source statements to their equivalent object code.

### *Expanding a Picture Specification*

Figures 2.39A and 2.39B present the P\_EXPAND subroutine procedure, which accepts a PICTURE specification in list form and expands the list so that it contains no repetition factors.

This procedure could be used in a PL/I compiler to convert source text to a standard internal format. It could also be used in a conversion program to alter source text so that it becomes acceptable to a restricted compiler that does not process all PL/I features, such as repetition factors.

```
A_EXPAND:
PROCEDURE (ASSIGNMENT);
DECLARE
  N FIXED DECIMAL(5),
  (ASSIGNMENT, RIGHT_HALF) POINTER;
/* INITIALIZE. */
RIGHT_HALF = NULL;
/* FIND POSITION OF EQUAL SIGN IN
ASSIGNMENT LIST. */
DO
  N = 1 TO SIZE(ASSIGNMENT) BY 1;
  IF
    (GET_ND(ASSIGNMENT, N))= ('=' )
  THEN
    GO TO
    BREAK;
END;
  PUT SKIP LIST('NO EQUAL SIGN');
  GO TO OVER;
/* SPLIT RIGHT HALF FROM ASSIGNMENT
LIST AND INSERT IN RIGHT_HALF
LIST. */
BREAK:
  CALL SPLIT(ASSIGNMENT,N,RIGHT_HALF);
/* REPLACE EACH COMMA IN ASSIGNMENT
LIST WITH RIGHT_HALF LIST. */
DO
  N = 1 BY 1 WHILE
    (N -> SIZE(ASSIGNMENT));
  IF
    (GET_ND(ASSIGNMENT,N))=(',',)
  THEN
    DO;
      CALL DELETE_ND(ASSIGNMENT,N);
      CALL INSERT_LIST(ASSIGNMENT,N,
        RIGHT_HALF);
    END;
END;
/* LINK RIGHT_HALF LIST TO
ASSIGNMENT LIST. */
CALL LINK(ASSIGNMENT, RIGHT_HALF);
OVER:
END
  A_EXPAND;
```

Figure 2.38A. Expanding an assignment statement

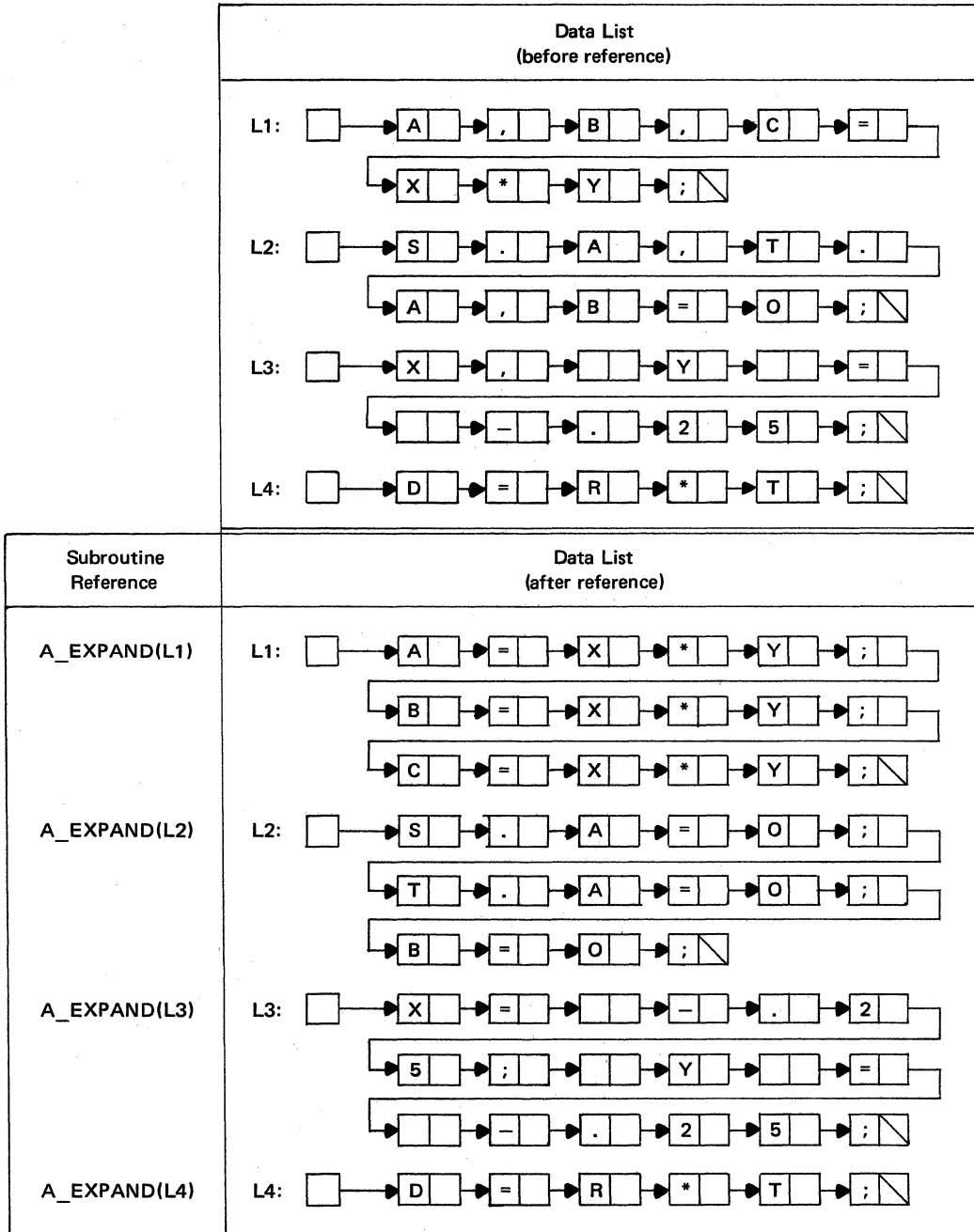


Figure 2.38B. Examples of references to the A\_EXPAND subroutine

```

P_EXPAND:
  PROCEDURE(PICTURE);
  DECLARE
    C CHARACTER(1),
    (PICTURE, EXPANDED) POINTER,
    FACTOR CHARACTER (6) VARYING;
    /* INITIALIZE. */
    EXPANDED = NULL;
    /* INSERT LEFT END OF PICTURE LIST
    (UP TO AND INCLUDING FIRST QUOTE) AT
    FRONT OF EXPANDED LIST. */
  QUOTE1:
    C = REMOVE_FD(PICTURE);
    CALL INSERT_LD(EXPANDED,C);
    IF
      (C ~='''')
    THEN
      GO TO
      QUOTE1;
    /* SCAN REMAINDER OF PICTURE LIST
    (UP TO AND INCLUDING CLOSING
    QUOTE). */
  QUOTE2:
    C = REMOVE_FD(PICTURE);
    IF
      (C = ''')
    THEN
      GO TO
      FINISH;
    /* WHEN CHARACTER IS LEFT PAREN, GO
    TO STATEMENTS THAT OBTAIN REPETITION
    FACTOR. */
    IF
      (C = '(')
    THEN
      GO TO
      REPETITION_FACTOR;
    /* OTHERWISE, INSERT CHARACTER AT
    END OF EXPANDED LIST AND GET NEXT
    CHARACTER. */
    CALL INSERT_LD(EXPANDED, C);

    GO TO
    QUOTE2;
    /* INSERT REPETITION FACTOR IN
    FACTOR. */
  REPETITION_FACTOR:
    FACTOR = ' ';
  NEXT_CHARACTER:
    C = REMOVE_FD(PICTURE);
    IF
      (C = '(')
    THEN
      GO TO
      EXPAND;
    EXPAND:
      FACTOR = FACTOR||C;
      GO TO
      NEXT_CHARACTER;
    /* INSERT NEXT CHARACTER THE
    NUMBER OF TIMES SPECIFIED BY FACTOR.
    */
  EXPAND:
    C = REMOVE_FD(PICTURE);
    DO
      N = 1 TO FACTOR;
      CALL INSERT_LD(EXPANDED,C);
    END;
    /* GET NEXT CHARACTER. */
    GO TO
    QUOTE2;
    /* AT THIS POINT, EXPANSION IS
    COMPLETE. INSERT FINAL QUOTE AT END
    OF EXPANDED LIST AND LINK EXPANDED
    LIST TO PICTURE LIST. */
  FINISH:
    CALL INSERT_LD(EXPANDED,C);
    CALL LINK(PICTURE, EXPANDED);
  END
  P_EXPAND;

```

Figure 2.39A. Expanding a picture specification

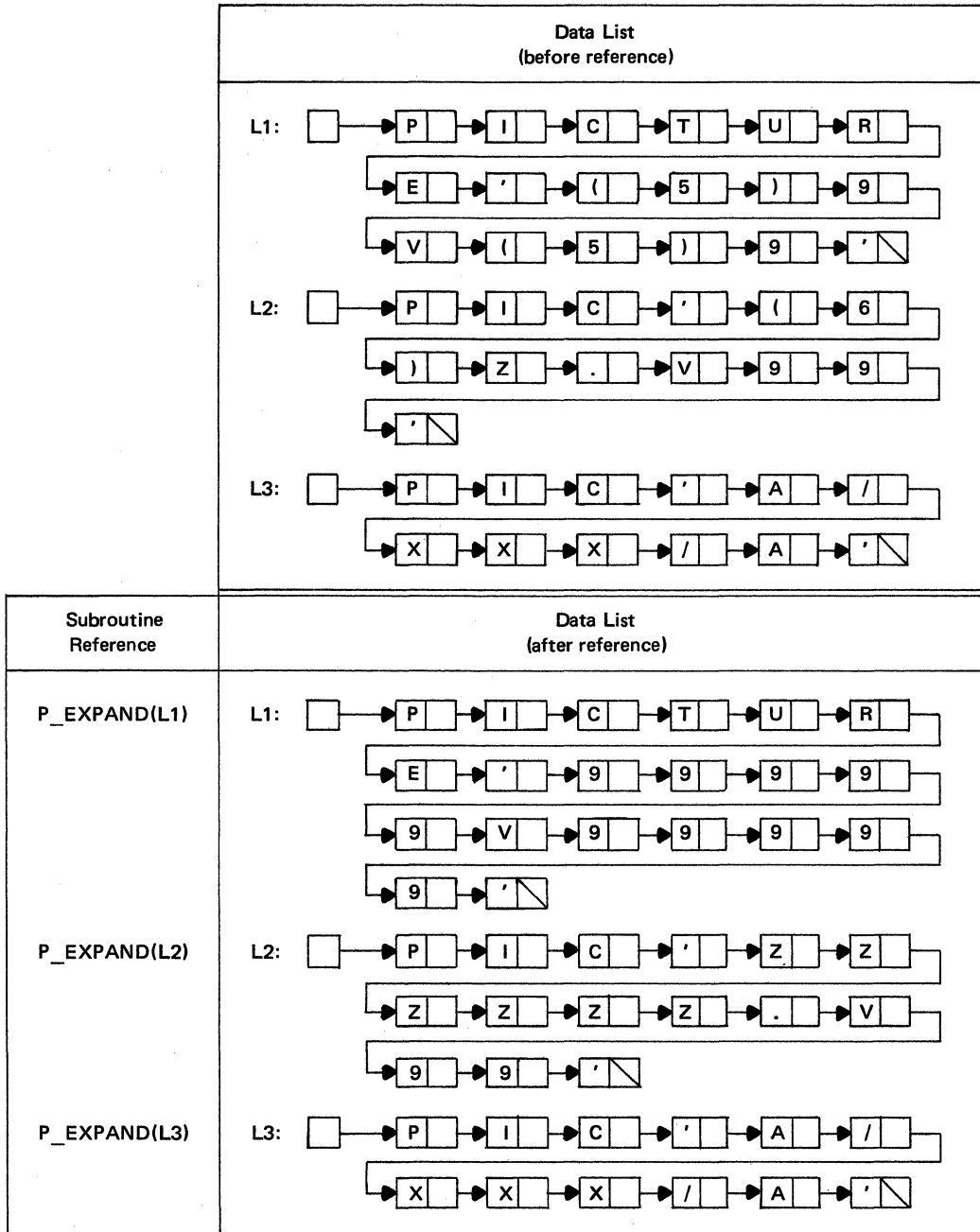


Figure 2.39B. Examples of references to the P\_EXPAND subroutine

### Contracting a Picture Specification

Figures 2.40A and 2.40B present the P\_CNTRT subroutine procedure, which accepts a PICTURE specification in list form and contracts the list by replacing repeated character sequences five or more in length with repetition factors.

This procedure may be treated as the inverse of the P\_EXPAND procedure (Figures 2.39A and 2.39B).

```

P_CNTRT:
PROCEDURE (PICTURE);
DECLARE
  C CHARACTER(1),
  (PICTURE, FACTOR_LIST) POINTER,
  (N,M,L,I) FIXED DECIMAL(5),
  REPETITION_FACTOR PICTURE 'ZZZZ9',
  FACTOR_STRING CHARACTER(5);
  /* INITIALIZE. */
  FACTOR_LIST = NULL;
  /* FIND POSITION OF FIRST QUOTE IN
  PICTURE LIST. */
DO
  N = 1 BY 1;
  IF
  (GET_ND(PICTURE,N) = '')
  THEN
  DO;
  N = N + 1;
  GO TO
  NEXT_CHARACTER;
END;
END;
  /* SCAN PICTURE LIST FOR CHARACTER
  SEQUENCES OF LENGTH FIVE OR MORE. */
NEXT_CHARACTER:
  C = GET_ND(PICTURE,N);
  /* IF SECOND QUOTE IS FOUND,
  PROGRAM IS FINISHED. */
  IF
  (C = '')
  THEN
  RETURN;
  /* TEST FOR REPEATED SEQUENCE. */
DO
  M = N + 1 BY 1;
  IF
  (GET_ND(PICTURE,M)~=C)
  THEN
  GO TO
  LENGTH_TEST;
END;
  /* IF SEQUENCE-LENGTH IS LESS THAN
  FIVE, RESUME SEARCH */
LENGTH_TEST:
  L = M - N;
  IF
  (L<5)

```

Figure 2.40A. Contracting a picture specification

### Adding Variable-Length Integers

Figures 2.41A and 2.41B present the ADD\_INT subroutine procedure, which adds two variable-length integers in list form. The length of each integer is arbitrary and is limited only by available storage. The integers need not have equal lengths.

ADD\_INT assumes that the argument lists contain only numeric characters (signs and decimal points are not permitted). The integer in the first argument list is added to the integer in the second argument list. The third argument receives the result list.

```

THEN
DO;
  N = M;
  GO TO
  NEXT_CHARACTER;
END;
  /* CONVERT SEQUENCE-LENGTH TO
  EDITED STRING WITH LEADING ZEROS
  SUPPRESSED. */
  REPETITION_FACTOR = L;
  /* ASSIGN EDITED STRING TO
  FACTOR_STRING, WHICH HAS ATTRIBUTES
  REQUIRED BY THE STRING_TO_LIST
  SUBROUTINE. */
  FACTOR_STRING = REPETITION_FACTOR;
  /* CONVERT FACTOR_STRING TO
  FACTOR_LIST. */
  CALL STRING_TO_LIST(FACTOR_STRING,
  FACTOR_LIST);
  /* DELETE LEADING BLANKS FROM
  FACTOR_LIST. */
DO I = 1 BY 1;
  IF
  (GET_FD(FACTOR_LIST) ~= ' ')
  THEN
  GO TO
  PARENTHESES;
  CALL DELETE_FD(FACTOR_LIST);
END;
  /* INSERT PARENTHESES AT FRONT AND
  BACK OF FACTOR LIST. */
PARENTHESES:
  CALL INSERT_FD(FACTOR_LIST, '(');
  CALL INSERT_LD(FACTOR_LIST, ')');
  /* CONTRACT REPEATED SEQUENCE AND
  INSERT FACTOR_LIST. */
  CALL DELETE_SUB(PICTURE, N, L-1);
  CALL INSERT_LIST(PICTURE,N,
  FACTOR_LIST);
  /* SET N TO POSITION OF NEXT
  CHARACTER. DELETE FACTOR_LIST. GO TO
  NEXT_CHARACTER. */
  N = N + SIZE(FACTOR_LIST) + 1;
  CALL DELETE_LIST(FACTOR_LIST);
  GO TO
  NEXT_CHARACTER;
END
  P_CNTRT;

```

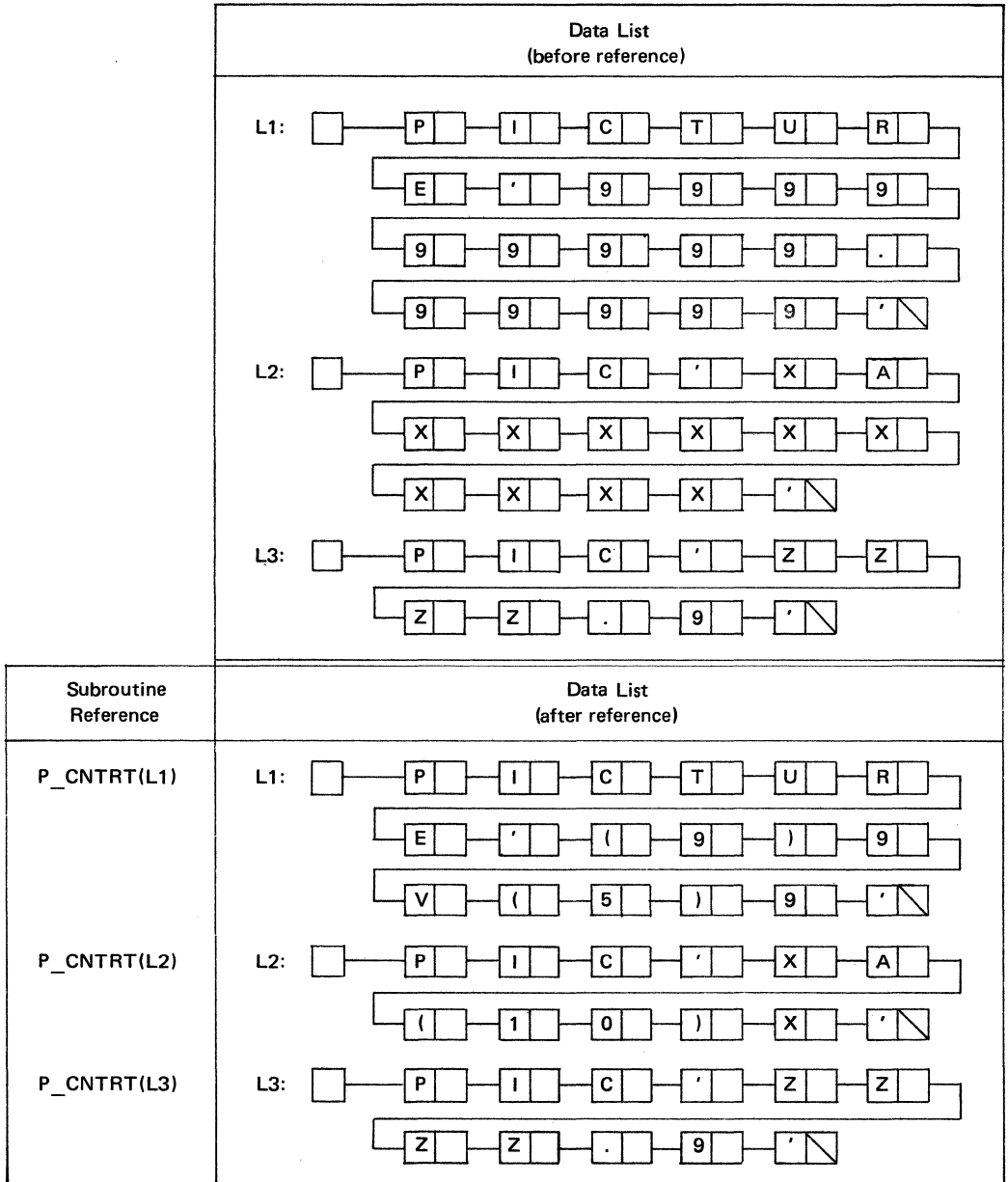


Figure 2.40B. Examples of references to the P\_CNTRT subroutine

```

ADD_INT:
PROCEDURE (ADDEND1, ADDEND2, SUM);
DECLARE
  (ADDEND1, ADDEND2, SUM) POINTER,
  (CARRY, DIGIT1, DIGIT2)
  CHARACTER(1),
  1 SUM_TABLE(0:1,0:9,0:9),
  2 TENS CHARACTER(1),
  2 UNITS CHARACTER(1),
  SUMS_OF_DIGITS_AND_CARRY
  CHARACTER(400) DEFINED SUM_TABLE;
/* INITIALIZE SUM_TABLE. */
SUMS_OF_DIGITS_AND_CARRY =
  '00010203040506070809'
  || '01020304050607080910'
  || '02030405060708091011'
  || '03040506070809101112'
  || '04050607080910111213'
  || '05060708091011121314'
  || '06070809101112131415'
  || '07080910111213141516'
  || '08091011121314151617'
  || '09101112131415161718'
  || '01020304050607080910'
  || '02030405060708091011'
  || '03040506070809101112'
  || '04050607080910111213'
  || '05060708091011121314'
  || '06070809101112131415'
  || '07080910111213141516'
  || '08091011121314151617'
  || '09101112131415161718'
  || '10111213141516171819';
/* CLEAR SUM AND CARRY. */
SUM = NULL;
CARRY = '0';
/* TEST FOR NULL LISTS */
IF ((ADDEND1 = NULL) &
  (ADDEND2 = NULL))
  THEN DO;
  CALL INSERT_FD(SUM, '0');
  RETURN;
  END;
IF ADDEND1 = NULL
  THEN DO;
  SUM = ADDEND2;
  RETURN;
  END;
IF ADDEND2 = NULL
  THEN DO;
  SUM = ADDEND1;
  RETURN;
  END;
/* INSERT LEADING ZERO AT FRONT OF
  ADDEND1 LIST AND ADDEND2 LIST. */
CALL INSERT_FD(ADDEND1, '0');
CALL INSERT_FD(ADDEND2, '0');
/* ENTER LOOP THAT COMPUTES SUM OF
  ADDEND1 AND ADDEND2. */
LOOP:
DO WHILE((ADDEND1 != NULL) |
  (ADDEND2 != NULL));
/* REMOVE RIGHTMOST DIGITS OF
  ADDENDS, AND ASSIGN THEM TO DIGIT1
  AND DIGIT2. */
DIGIT1 = REMOVE_LD(ADDEND1);
  IF
  (DIGIT1 = ' ')
  THEN
  DIGIT1 = '0';
  DIGIT2 = REMOVE_LD(ADDEND2);
  IF
  (DIGIT2 = ' ')
  THEN
  DIGIT2 = '0';
/* OBTAIN UNITS DIGIT FOR THE SUM OF
  CARRY, DIGIT2, AND DIGIT1; AND
  INSERT UNITS DIGIT IN THE FIRST
  POSITION OF SUM LIST. */
CALL INSERT_FD(SUM, UNITS(CARRY,
  DIGIT2, DIGIT1));
/* SET CARRY EQUAL TO TENS DIGIT
  FROM THE SUM OF CARRY, DIGIT2, AND
  DIGIT1. */
CARRY=TENS(CARRY,DIGIT2,DIGIT1);
END LOOP;
/* REMOVE LEADING ZEROS FROM SUM */
ZEROS:
  IF (GET_FD(SUM) = '0')
  THEN DO;
  CALL DELETE_FD(SUM);
  GO TO ZEROS;
  END;
/* IF SUM IS NULL, INSERT 0 */
IF SUM = NULL
  THEN CALL INSERT_FD(SUM, '0');
END
ADD_INT;

```

Figure 2.41A. Adding variable-length integers

		DIGIT1									
		0	1	2	3	4	5	6	7	8	9
D	0	00	01	02	03	04	05	06	07	08	09
I	1	01	02	03	04	05	06	07	08	09	10
G	2	02	03	04	05	06	07	08	09	10	11
I	3	03	04	05	06	07	08	09	10	11	12
T	4	04	05	06	07	08	09	10	11	12	13
2	5	05	06	07	08	09	10	11	12	13	14
	6	06	07	08	09	10	11	12	13	14	15
	7	07	08	09	10	11	12	13	14	15	16
	8	08	09	10	11	12	13	14	15	16	17
	9	09	10	11	12	13	14	15	16	17	18

Sums of digits when previous carry is zero

		DIGIT1									
		0	1	2	3	4	5	6	7	8	9
D	0	01	02	03	04	05	06	07	08	09	10
I	1	02	03	04	05	06	07	08	09	10	11
G	2	03	04	05	06	07	08	09	10	11	12
I	3	04	05	06	07	08	09	10	11	12	13
T	4	05	06	07	08	09	10	11	12	13	14
2	5	06	07	08	09	10	11	12	13	14	15
	6	07	08	09	10	11	12	13	14	15	16
	7	08	09	10	11	12	13	14	15	16	17
	8	09	10	11	12	13	14	15	16	17	18
	9	10	11	12	13	14	15	16	17	18	19

Sums of digits when previous carry is one

Figure 2.41B. Tables for the sums of two digits and a previous carry

The procedure uses a table lookup technique to add the two integers digit by digit from right to left. Figure 2.41B contains illustrations of the tables used by ADD\_INT. The first table gives the sums of digit pairs when no carry is involved from the previous digit sum. The sums in the second table account for a carry of one from the previous digit sum. The sum of the rightmost digits in the two integers always assumes a previous carry of zero. The digit in the tens position of a sum specifies the carry for the next digit pair.

The following example shows how the tables are used:

1	1	0	0	Carried digits
2	9	7	8	Second integer
		3	1	First integer
3	0	0	9	Sum

The first digit pair is 1 and 8, and it assumes a previous carry of 0; therefore, the first table is used to obtain the sum 09. The 9 in the sum becomes the units digit in the final result, and the 0 becomes the carry for the second digit pair.

The second digit pair is 3 and 7, and its sum (10) is obtained from the first table. The 0 in this sum becomes the tens digit in the final result, and the 1 becomes the carry for the third digit pair.

At this point, all digits have been used in the first integer; therefore, zeros are assumed in the remaining positions until the final result is obtained. The third digit pair, then, is 0 and 9, and its associated sum (10) is obtained from the second table, since the previous carry is 1. The 0 in this sum forms the hundreds digit of the final result, and the 1 is carried to the next position.

The fourth digit pair is 0 and 2, and its associated sum (03) is obtained from the second table. The 3 in this sum becomes the leftmost digit in the final result, which is 3009.

ADD\_INT combines the tables in Figure 2.41B into one three-dimensional array. It is then possible to use the two digits being added and the carry digit as subscript values in references to the associated sum in the array. The sums are also arranged as structures in the array, so that the units and tens digits of each sum can be referred to separately.

The aggregate total is contained in the list named SUM.

The advantage of using data lists in this application is that they do not impose a specific maximum length on the integers being added. As long as the combined lengths of the two integers do not exceed available list storage, they can have any individual lengths.

#### Subtracting Variable-Length Integers

Figures 2.42A and 2.42B present the SUB\_INT subroutine procedure, which subtracts two variable-length integers in list form. The length of each integer is arbitrary and is limited only by available storage. The integers need not have equal lengths.



```

SUB_INT:
  PROCEDURE (SUBTRAHEND, MINUEND,
    DIFFERENCE);
  DECLARE
    (BORROW, DIGIT1, DIGIT2)
    CHARACTER(1),
    (SUBTRAHEND, MINUEND, DIFFERENCE)
    POINTER,
    1 DIFFERENCE_TABLE(0:1,0:9,0:9),
    2 TENS CHARACTER(1),
    2 UNITS CHARACTER(1),
    DIFFERENCES_AND_BORROWED_DIGITS
    CHARACTER(400)
    DEFINED DIFFERENCE_TABLE;
  /* INITIALIZE DIFFERENCE_TABLE. */
  DIFFERENCES_AND_BORROWED_DIGITS =
    '00191817161514131211'
  || '01001918171615141312'
  || '02010019181716151413'
  || '03020100191817161514'
  || '04030201001918171615'
  || '05040302010019181716'
  || '06050403020100191817'
  || '07060504030201001918'
  || '08070605040302010019'
  || '09080706050403020100'
  || '1918171615141312110'
  || '00191817161514131211'
  || '01001918171615141312'
  || '02010019181716151413'
  || '03020100191817161514'
  || '04030201001918171615'
  || '05040302010019181716'
  || '06050403020100191817'
  || '07060504030201001918'
  || '08070605040302010019';
  /* CLEAR DIFFERENCE AND BORROW */
  DIFFERENCE = NULL;
  BORROW = '0';
  /* TEST FOR NULL LISTS */
  IF ((SUBTRAHEND = NULL) &
    (MINUEND = NULL))
  THEN DO;
  CALL INSERT_FD(DIFFERENCE,'0');
  RETURN;
  END;
  IF SUBTRAHEND = NULL
  THEN DO;
    DIFFERENCE = MINUEND;
    RETURN;
  END;
  /* ENTER LOOP THAT COMPUTES
  DIFFERENCE OF SUBTRAHEND AND
  MINUEND. */
  LOOP:
    DO WHILE((SUBTRAHEND~=NULL) |
      (MINUEND ~= NULL));
    /* REMOVE RIGHTMOST DIGITS OF
    SUBTRAHEND AND MINUEND AND ASSIGN
    THEM TO DIGIT1 AND DIGIT2. */
    DIGIT1 = REMOVE_LD(SUBTRAHEND);
    IF
      (DIGIT1 = ' ')
    THEN
      DIGIT1 = '0';
      DIGIT2 = REMOVE_LD(MINUEND);
      IF
        (DIGIT2 = ' ')
      THEN
        DIGIT2 = '0';
        /* OBTAIN UNITS DIGIT FROM THE
        DIFFERENCE OF DIGIT2 AND THE SUM OF
        DIGIT1 AND BORROW;
        AND INSERT UNITS DIGIT IN THE
        FIRST POSITION OF DIFFERENCE LIST. */
        CALL INSERT_FD(DIFFERENCE, UNITS
          (BORROW, DIGIT2, DIGIT1));
        /* SET BORROW EQUAL TO TENS DIGIT
        FROM THE DIFFERENCE OF DIGIT2 AND
        THE SUM OF DIGIT1 AND BORROW. */
        BORROW = TENS(BORROW, DIGIT2,
          DIGIT1);
      END LOOP;
      /* REMOVE LEADING ZEROS FROM
      DIFFERENCE LIST */
      ZEROS:
        IF (GET_FD(DIFFERENCE) = '0')
        THEN DO;
          CALL DELETE_FD(DIFFERENCE);
          GO TO ZEROS;
        END;
        /* IF DIFFERENCE=NULL, INSERT 0 */
        IF DIFFERENCE = NULL
        THEN CALL INSERT_FD(DIFFERENCE,'0');
    END
    SUB_INT;

```

Figure 2.42A. Subtracting variable-length integers

		SUBTRAHEND									
		0	1	2	3	4	5	6	7	8	9
M I N U E N D	0	00	19	18	17	16	15	14	13	12	11
	1	01	00	19	18	17	16	15	14	13	12
	2	02	01	00	19	18	17	16	15	14	13
	3	03	02	01	00	19	18	17	16	15	14
	4	04	03	02	01	00	19	18	17	16	15
	5	05	04	03	02	01	00	19	18	17	16
	6	06	05	04	03	02	01	00	19	18	17
	7	07	06	05	04	03	02	01	00	19	18
	8	08	07	06	05	04	03	02	01	00	19
	9	09	08	07	06	05	04	03	02	01	00

Differences of digits when previous borrow is zero

		SUBTRAHEND									
		0	1	2	3	4	5	6	7	8	9
M I N U E N D	0	19	18	17	16	15	14	13	12	11	10
	1	00	19	18	17	16	15	14	13	12	11
	2	01	00	19	18	17	16	15	14	13	12
	3	02	01	00	19	18	17	16	15	14	13
	4	03	02	01	00	19	18	17	16	15	14
	5	04	03	02	01	00	19	18	17	16	15
	6	05	04	03	02	01	00	19	18	17	16
	7	06	05	04	03	02	01	00	19	18	17
	8	07	06	05	04	03	02	01	00	19	18
	9	08	07	06	05	04	03	02	01	00	19

Differences of digits when previous borrow is one

Figure 2.42B. Tables for the differences of two digits with and without a previous borrow

SUB\_INT assumes that the argument lists contain only numeric characters (signs and decimal points are not permitted). The integer in the first argument list (the subtrahend) is subtracted from the integer in the second argument list (the minuend). The value of the subtrahend must not exceed the value of the minuend. The third argument receives the result list.

The procedure uses a table lookup technique to subtract the two integers digit by digit from right to left. Figure 2.42B contains illustrations of the tables used by SUB\_INT. The first table gives the differences of digit pairs when no borrow was required by the previous digit pair to the right. The entries in the second table account for a borrow of one by the previous digit pair to the right. The difference of the rightmost digits in the two integers always assumes a previous borrow of zero.

The digit in the tens position of a table entry specifies the amount borrowed from the minuend digit on the left to obtain the difference of two digits. This difference appears as the units digit in the table entry.

The following example shows how the tables are used:

1	1	0	0	Borrowed digits
3	0	0	9	Minuend
		3	1	Subtrahend
2	9	7	8	Difference

The first digit pair is 1 and 9, and it assumes a previous borrow of 0; therefore, the first table is used to obtain the associated entry 08. The 8 in this entry becomes the units digit in the final result, and the 0 specifies the amount borrowed from the minuend digit on the left.

The second digit pair is 3 and 0, and its associated entry (17) is obtained from the first table. The 7 in this entry becomes the tens digit in the final result, and the 1 specifies the amount borrowed.

At this point, all digits have been used in the subtrahend; therefore, zeros are assumed in the remaining positions of the subtrahend until the final result is obtained. The third digit pair, then, is 0 and 0, and its associated entry (19) is obtained from the second table, since the previous borrow is 1. The 9 in this entry becomes the hundreds digit of the final result, and the 1 specifies the amount borrowed.

The fourth digit pair is 0 and 3, and its associated entry (02) is obtained from the second table. The 2 in this entry becomes the leftmost digit in the final result, which is 2978.

SUB\_INT combines the tables in Figure 2.42B into one three-dimensional array. It is then possible to use the two digits being subtracted and the previously borrowed digit as subscript values in references to the associated entry in the array. The entries are also arranged as structures in the array, so that the units and tens digits of each entry can be referred to separately.

#### Gathering Declare Statements in a Procedure

So far the applications in this section have been developed in subroutine form. As a result, the subroutines have assumed the existence of a list of available storage components. The present application is developed as a complete program, which generates a list of available storage components for list-processing techniques within the program.

Figure 2.43 presents the DGATHER program, which gathers the DECLARE statements in a PL/I procedure so that they appear as a single DECLARE statement at the front of the procedure.

```

DGATHER:
PROCEDURE:
DECLARE
    SPACE AREA(32767),
    (DECLARE_LIST, NEW_DECLARE,
    MARGIN, PROCEDURE, AVAIL EXTERNAL,
    SAVE_LIST) POINTER,
    CARD(80) CHARACTER(1),
    C CHARACTER(1),
    DECLARE_STRING CHARACTER(9),
    DECLARE_TABLE(4) CHARACTER(9),
    (STRING_SWITCH, COMMENT_SWITCH,
    FINISH_SWITCH)
    BIT(1) INITIAL ('0'B),
    (N,I,FIRST_SEMICOLON,PROCEDURE_SIZE)
    FIXED DECIMAL(5) INITIAL(0);
/* AT THE END OF THE SYSIN FILE,
COMPUTE THE SIZE OF THE PROCEDURE
LIST AND GO TO SCAN. */
ON ENDFILE(SYSIN)

BEGIN;
    PROCEDURE_SIZE = SIZE(PROCEDURE);
GO TO
SCAN;
END;
/* INITIALIZE. */
AVAIL, MARGIN, PROCEDURE,
DECLARE_LIST,
NEW_DECLARE, SAVE_LIST = NULL;
CALL AREA_OPEN(SPACE,AVAIL);
DECLARE_TABLE(1) = ' DECLARE ';
DECLARE_TABLE(2) = ' DECLARE (';
DECLARE_TABLE(3) = ' DECLARE (';
DECLARE_TABLE(4) = ' DECLARE (';
CALL STRING_TO_LIST('DECLARE ',
NEW_DECLARE);
/* READ INPUT CARDS. FOR EACH CARD,
INSERT COLUMNS 1 AND 73 THROUGH 80
AT END OF MARGIN LIST, AND
COLUMNS 2 THROUGH 72 AT END OF
PROCEDURE LIST. */
INPUT:
GET
    EDIT (CARD) (80 A(1));
    CALL INSERT_LD(MARGIN, CARD(1));
DO
    N = 2 TO 72;
    CALL INSERT_LD(PROCEDURE,CARD(N));
END;
DO
    N = 73 TO 80;
    CALL INSERT_LD(MARGIN, CARD(N));
END;
GO TO
INPUT;
/* SCAN PROCEDURE LIST FOR DECLARE
STATEMENTS. */
SCAN:
DO
    N = 1 TO PROCEDURE_SIZE;
/* GET NEXT CHARACTER AND TEST IT. */
    C = GET_ND(PROCEDURE, N);
    IF
        (C='''')
    THEN
        GO TO
        STRING_TEST;
        IF
            (C='/'')
                THEN
                    GO TO
                    COMMENT_TEST;
                    IF
                        STRING_SWITCH
                    THEN
                        GO TO
                        END_SCAN;
                        IF
                            COMMENT_SWITCH
                        THEN
                            GO TO
                            END_SCAN;
                            IF
                                (FIRST_SEMICOLON = 0)
                            THEN
                                IF
                                    (C = ';'')
                                THEN
                                    FIRST_SEMICOLON = N;
                                    /* TEST FOR DECLARE STATEMENT. */
                                FIND_DECLARE:
                                    CALL ASSIGN_SUB(DECLARE_LIST,
                                        PROCEDURE, N, 9);
                                    CALL LIST_TO_STRING(DECLARE_LIST,
                                        DECLARE_STRING);
                                DO
                                    I = 1 TO 4;
                                    IF
                                        (DECLARE_STRING = DECLARE_TABLE(I))
                                    THEN
                                        DO;
                                            /* REPLACE KEYWORD DECLARE WITH BLANKS */
                                            DO N = N + 1 TO N + 7;
                                            CALL REPLACE_ND(PROCEDURE,N,' ');
                                            END;
                                            GO TO
                                            PROCESS_DECLARE;
                                        END;
                                        END;
                                        GO TO
                                        END_SCAN;
                                        /* PROCESS DECLARE STATEMENT. */
                                    PROCESS_DECLARE:
                                        /* INSERT DECLARE STATEMENT
                                        WITHOUT KEYWORD DECLARE AND
                                        TERMINATING SEMICOLON AT END OF
                                        NEW_DECLARE LIST. */
                                        DO N = N BY 1;
                                        C = GET_ND(PROCEDURE,N);
                                        IF C = ';' THEN DO;
                                            CALL INSERT_LD(NEW_DECLARE,C);
                                            CALL REPLACE_ND(PROCEDURE,N,' ');
                                            END;
                                        ELSE DO;
                                            CALL INSERT_LD(NEW_DECLARE,' ');
                                            CALL REPLACE_ND(PROCEDURE,N,' ');
                                            N = N - 1;
                                            GO TO END_SCAN;
                                        END;
                                        END PROCESS_DECLARE;
                                        /* TEST FOR START OR END OF
                                        STRING. */
                                    STRING_TEST:
                                        IF
                                            STRING_SWITCH
                                        THEN
                                            IF
                                                (GET_ND(PROCEDURE, N + 1) = ' ''')

```

Figure 2.43. Gathering declare statements

```

THEN
  N = N + 1;
ELSE
  STRING_SWITCH = '0'B;
ELSE
  STRING_SWITCH = '1'B;
  GO TO
  END_SCAN;
  /* TEST FOR START OR END OF
  COMMENT. */
COMMENT_TEST:
  IF
  COMMENT_SWITCH
  THEN
    IF
    GET_ND(PROCEDURE,N-1) = '*'
  THEN
    COMMENT_SWITCH = '0'B;
  ELSE;
  ELSE
    IF
    GET_ND(PROCEDURE, N + 1) = '*'
  THEN
    COMMENT_SWITCH = '1'B;
    N = N + 1;
END;
END_SCAN:
END
SCAN;
/* TERMINATE NEW_DECLARE LIST
WITH A SEMICOLON */
CALL REPLACE_LD(NEW_DECLARE,';');
/* RECONSTRUCT ORIGINAL PROCEDURE
AND PLACE NEW_DECLARE LIST BEHIND
LEADING PROCEDURE STATEMENT. */
/* SPLIT PROCEDURE LIST AFTER FIRST
SEMICOLON. */
CALL SPLIT(PROCEDURE,
FIRST_SEMICOLON + 1, SAVE_LIST);
PUT
PAGE;
/* PRINT MERGED PROCEDURE AND
MARGIN LISTS. */
OUTPUT1:
C = REMOVE_FD(MARGIN);
PUT
EDIT(C)(A(1));
DO
  N = 1 TO 71;
  C = REMOVE_FD(PROCEDURE);
  PUT
  EDIT(C)(A(1));
END;
DO
  N = 1 TO 8;
  C = REMOVE_FD(MARGIN);
  PUT
  EDIT(C)(A(1));
END;
END;
PUT
SKIP;
IF
(PROCEDURE->=NULL)
THEN
  GO TO
  OUTPUT1;
  /* IF FINISH_SWITCH IS ONE, GO TO
  END_DCL_GATHER. */
  IF
  FINISH_SWITCH
  THEN
    GO TO
    END_DCL_GATHER;
    /* PRINT NEW_DECLARE LIST WITH BLANK
    MARGINS. */
  OUTPUT2:
  PUT
  EDIT('')(A(1));
  DO
    N = 1 TO 71;
    C = REMOVE_FD(NEW_DECLARE);
    PUT
    EDIT(C)(A(1));
  END;
  PUT
  SKIP;
  IF
  (NEW_DECLARE->=NULL)
  THEN
    GO TO
    OUTPUT2;
    /* LINK REMAINDER OF PROCEDURE IN
    SAVE_LIST TO PROCEDURE_LIST. */
    CALL LINK(PROCEDURE, SAVE_LIST);
    /* POSITION REMAINDER OF PROCEDURE
    SO THAT ORIGINAL FORMAT IS NOT
    DISTURBED. */
    I = MOD(FIRST_SEMICOLON, 71) + 1;
    PUT
    EDIT('')(COLUMN(I), A(1));
    DO
      N = 1 TO (71-I);
      C = REMOVE_FD(PROCEDURE);
      PUT
      EDIT(C)(A(1));
    END;
    PUT
    SKIP;
    /* TURN FINISH_SWITCH ON, AND
    CONTINUE PRINTING REMAINDER OF
    PROCEDURE_LIST AT OUTPUT1. */
    FINISH_SWITCH = '1'B;
    GO TO
    OUTPUT1;
  END_DCL_GATHER:
  END
  DGATHER;

```

Figure 2.43. Gathering declare statement (Continued)

DGATHER obtains the procedure from the standard system-input file (SYSIN), gathers the DECLARE statements at the front of the procedure, and prints the procedure on the standard system-output file (SYSPRINT). As a simplification, DGATHER does not permit other procedures or begin blocks to appear in the procedure being processed. The number of DECLARE statements in the procedure is arbitrary, and they may appear anywhere within the procedure.

The major processing steps performed by DGATHER are to:

1. Generate the list of available storage components, AVAIL, in the area called SPACE
2. Obtain successive input cards and insert column 1 and columns 73 through 80 into the MARGIN list and columns 2 through 72 into the PROCEDURE list
3. Search for DECLARE statements and form a NEW\_DECLARE list; skip character strings and PL/I comments when searching for DECLARE statements
4. Remove all DECLARE statements from the procedure and insert the NEW\_DECLARE list behind the leading PROCEDURE statement
5. Print the procedure, maintaining as much of the original format as possible

These steps provide an elementary illustration of how list-processing techniques can be used by a compiler to reorganize and analyze a source program before it is translated into an object program.

*Note:* All of the sample programs in Chapter 2 were internal to a single containing procedure. Each entry name was declared in the containing procedure.

### REVIEW OF SIMPLE DATA LISTS

Chapter 2 shows how to generate allocations of based storage throughout an area and how to link such allocations into a list of available storage components (see AVAIL in Figure 2.44). It also shows how to develop subroutines and functions that use available storage components to form new lists. The subroutines and functions are developed in hierarchical fashion so that routines concerned with the primitive aspects of storage manipulation can be used in turn to create higher level procedures. This approach limits the number of procedures that deal with environmental factors and permits the complete collection of subroutines and functions to possess an application-oriented emphasis.

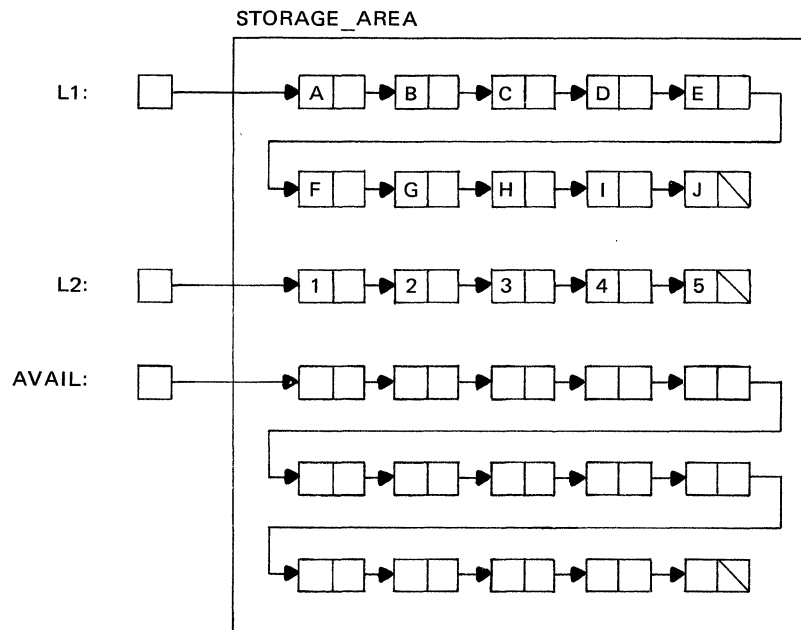


Figure 2.44. Simple data lists linked within an area

The main advantage of simple data lists over array and structure organizations is that lists need reserve only the storage they are currently using. As a list grows, new storage is obtained from the list of available storage components. Similarly, when list components become free, they are returned (relinked) to the list of available storage components. As a result, the same storage can be used by many different lists during the course of program execution. Sharing storage in this manner reduces the amount of storage that might lie dormant within a list in anticipation of maximum storage requirements for the list.

## **SUMMARY**

1. Chapter 2 shows how to organize and process simple data lists in which each component contains a single data character.
2. Representative subroutines and functions are presented for the following types of operations:
  - a. Creating a list of available storage components
  - b. Manipulating the elements of list components
  - c. Manipulating list components
  - d. Manipulating sublists and lists
  - e. Manipulating lists recursively
3. Primitive subroutines and functions are developed first and are used in turn to create higher level procedures.

## Chapter 3: Complex Data Lists

This chapter shows how the simple list organization of Chapter 2 may be extended to obtain more general types of data lists. These extensions fall into three categories, which involve:

1. Using other types of data in list components besides single characters
2. Storing descriptive information about a list in the head of the list
3. Using additional pointer elements to obtain alternative orderings of list components besides a simple linear ordering

The types of lists produced by these extensions are referred to collectively as complex data lists to distinguish them from the simple list organization presented in Chapter 2. No attempt is made, however, at developing a collection of subroutines and functions that organize and process complex lists. The intent of this chapter is to indicate how the techniques of the previous chapter may be applied to more general list organizations.

### MORE GENERAL DATA IN LIST COMPONENTS

An elementary extension that can be made to simple list organization involves replacing the single-character elements of list components with more general types of data. As Figure 3.1 illustrates, data of any type and precision may appear in list components. Even arrays and structures are permitted. List-processing techniques, as a result, can be applied to numeric as well as nonnumeric data, and to data collections such as tables, records, and reports.

Many of the procedures developed in Chapter 2 still apply to lists that contain more general types of data. Figure 3.2, for example, shows how a list of available storage components may be created for lists that contain arrays of structures. Except for differences in the declarations of the list components, this procedure is identical to the corresponding procedure in Figure 2.1B.

It is also possible to use the compile-time facilities of PL/I to simplify the creation of list-processing procedures for new types of data in list components. Figure 3.3 shows how a general version of the AREA\_OPEN procedure may be created for different list components. The procedure contains two compile-time statements. The first statement (%DECLARE) defines the identifier LIST\_COMPONENT to be a compile-time character-string variable. The second statement is a compile-time assignment statement that assigns a character-string value to LIST\_COMPONENT.

These two statements cause the PL/I compiler to modify the text of AREA\_OPEN before its machine-language equivalent is generated. Each appearance of the identifier LIST\_COMPONENT in AREA\_OPEN is replaced with the value of the identifier. Since the value of LIST\_COMPONENT in this example is identical to the component declaration used throughout Chapter 2, Figure 3.3 is equivalent to the AREA\_OPEN procedure in Figure 2.1B.

Assigning a different component declaration as the value of LIST\_COMPONENT would produce a corresponding change in the AREA\_OPEN procedure and avoid the need for separate copies of the subroutine.

Similar use of compile-time statements can be applied to other list-processing procedures, but some function procedures may require extensive modification or complete replacement by equivalent subroutine procedures. For example, a function procedure cannot return the value of a data item that is not an element item. Consequently, a list-processing function such as GET\_ND, which gets the data item in the nth component of a data list, cannot return the specified data item when it is either an array or a structure. Retrieval of the array or structure would have to be made with a subroutine that uses a parameter to return the desired item.

Other compile-time statements can be used to choose among alternative versions of procedures stored in a list-processing library.

An introductory presentation of the compile-time facilities and their applications appear in the IBM publication *An Introduction to the Compile-Time Facilities of PL/I* (SC20-1689).

### DESCRIPTIVE DATA IN LIST HEADS

A further generalization of list organization involves the head of a list, which so far has been restricted to a pointer variable that specifies the address of the first list component. The head of a list can be enlarged, however, so that it contains descriptive information about the list besides the location of its first component.

The size of a list, for example, need not be computed each time it is requested. The size can be stored in the head of the list, where it is readily obtained by direct reference. With this convention, procedures that insert or delete list components would automatically adjust the size value in the head.


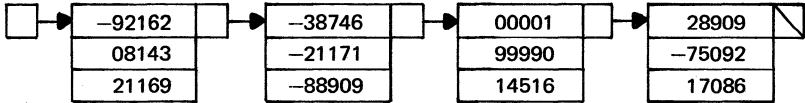
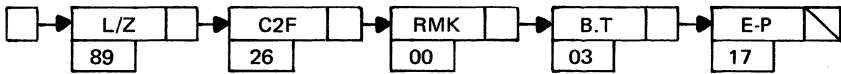
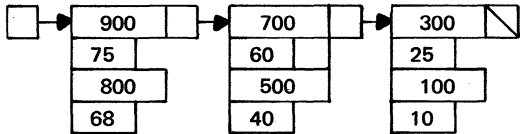
Component Declaration	Example of Data List
1 COMPONENT BASED(P), 2 DATA FIXED DECIMAL(5), 2 POINTER POINTER	L1:  <p style="text-align: center;">Fixed-point decimal items in a data list</p>
1 COMPONENT BASED(P), 2 DATA(3) FIXED DECIMAL(5), 2 POINTER POINTER	L2:  <p style="text-align: center;">Arrays in a data list</p>
1 COMPONENT BASED(P), 2 DATA, 3 PART# PICTURE 'AXA', 3 QUANTITY FIXED DECIMAL(2), 2 POINTER POINTER	L3:  <p style="text-align: center;">Structures in a data list</p>
1 COMPONENT BASED(P), 2 DATA(2), 3 VOLUME FIXED DECIMAL(3), 3 COST FIXED DECIMAL(2), 2 POINTER POINTER	L4:  <p style="text-align: center;">Arrays of structures in a data list</p>

Figure 3.1. Examples of data lists with different types of components



```

AREA_OPEN2:
  PROCEDURE(AREA2,LIST);
  DECLARE
    AREA2 AREA(*),
    (LIST, T) POINTER,
    1 COMPONENT2 BASED(P),
    2 DATA CHARACTER(1),
    2 VOLUME FIXED DECIMAL(3),
    2 COST FIXED DECIMAL(2),
    2 POINTER POINTER;
    ON AREA
  BEGIN;
    IF
      P->NULL
    THEN
      P->POINTER = NULL;
      GO TO
        END_AREA_OPEN2;
  END;
  P = NULL;
  ALLOCATE COMPONENT2 IN(AREA2)SET(P);
  LIST = P;
L:
  T = P;
  ALLOCATE COMPONENT2 IN(AREA2)SET(P);
  T->POINTER = P;
  GO TO
    L;
END_AREA_OPEN2:
END
  AREA_OPEN2;

```

Figure 3.2 Linking components

```

%DECLARE LIST_COMPONENT CHARACTER;
%LIST_COMPONENT = '1 COMPONENT
  BASED(P),
  2 DATA CHARACTER(1),
  2 POINTER POINTER';
AREA_OPEN3:
  PROCEDURE(AREA3,LIST);
  DECLARE
    AREA3 AREA(*),
    (LIST, T) POINTER,
    LIST_COMPONENT;
    ON AREA
  BEGIN;
    IF
      P->NULL
    THEN
      P->POINTER = NULL;
      GO TO
        END_AREA_OPEN3;
  END;
  P = NULL;
  ALLOCATE COMPONENT IN(AREA3) SET(P);
  LIST = P;
L:
  T = P;
  ALLOCATE COMPONENT IN(AREA3) SET(P);
  T->POINTER = P;
  GO TO
    L;
END_AREA_OPEN3:
END
  AREA_OPEN3;

```

Figure 3.3. Using compile-time statements to specify the structure of list components

A possible organization for this type of list head appears in the structure declaration:

```

1 LIST,
2 SIZE FIXED DECIMAL(5),
2 BODY POINTER

```

The identifier LIST serves as the name of the list, and the pointer BODY specifies the address of the first component in the body of the list. The value of SIZE represents the number of components in the list. For an empty list, BODY is null, and SIZE has a zero value.

The DELETE\_ND subroutine in Figure 3.4A provides an example of a procedure that processes data lists with this type of head. The subroutine is similar to the procedure given in Figure 2.12B, except that the head of the list being processed is a data structure and not a pointer element. When a data item is deleted, the size value in the list head is decreased by one. Similarly, because the deleted component is inserted into the list of available storage components, AVAIL, the size value in the head of AVAIL is increased by one. Figure 3.4B contains examples of references to DELETE\_ND.

Observe that DELETE\_ND uses two other list-processing procedures: ADDRESS\_N2 and SET\_POINTER. ADDRESS\_N2 (Figure 3.4C) resembles the ADDRESS\_N procedure in Chapter 2. The two versions cannot be the same, however, because they process lists with different types of heads. The same SET\_POINTER routine is used, since it is not concerned with list heads and assumes similar organizations for list components.

The organization of a list head can be as complicated as desired so that a wide variety of information can be stored in the head, such as:

1. Maximum size achieved by the list
2. Number of references made to the list
3. Name of the list (for output identification)
4. Who has access to the list
5. When the list was last processed

In many respects, an expanded list head resembles the label record used for identification and protection purposes at the front of many data files. The one essential item that the list head must contain, however, is the address of the first list component.

```

DELETE_ND2:
PROCEDURE(LIST_HEAD,N);
DECLARE
  N FIXED DECIMAL(5),
  (ADDRESS1, ADDRESS2, ADDRESS3)
  POINTER,
  1 LIST_HEAD,
  2 SIZE FIXED DECIMAL(5),
  2 BODY POINTER,
  1 AVAIL_HEAD EXTERNAL,
  2 SIZE FIXED DECIMAL(5),
  2 BODY POINTER;
/* IF LIST_HEAD IS EMPTY OR N IS
LESS THAN 1, THEN RETURN. */
IF
(LIST_HEAD.SIZE = 0) | (N<1)
THEN
RETURN;
/* DELETE FIRST COMPONENT WHEN
N = 1. */
IF
N = 1
THEN
DO;
ADDRESS2 = LIST_HEAD.BODY;
LIST_HEAD.BODY = ADDRESS_N2
(LIST_HEAD,2);
GO TO
L;
END;
/* OBTAIN ADDRESS OF N-TH
COMPONENT. */
ADDRESS2 = ADDRESS_N2(LIST_HEAD,N);
IF
ADDRESS2 = NULL
THEN
RETURN;
ADDRESS1 = ADDRESS_N2
(LIST_HEAD, N - 1);
ADDRESS3 = ADDRESS_N2
(LIST_HEAD, N + 1);
/* DELETE N-TH COMPONENT AND
DECREASE LIST_HEAD.SIZE BY 1. */
CALL SET_POINTER(ADDRESS1,
ADDRESS3);
LIST_HEAD.SIZE = LIST_HEAD.SIZE - 1;
/* INSERT DELETED COMPONENT INTO
LIST OF AVAILABLE STORAGE
COMPONENTS AND INCREASE
AVAIL_HEAD_SIZE BY 1. */
L:
ADDRESS1 = AVAIL_HEAD.BODY;
AVAIL_HEAD.BODY = ADDRESS2;
CALL SET_POINTER(AVAIL_HEAD.BODY,
ADDRESS1);
AVAIL_HEAD.SIZE = AVAIL_HEAD.SIZE+1;
END
DELETE_ND2;

```

Figure 3.4A. Deleting the data item in the nth position of a data list, the size of which is stored in the list head

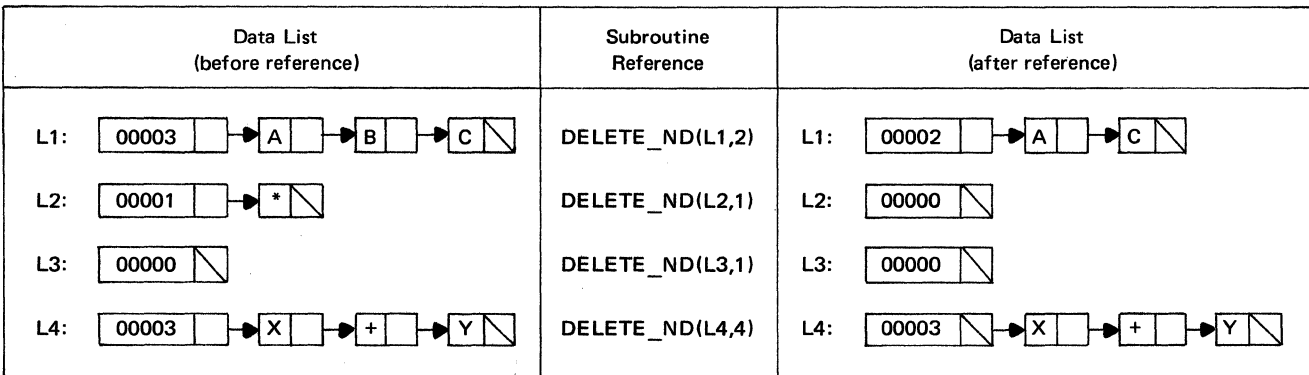


Figure 3.4B. Examples showing how size values in list heads are changed when data items are deleted

```

ADDRESS_N2:PROCEDURE(LIST_HEAD, N)
  RETURNS (POINTER);
DECLARE
  1 LIST_HEAD,
  2 SIZE FIXED DECIMAL(5),
  2 BODY POINTER,
  N FIXED DECIMAL(5),
  1 COMPONENT BASED(ADDRESS),
  2 DATA CHARACTER(1),
  2 POINTER POINTER;
IF
(LIST_HEAD.SIZE = 0) | (N < 1)
THEN
  RETURN (NULL);
  ADDRESS = LIST_HEAD.BODY;
DO
  I = 1 BY 1;
IF
  (ADDRESS->POINTER=NULL)&(I≠N)
  THEN RETURN (NULL);
IF
  I = N
  THEN RETURN(ADDRESS);
  ADDRESS = ADDRESS->POINTER;
END;
END ADDRESS_N2;

```

Figure 3.4C. Obtaining the address of the nth component in a list, as used by DELETE\_ND2

**ALTERNATIVE METHODS FOR LINKING LIST COMPONENTS**

Using more than one pointer element in each list component permits the components to possess a more complex ordering than the simple linear ordering developed so far. The following discussions show how additional pointer elements may be used to create three general categories of lists with complex orderings:

1. Two-way lists
2. Circular lists
3. Multidirectional lists

**Two-Way Lists**

The components of a two-way list are linked in both a forward and a backward direction, as illustrated by Figure 3.5A. Each component contains two pointer elements: one for forward linking and the other for backward linking. The forward pointer contains the address of the next component in the list; the backward pointer contains the address of the previous component. The forward pointer of the last component and the backward pointer of the first component both contain null address values.

Although the pointers in the diagram of Figure 3.5A point to other pointer elements, it should be understood that the pointers always contain the addresses of entire list components and not the addresses of elements within the components.

This point is further illustrated by the subroutine procedure FORM\_TWO\_WAY in Figure 3.5B, which forms the two-way list shown in Figure 3.5A. As a simplification,

the procedure uses a storage area having external scope and assumes that the area contains sufficient storage for another list.

With appropriate modifications, the techniques developed in Chapter 2 can also be applied to this type of list.

The major advantage of a two-way list is that it permits scanning operations to be performed with equal efficiency in both a forward and a backward direction. In many list-processing applications, it is necessary to stop at a certain position in a list and to process earlier portions of the list. To reach the earlier portions in a one-way list, a procedure must scan through the list from the beginning. Such scanning increases program running time, particularly when the portions being sought are not at the front of the list. In a two-way list, the desired position can be reached by backing through the list.

For example, in a text-editing program that uses list-processing techniques, it might be necessary to delete the first sentence of all paragraphs that end with a specified word. Such a program would scan to the end of each paragraph before determining whether the first sentence was to be deleted. With the text arranged in a two-way list, it would generally be more efficient to move backwards from the end of a paragraph than to come through the entire text to the beginning of the paragraph.

Similar considerations apply to arithmetic expressions that contain nested subexpressions. Scanning such expressions by list-processing techniques generally requires frequent repositioning in forward and backward directions as subexpressions are processed.

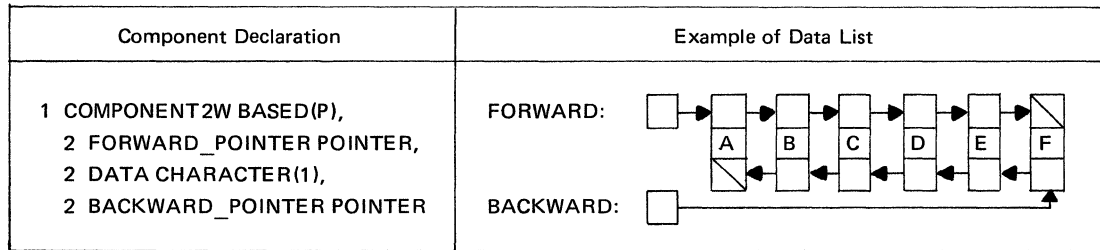


Figure 3.5A. Example of a two-way data list

```

FORM_TWO_WAY:
  PROCEDURE(FORWARD, BACKWARD);
  DECLARE
    AREA2W AREA EXTERNAL,
    (FORWARD, BACKWARD,T) POINTER,
    TABLE(6) CHARACTER(1)
    INITIAL('A', 'B', 'C', 'D', 'E',
    'F'),
    1 COMPONENT2W BASED(P),
    2 FORWARD_POINTER POINTER,
    2 DATA CHARACTER(1),
    2 BACKWARD_POINTER POINTER;
    /* ALLOCATE FIRST LIST COMPONENT,
    AND LINK IT TO LIST HEAD. */
    ALLOCATE COMPONENT2W
    IN (AREA2W) SET(P);
    FORWARD = P;
    P->DATA = TABLE(1);
    P->BACKWARD_POINTER = NULL;
    /* ALLOCATE REMAINING FIVE
    COMPONENTS, AND LINK ALL COMPONENTS
    IN FORWARD AND BACKWARD DIRECTION. */
  DO
    I = 2 TO 6;
    T = P;
    ALLOCATE COMPONENT2W
    IN (AREA2W) SET(P);
    T->FORWARD_POINTER = P;
    P->DATA = TABLE(I);
    P->BACKWARD_POINTER = T;
  END;
  P->FORWARD_POINTER = NULL;
  BACKWARD = P;
END
  FORM_TWO_WAY;

```

Figure 3.5B. Forming a two-way list

### Circular Lists

A circular list is obtained by linking the last component of a one-way list to the first component. Similar linking, when applied to a two-way list, produces a two-way circular list, as shown in Figure 3.6A.

Subroutine procedure FORM\_TWO\_WAY\_CIRCULAR in Figure 3.6B forms the two-way circular list illustrated in Figure 3.6A. The subroutine uses an external storage area, which is assumed to contain enough free storage for another list. When the list is formed, it is linked to the list head specified in the invocation of the subroutine.

Circular lists prove useful in applications that perform repeated processing of list items. In a time-sharing system, for example, each component of a list may contain control information for a remote terminal that is requesting computer time. Since each terminal receives service for brief periods so that no terminal is forced to remain idle very long, repeated servicing of the terminals is required until each has completed its task. When a terminal becomes inactive, its corresponding component is deleted from the circular list. Reactivation of a terminal reinserts the associated component into the list.

Graphic display devices provide another application of circular lists. Any data that is displayed on a graphic console must be transmitted continually; otherwise, it will fade from the display screen. Placing the display data in a circular list provides a convenient way of repeatedly retrieving and displaying the data until an interrupt condition terminates the display.

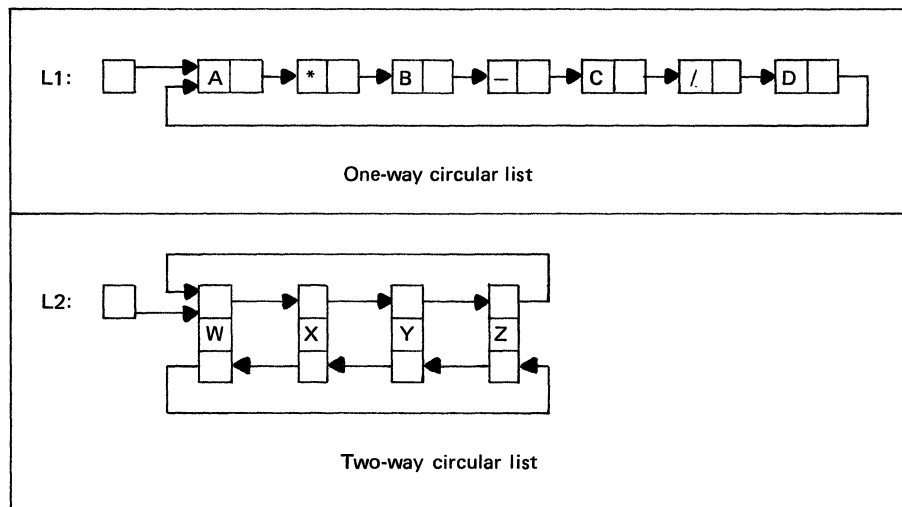


Figure 3.6A. Examples of circular lists

```

FORM_TWO_WAY_CIRCULAR:
  PROCEDURE(L2);
  DECLARE
    AREAC AREA EXTERNAL,
    (L2, T) POINTER,
    TABLE(4) CHARACTER(1)
    INITIAL('W', 'X', 'Y', 'Z'),
    1 COMPONENTC BASED(P),
    2 FORWARD_POINTER POINTER,
    2 DATA CHARACTER(1),
    2 BACKWARD_POINTER POINTER;
    /* ALLOCATE FIRST LIST COMPONENT,
    AND LINK IT TO LIST HEAD. */
    ALLOCATE COMPONENTC IN(AREAC)SET(P);
    L2 = P;
    P->DATA = TABLE(1);
    /* ALLOCATE REMAINING THREE
    COMPONENTS, AND LINK ALL COMPONENTS
    IN FORWARD AND BACKWARD
    DIRECTION. */
  DO
    I = 2 TO 4;
    T = P;
    ALLOCATE COMPONENTC IN(AREAC)SET(P);
    T->FORWARD_POINTER = P;
    P->DATA = TABLE(I);
    P->BACKWARD_POINTER = T;
  END;
  /* COMPLETE FORWARD AND BACKWARD
  CIRCULAR LINKS. */
  P->FORWARD_POINTER = L2;
  L2->BACKWARD_POINTER = P;
END
FORM TWO WAY CIRCULAR;

```

Figure 3.6B. Forming a two-way circular list

Circular lists also permit more accurate modeling of data organizations that contain inherent circularities. Later discussions present illustrations of such organizations in chemistry, geometry, and the game of chess.

Note that the circular lists presented in Figure 3.6A do not include the list head as part of the circular linkage. As Figure 3.7A shows, however, the list head can be included in the circular linkage. Each component contains a second pointer element that points to the list head. This organization provides each component with quick access to the list head.

An application for this type of list involves the light pen on a graphic display device. The light pen can be used to refer to a portion of the data on the display screen. If the display data is stored within the computer in list form, the effect of the light pen is equivalent to selecting a list component without first determining what list contains the component. Storing the address of the list head within each component identifies the containing list immediately.

Subroutine procedure FORM\_CIRCULAR\_TO\_HEAD in Figure 3.7B shows how to create the list shown in Figure 3.7A. Observe that this list is actually a two-way circular list. The forward linkage does not include the list head, but the backward linkage does.

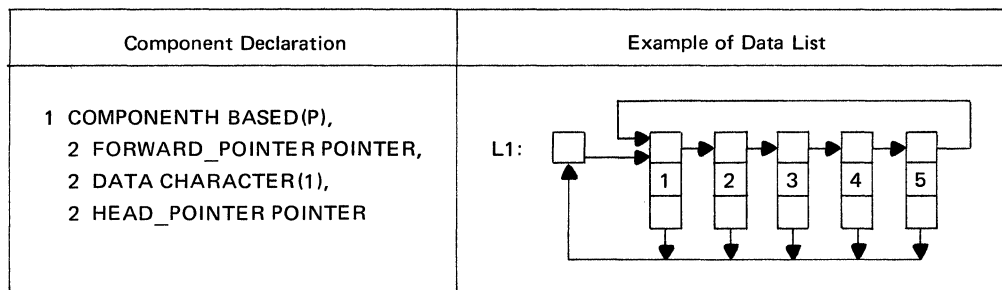


Figure 3.7A. Circular list with additional pointer elements that point to the list head

```

FORM_CIRCULAR_TO_HEAD:
  PROCEDURE(L1);
  DECLARE
    AREAH AREA EXTERNAL,
    (L1, ADDRESS_OF_HEAD, T) POINTER,
    TABLE(5) CHARACTER(1)
    INITIAL('1', '2', '3', '4', '5'),
    1 COMPONENTH BASED(P),
    2 FORWARD_POINTER POINTER,
    2 DATA CHARACTER(1),
    2 HEAD_POINTER POINTER;
    /* OBTAIN ADDRESS OF LIST HEAD. */
    ADDRESS_OF_HEAD = ADDR(L1);
    /* ALLOCATE AND LINK FIRST LIST
    COMPONENT. */
    ALLOCATE COMPONENTH IN(AREAH)SET(P);
    L1 = P;
    P->DATA = TABLE(1);
    P->HEAD_POINTER = ADDRESS_OF_HEAD;
    /* ALLOCATE AND LINK REMAINING FOUR
    COMPONENTS. */
  DO
    I = 2 TO 5;
    T = P;
    ALLOCATE COMPONENTH IN(AREAH)SET(P);
    T->FORWARD_POINTER = P;
    P->DATA = TABLE(I);
    P->HEAD_POINTER = ADDRESS_OF_HEAD;
  END;
  /* COMPLETE FORWARD LINK. */
  P->FORWARD_POINTER = L1;
END
FORM_CIRCULAR_TO_HEAD;

```

Figure 3.7B. Forming a circular list with additional pointer elements that point to the list head

## Multidirectional Lists

When more than one pointer element is permitted in each list component, it is possible for each component to possess more than one immediate predecessor component and more than one immediate successor component. The multidirectional organization of data lists with such components serves as a convenient tool for modeling complex systems that possess discrete arrangements of their parts. Examples of such systems occur in many fields:

- Electrical and communication networks
- Industrial-process scheduling
- PERT and critical path analyses
- Economic and social structures
- Military tactics and logistics
- Switching circuits
- Chemical structures
- Optimization of transportation routes and network flows
- Games and puzzles

The general subject of *graph theory* provides a mathematical foundation for the study of these systems. This theory uses circles (or points) interconnected by lines to represent the essential organization of a system. In a highway network, for example, the lines of a graph may represent roads, and circles may denote the points where the roads intersect.

When it is possible to trace a path through a graph and return to a previously encountered circle, the graph is said to contain a circuit. A graph without circuits is called a tree. Examples of both types of graphs appear in Figure 3.8.

A special case of a tree graph occurs when each circle has at most two immediate successor circles. Such a graph is called a binary tree. Figure 3.9A shows how a binary tree may be used to represent a mathematical expression in parentheses -- free form. The figure also contains a representation of the binary tree as a data list. Each component in the list contains a left pointer and a right pointer, which provide the two possible branches from the component. This type of list forms a useful feature in many compilation techniques.

The subroutine procedure `FORM_BINARY_TREE` in Figure 3.9B shows how to construct the data list presented in Figure 3.9A. In common with the three remaining illustrations of specialized data lists in this chapter, processing of `FORM_BINARY_TREE` has been limited to compilation and execution.

## ADVANCED APPLICATIONS OF COMPLEX DATA LISTS

As indicated in the discussion of multidimensional lists, the range of possible applications for complex data lists is extensive and easily exceeds the scope of this text. The following discussions, therefore, deal with only three application areas: chemistry, geometry, and the game of chess. Since each of these areas is in turn quite broad, no attempt is made to develop complete applications. The intent of the discussions is to outline some of the ways complex list organizations may be used in advanced applications.

### List Representation of Structural Formulas in Chemistry

Chemistry deals with the structure of matter and its changes and uses two types of formulas to describe the atomic structure of substances: empirical formulas and structural formulas. Examples of empirical formulas for common substances are:

$H_2O$	--	Water
$CO_2$	--	Carbon Dioxide
$NaCl$	--	Sodium Chloride (salt)
$H_2SO_4$	--	Sulfuric acid

Single or double letters specify atoms: H (hydrogen), O (oxygen), C (carbon), Na (sodium), Cl (chlorine), and S (sulfur). Subscripts indicate the number of atoms in a molecule (the smallest amount) of a substance. A single atom employs no subscript. Therefore, the formula  $H_2O$  states that a molecule of water contains two hydrogen atoms ( $H_2$ ) and one oxygen atom (O).

The formula  $H_2O$  also indicates that an oxygen atom has two possible "places of attachment" with other atoms and that a hydrogen atom has only one place of attachment. In structural formulas these places of attachment are made visible by lines, called bonds, which are attached to the letter symbol for each atom.

The top row of Figure 3.10 contains the structural formulas for hydrogen (one bond), oxygen (two bonds), nitrogen (three bonds), and carbon (four bonds). Other atoms have larger numbers of bonds; the maximum is seven.

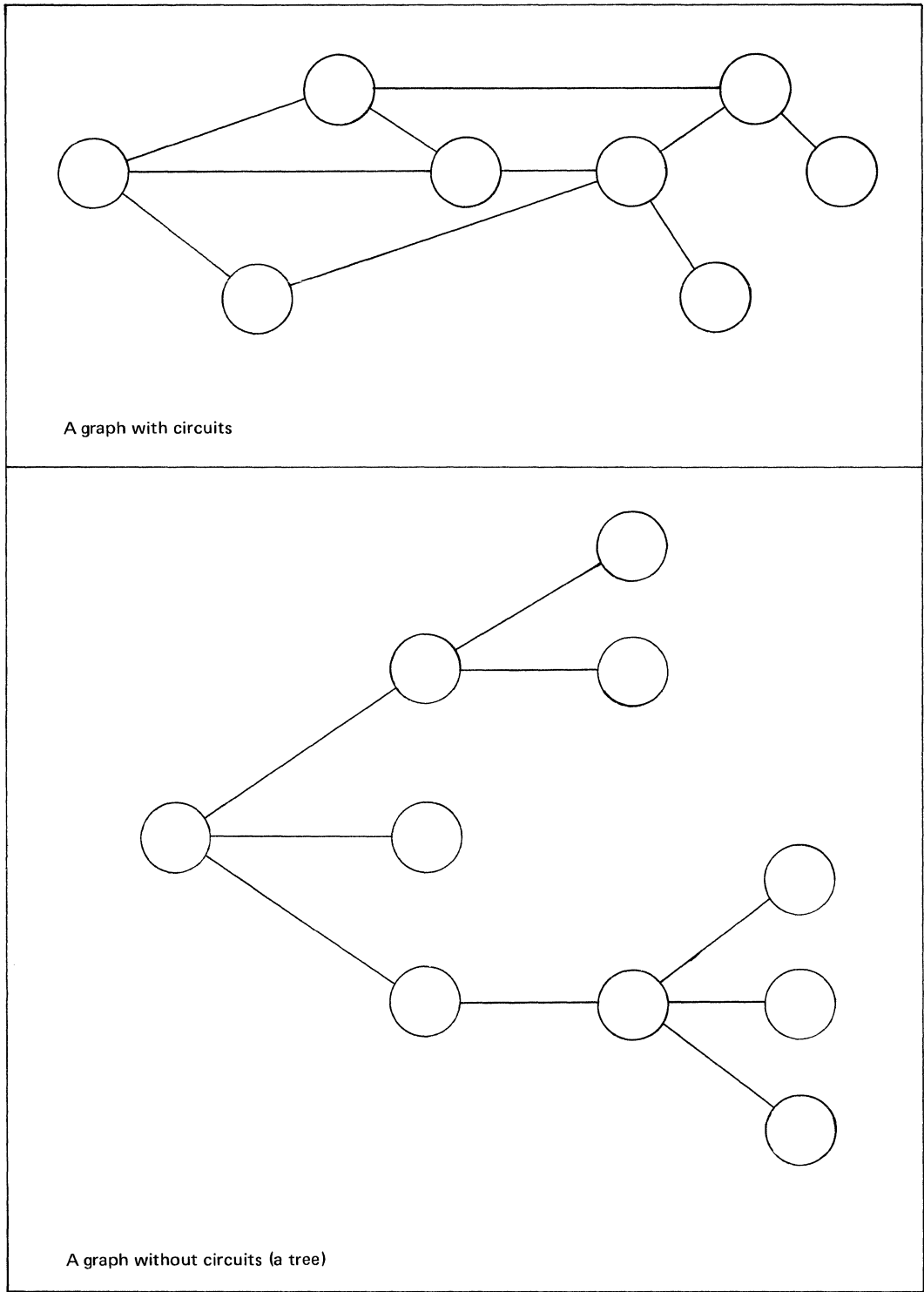
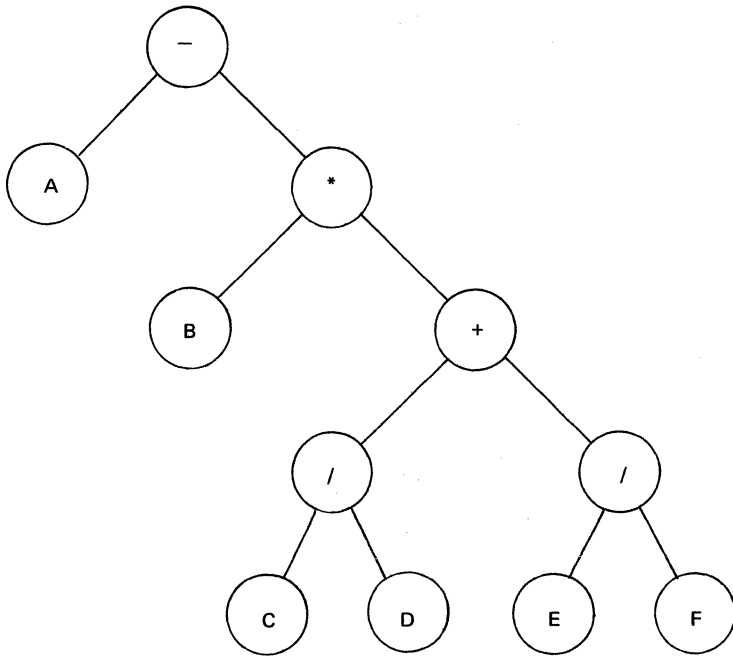
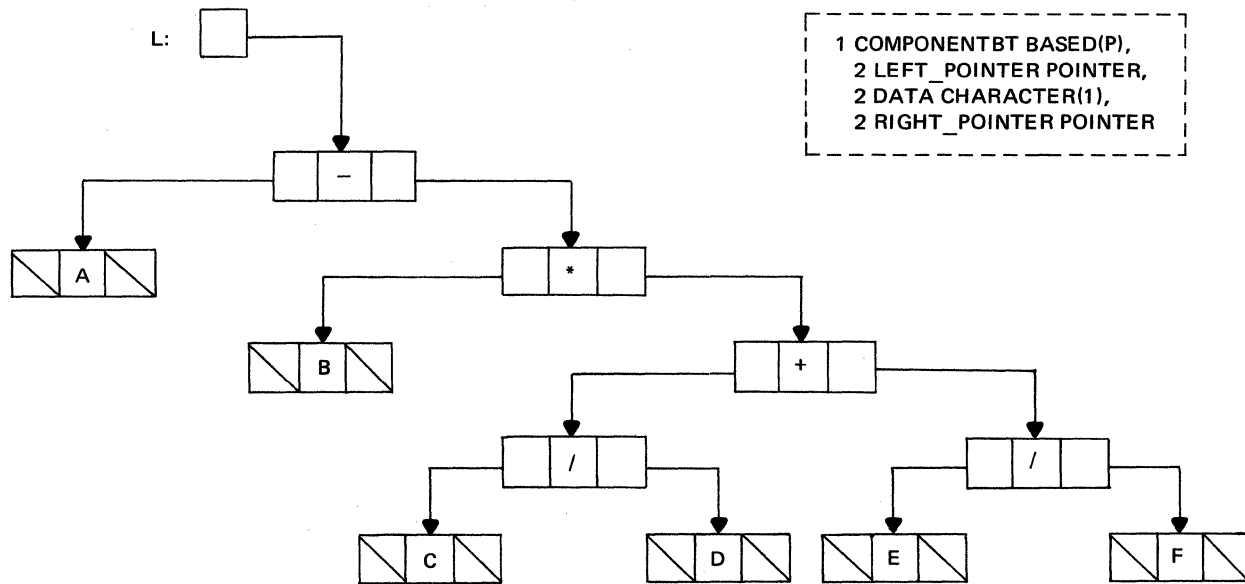


Figure 3.8. Examples of graphs

Expression:  $A - (B * ((C/D) + (E/F)))$



Expression as a binary tree



Expression as a data list

Figure 3.9A. Representation of a binary tree as a data list



```

FORM_BINARY_TREE:
  PROCEDURE(L);
  DECLARE
    AREA6 AREA EXTERNAL,
    L POINTER,
    (P, Q, R, S) POINTER,
    1 COMPONENTBT BASED(P),
    2 LEFT_POINTER POINTER,
    2 DATA CHARACTER(1),
    2 RIGHT_POINTER POINTER;
    /* FORM E/F. */
    ALLOCATE COMPONENTBT IN(AREA6)SET(P);
    ALLOCATE COMPONENTBT IN(AREA6)SET(Q);
    P->LEFT_POINTER = NULL;
    P->DATA = 'E';
    P->RIGHT_POINTER = NULL;
    Q->LEFT_POINTER = NULL;
    Q->DATA = 'F';
    Q->RIGHT_POINTER = NULL;
    ALLOCATE COMPONENTBT IN(AREA6)SET(R);
    R->LEFT_POINTER = P;
    R->DATA = '/';
    R->RIGHT_POINTER = Q;
    /* FORM C/D. */
    ALLOCATE COMPONENTBT IN(AREA6)SET(P);
    P->LEFT_POINTER = NULL;
    P->DATA = 'C';
    P->RIGHT_POINTER = NULL;
    ALLOCATE COMPONENTBT IN(AREA6)SET(Q);
    Q->LEFT_POINTER = NULL;
    Q->DATA = 'D';
    Q->RIGHT_POINTER = NULL;
    ALLOCATE COMPONENTBT IN(AREA6)SET(S);
    S->LEFT_POINTER = P;
    S->DATA = '/';
    S->RIGHT_POINTER = Q;
    /* FORM (C/D) + (E/F). */
    ALLOCATE COMPONENTBT IN(AREA6)SET(P);
    P->LEFT_POINTER = S;
    P->DATA = '+';
    P->RIGHT_POINTER = R;
    /* FORM B*((C/D) + (E/F)). */
    ALLOCATE COMPONENTBT IN(AREA6)SET(Q);
    Q->LEFT_POINTER = NULL;
    Q->DATA = 'B';
    Q->RIGHT_POINTER = NULL;
    ALLOCATE COMPONENTBT IN(AREA6)SET(R);
    R->LEFT_POINTER = Q;
    R->DATA = '*';
    R->RIGHT_POINTER = P;
    /* FORM A-(B*((C/D) + (E/F))). */
    ALLOCATE COMPONENTBT IN(AREA6)SET(Q);
    Q->LEFT_POINTER = NULL;
    Q->DATA = 'A';
    Q->RIGHT_POINTER = NULL;
    ALLOCATE COMPONENTBT IN(AREA6)SET(P);
    P->LEFT_POINTER = Q;
    P->DATA = '-';
    P->RIGHT_POINTER = R;
    L = P;
  END
FORM_BINARY_TREE;

```

Figure 3.9B. Forming a binary tree as a data list

$\text{H} -$ Hydrogen atom	$- \text{O} -$ Oxygen atom	$- \text{N} -$ Nitrogen atom	$\begin{array}{c}   \\ - \text{C} - \\   \end{array}$ Carbon atom
$\text{O} = \text{O}$ $\text{O}_2$ : Oxygen molecule	$\text{O} = \text{C} = \text{O}$ $\text{CO}_2$ : Carbon dioxide	$\begin{array}{c} \text{H} - \text{N} - \text{H} \\   \\ \text{H} \end{array}$ $\text{NH}_3$ : Ammonia	$\begin{array}{c} \text{H} \\   \\ \text{H} - \text{C} - \text{H} \\   \\ \text{H} \end{array}$ $\text{CH}_4$ : Methane
$\begin{array}{cccccccc} & \text{H} & \text{H} & \text{H} & \text{H} & \text{H} & \text{H} & \text{H} \\ &   &   &   &   &   &   &   \\ \text{H} & - \text{C} & - \text{C} & - \text{C} & - \text{C} & - \text{C} & - \text{C} & - \text{C} - \text{H} \\ &   &   &   &   &   &   &   \\ & \text{H} & \text{H} & \text{H} & \text{H} & \text{H} & \text{H} & \text{H} \end{array}$ $\text{C}_8\text{H}_{18}$ : Normal octane			
$\begin{array}{ccccccc} & & \text{H} & \text{H} & \text{H} & & \\ & &   &   &   & & \\ & & \text{C} & & \text{C} & & \\ \text{H} & \diagdown &   &   &   & \diagup & \text{H} \\   & &   &   &   & &   \\ \text{H} & & \text{C} & - \text{C} & - \text{C} & - \text{C} & \text{H} \\ & &   &   &   & & \\ & & \text{C} & & \text{C} & & \\ & &   &   &   & & \\ & & \text{H} & \text{H} & \text{H} & & \end{array}$ $\text{C}_8\text{H}_{18}$ : Isooctane		$\begin{array}{c} \text{H} \\   \\ \text{C} \\ / \quad \backslash \\ \text{H} - \text{C} \quad \text{C} - \text{H} \\    \quad \backslash \quad / \\ \text{H} - \text{C} \quad \text{C} - \text{H} \\ \backslash \quad / \\ \text{C} \\   \\ \text{H} \end{array}$ $\text{C}_6\text{H}_6$ : Benzene ring	

Figure 3.10. Examples of structural formulas in chemistry

The structural formulas in the remaining rows of Figure 3.10 show how atoms are joined by their bonds to form a variety of substances: molecular oxygen (in the air we breathe), carbon dioxide (formed during respiration), ammonia (the gas contained in household cleaners), methane (natural gas used for fuel), normal octane (used in gasoline), isooctane (used in high-octane gasoline), and benzene (a coal-tar derivative used as a solvent).

Structural formulas prove to be of great importance to chemists, because they not only specify the type and number of atoms but also show how the atoms are arranged and interconnected within a molecule of a substance. Although two molecules may contain the same type and number of atoms, the atoms may be arranged differently and, as a result, form different substances. For example, Figure 3.10 shows how 8 carbon and 18 hydrogen atoms may be combined to form two different molecules: normal octane and isooctane. Both molecules possess the same empirical formula,  $C_8H_{18}$ , but are actually different substances.

Data lists provide a convenient way of representing the structural organizations of molecules. Figure 3.11A contains a list representation for the structural formula of water ( $H-O-H$ ). Each list component represents an atom and contains a two-position character string for the letter symbol of the atom and seven pointer elements, which serve as connecting bonds. When an atom has fewer than seven bonds, unused pointers are set to null.

Subroutine procedure `FORM_WATER` in Figure 3.11B shows a way of creating the data list in Figure 3.11A.

Once the structural formula for a molecule is represented as a data list, a variety of procedures can be written to analyze the structure of the molecule, to search for patterns of atoms, and to simulate chemical experiments.

### List Representation of Geometric Figures

The use of data lists to represent the geometric arrangement of atoms in molecules indicates that data lists may also be used to represent the structure of geometric figures in general.

Figure 3.12A shows declarations for the head and components of a data list that represents a pyramid. Besides containing the address of the first list component, the list head includes the name and volume of the pyramid. Each list component represents a vertex in the figure and contains the name and the three coordinates of the vertex. Six pointer elements also appear in each list component: one pointer links the component to its successor component, and the remaining five pointers each link the component to other components that form each of the five possible faces in the pyramid. If a vertex is not associated with a particular face in the pyramid, the corresponding pointer is null.

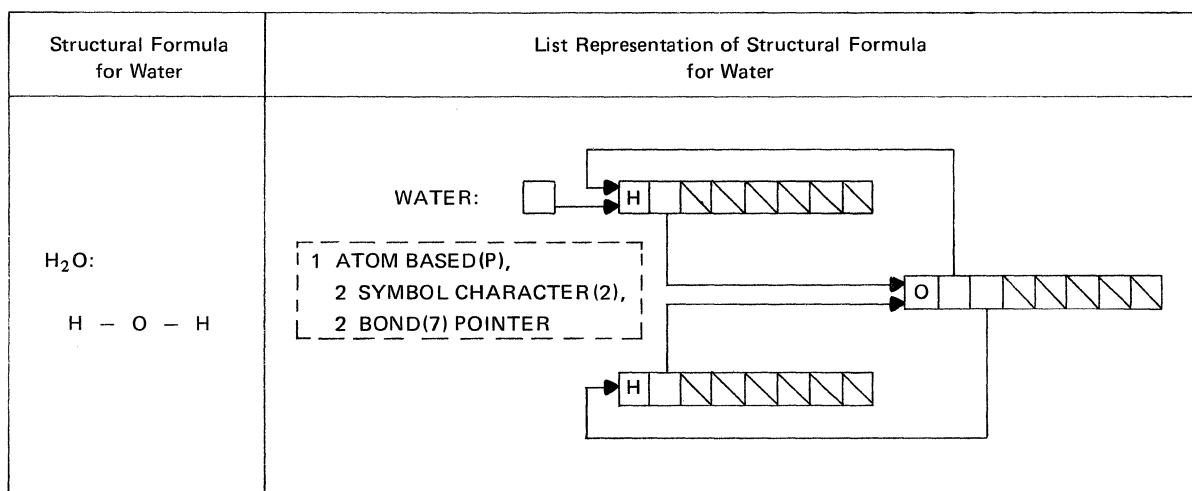


Figure 3.11A. List representation of water

```

FORM_WATER:
  PROCEDURE(WATER);
  DECLARE
    (WATER,P,Q) POINTER,
    AREA7 AREA EXTERNAL,
    1 ATOM BASED(P),
    2 SYMBOL CHARACTER(2),
    2 BOND(7) POINTER;
    /* FORM FIRST HYDROGEN ATOM AND
    LINK IT TO POINTER CALLED WATER. */
    ALLOCATE ATOM IN(AREA7) SET(WATER);
    WATER->SYMBOL = 'H';
    /* FORM OXYGEN ATOM. */
    ALLOCATE ATOM IN(AREA7) SET(P);
    P->SYMBOL = 'O';
    /* LINK HYDROGEN AND OXYGEN ATOMS.*/
    WATER->BOND(1) = P;
    P->BOND(1) = WATER;
    /* FORM SECOND HYDROGEN ATOM. */
    ALLOCATE ATOM IN(AREA7) SET(Q);
    Q->SYMBOL = 'H';
    /* LINK SECOND HYDROGEN ATOM TO
    OXYGEN ATOM. */
    Q->BOND(1) = P;
    P->BOND(2) = Q;
    /* IN EACH ATOM, SET UNUSED BONDS
    TO NULL */
    DO
      I = 2 TO 7; WATER->BOND(I) = NULL;
    END;
    DO
      I = 3 TO 7; P->BOND(I) = NULL;
    END;
    DO
      I = 2 TO 7; Q->BOND(I) = NULL;
    END;
  END;
FORM_WATER;

```

Figure 3.11B. Forming the list representation of water

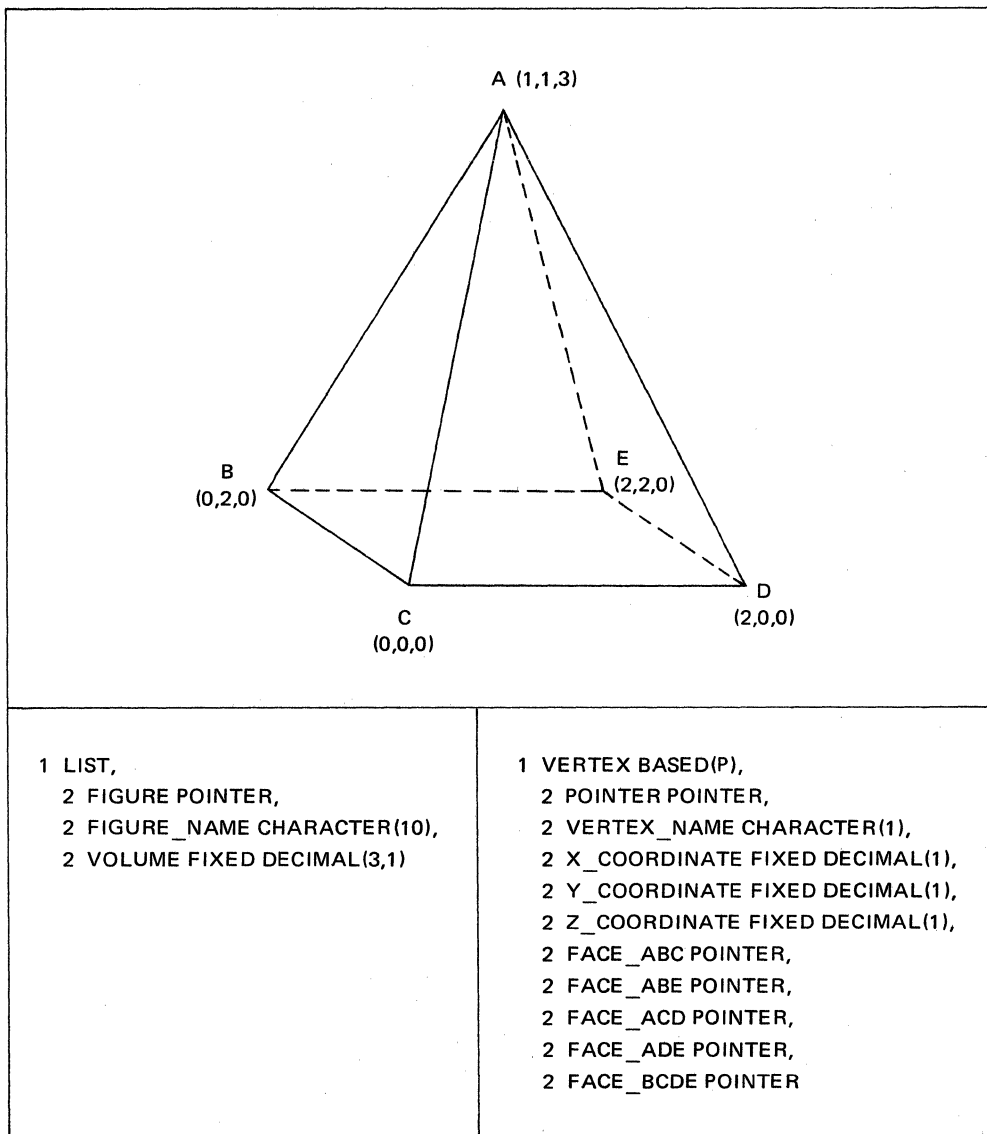


Figure 3.12A. Head and component descriptions for representing a pyramid as a data list

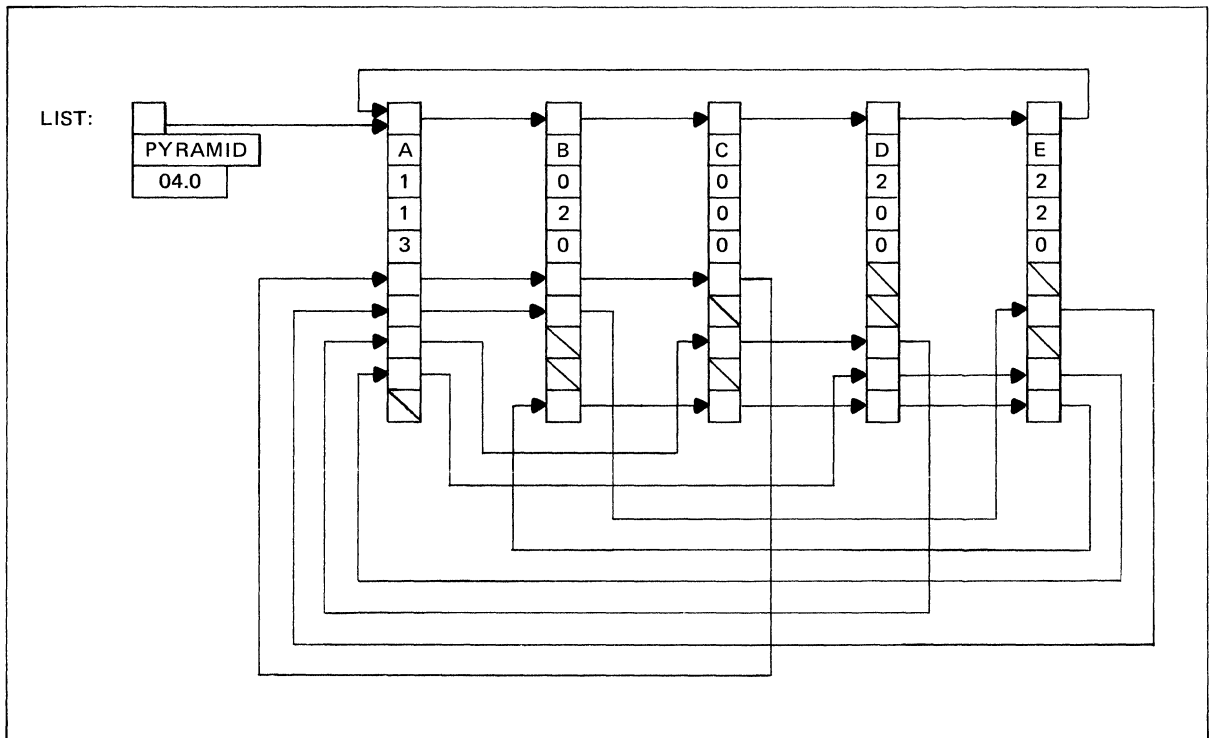


Figure 3.12B. Representation of a pyramid as a data list

Figure 3.12B shows the list representation of the pyramid in Figure 3.12A. Successive components for the entire pyramid are linked to form a one-way circular list. Successive components for each face are also linked in circular fashion. As discussed earlier, circular lists simplify the continual display of data on a graphic device.

Each vertex in the pyramid or in a face of the pyramid is obtained by moving through the associated circular list. By moving through the face pointers in a particular list

component, it is possible to select each face that contains the associated vertex.

Subroutine procedure FORM\_PYRAMID in Figure 3.12C shows one way of constructing the data list in Figure 3.12B.

Additional list-processing procedures can be designed to enlarge, contract, or change the orientation of a geometric figure on a graphic display device and also to create other figures by combining subfigures.

```

FORM_PYRAMID:
  PROCEDURE (LIST, NAME, VOLUME,
    X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,
    X4,Y4,Z4,X5,Y5,Z5);
  DECLARE
    NAME CHARACTER(10),
    VOLUME FIXED DECIMAL(3,1),
    (X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,
    X4,Y4,Z4,X5,Y5,Z5) FIXED DECIMAL(1),
    AREA9 AREA(5000) EXTERNAL,
    (A,B,C,D,E, V(5)) POINTER,
    1 LIST,
    2 FIGURE POINTER,
    2 FIGURE_NAME CHARACTER(10),
    2 VOLUME FIXED DECIMAL(2),
    1 VERTEX BASED(P),
    2 POINTER POINTER,
    2 VERTEX_NAME CHARACTER(1),
    2 X_COORDINATE FIXED DECIMAL(1),
    2 Y_COORDINATE FIXED DECIMAL(1),
    2 Z_COORDINATE FIXED DECIMAL(1),
    2 FACE_ABC POINTER,
    2 FACE_ABE POINTER,
    2 FACE_ACD POINTER,
    2 FACE_ADE POINTER,
    2 FACE_BCDE POINTER;
    /* ALLOCATE STORAGE FOR FIVE
    VERTICES. */
  DO
    I = 1 TO 5;
    ALLOCATE VERTEX IN(AREA9)SET(P);
    V(I) = P;
  END;
  /* INITIALIZE LIST HEAD, AND LINK
  VERTICES. */
  FIGURE_NAME = NAME;
  LIST.VOLUME = VOLUME;
  FIGURE = V(1);
  A = V(1); B = V(2); C = V(3);
  D = V(4); E = V(5);
  A->POINTER = B;
  B->POINTER = C;
  C->POINTER = D;
  D->POINTER = E;
  E->POINTER = A;
  /* ASSIGN NAMES TO VERTICES. */
  A->VERTEX_NAME = 'A';
  B->VERTEX_NAME = 'B';
  C->VERTEX_NAME = 'C';
  D->VERTEX_NAME = 'D';
  E->VERTEX_NAME = 'E';
  /* ASSIGN COORDINATES TO
  VERTICES. */
  A->X_COORDINATE = X1;
  A->Y_COORDINATE = Y1;
  A->Z_COORDINATE = Z1;
  B->X_COORDINATE = X2;
  B->Y_COORDINATE = Y2;
  B->Z_COORDINATE = Z2;
  C->X_COORDINATE = X3;
  C->Y_COORDINATE = Y3;
  C->Z_COORDINATE = Z3;
  D->X_COORDINATE = X4;
  D->Y_COORDINATE = Y4;
  D->Z_COORDINATE = Z4;
  E->X_COORDINATE = X5;
  E->Y_COORDINATE = Y5;
  E->Z_COORDINATE = Z5;
  /* LINK VERTICES TO FORM FIVE
  FACES. */
  A->FACE_ABC = B;
  B->FACE_ABC = C;
  C->FACE_ABC = A;
  A->FACE_ABE = B;
  B->FACE_ABE = E;
  E->FACE_ABE = A;
  A->FACE_ACD = C;
  C->FACE_ACD = D;
  D->FACE_ACD = A;
  A->FACE_ADE = D;
  D->FACE_ADE = E;
  E->FACE_ADE = A;
  B->FACE_BCDE = C;
  C->FACE_BCDE = D;
  D->FACE_BCDE = E;
  E->FACE_BCDE = B;
  /* SET UNUSED POINTERS TO NULL. */
  A->FACE_BCDE,
  B->FACE_ACD,
  B->FACE_ADE,
  C->FACE_ABE,
  C->FACE_ADE,
  D->FACE_ABC,
  D->FACE_ABE,
  E->FACE_ABC,
  E->FACE_ACD = NULL;
  END
  FORM_PYRAMID;

```

Figure 3.12C. Forming the list representation of a pyramid

### List Representation of a Chessboard

Various attempts have been made in the field of artificial intelligence to program a computer so that it can play a game of chess. Although no program has yet been able to master chess, modest success has been achieved by some programs in playing against human opponents, and the game still remains a fertile area for research on such topics as pattern recognition, heuristic methods, and machine learning.

The following discussion shows how a data list can be used to represent a chessboard and how the chessmen can

be arranged on the board in initial position. No attempt is made at developing a program that actually plays chess; such a program lies beyond the scope of this text.

Figure 3.13A contains an illustration of a chessboard and shows the chessmen for white and black arranged in initial position. Two-letter abbreviations represent the chessmen: WP (white pawn), WR (white rook), WN (white knight), WB (white bishop), WQ (white queen), and WK (white king). Substituting B for W provides similar abbreviations for the black chessmen.

Black								
8	BR	BN	BB	BQ	BK	BB	BN	BR
7	BP	BP	BP	BP	BP	BP	BP	BP
6								
5								
4								
3								
2	WP	WP	WP	WP	WP	WP	WP	WP
1	WR	WN	WB	WQ	WK	WB	WN	WR
	a	b	c	d	e	f	g	h
White								

Figure 3.13A. Chessboard in initial position, showing the coordinate system for identifying files and ranks

The coordinate system is used for identifying individual squares on the board. The letters a through h specify files (columns), and the digits 1 through 8 specify ranks (rows). The identifier a1, for example, represents the lower left-hand square, and the identifier h8 represents the upper right-hand square.

Figure 3.13B shows a possible declaration for list components that represent squares on the board. The component contains the name of the square (its coordinate) and the name of the chessman on the square (a blank name indicates an empty square). The component also contains eight pointer elements that represent the eight possible directions to neighboring squares (as illustrated on the right of Figure 3.13B).

Figure 3.13C shows the list representation for the chessboard. This illustration differs from the list representations used earlier. Double-angled lines specify two-way linking of squares, and the component elements for the names of the squares are not shown. These modifications of earlier conventions avoid a cluttered diagram.

The list head (BOARD) points to square A1. Observe also that the list is essentially a circular list with many circuits.

The procedure named CHESS in Figure 3.13D contains the subroutines named BUILD\_BOARD and SET\_MEN. The subroutine procedure BUILD\_BOARD shows how to construct the list in Figure 3.13C. Subroutine procedure SET\_MEN sets the chessmen on the board in initial position. Neither subroutine uses parameters; external variables provide the necessary communication with the subroutines.

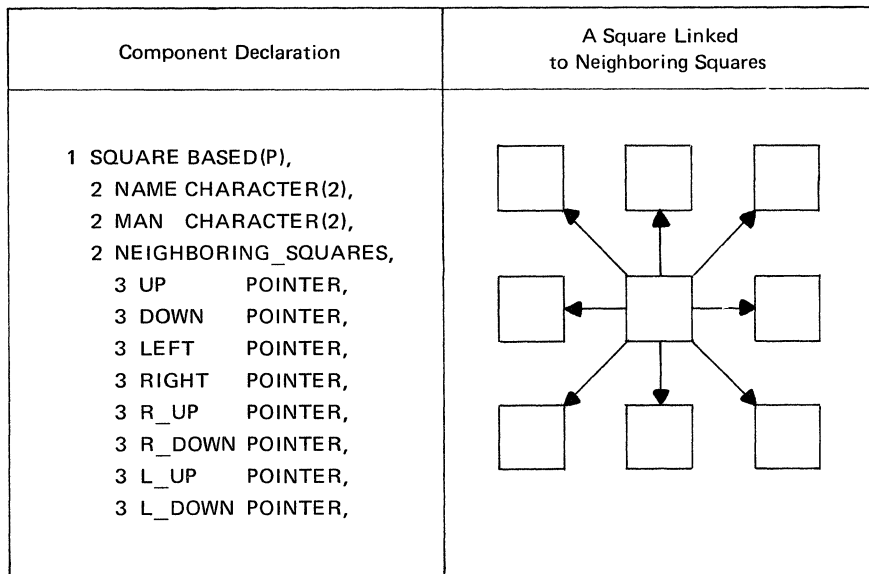


Figure 3.13B. Possible directions from a chessboard square to neighboring squares

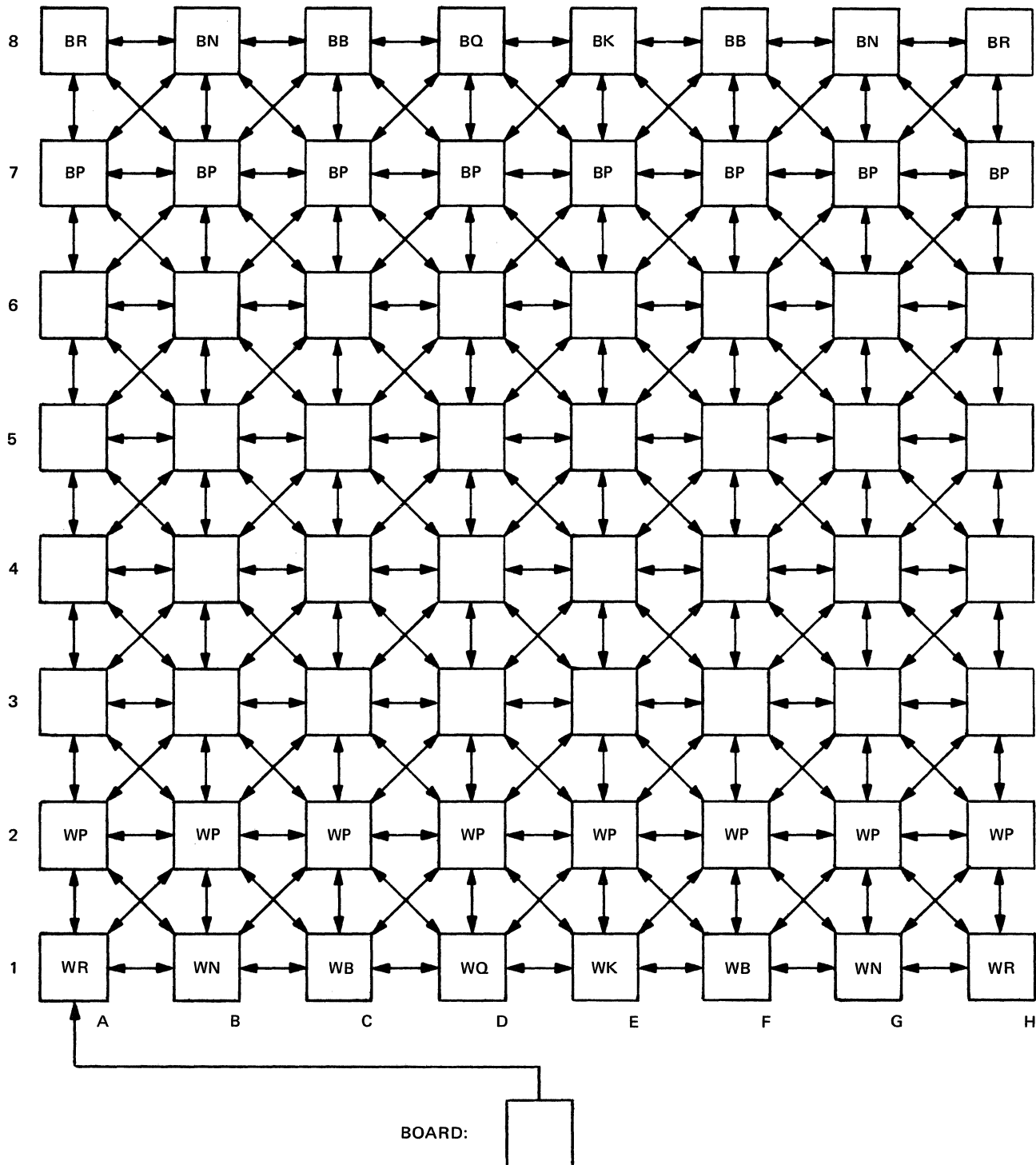


Figure 3.13C. List representation of a chessboard



```

CHESS:PROCEDURE OPTIONS(MAIN);
    CALL BUILD_BOARD;
BUILD_BOARD:
    PROCEDURE;
    DECLARE
        AREA9 AREA (5000),
        1 SQUARE BASED(P),
        2 NAME CHARACTER(2),
        2 MAN CHARACTER(2),
        2 NEIGHBORING_SQUARES,
        3(UP, DOWN,LEFT, RIGHT,
        R_UP,R_DOWN,L_UP,L_DOWN,POINTER)
        POINTER,
        (S,FILE(8),RANK(8),BOARD,
        A1,A2,A3,A4,A5,A6,A7,A8,
        B1,B2,B3,B4,B5,B6,B7,B8,
        C1,C2,C3,C4,C5,C6,C7,C8,
        D1,D2,D3,D4,D5,D6,D7,D8,
        E1,E2,E3,E4,E5,E6,E7,E8,
        F1,F2,F3,F4,F5,F6,F7,F8,
        G1,G2,G3,G4,G5,G6,G7,G8,
        H1,H2,H3,H4,H5,H6,H7,H8)
        EXTERNAL POINTER;
        /* FORM 8 BOARD FILES WITH
        NEIGHBORING SQUARES CONNECTED BY
        UP_POINTERS. */
    DO
        I = 1 TO 8;
        S = NULL;
    DO
        J = 1 TO 8;
        ALLOCATE SQUARE IN (AREA9) SET (P);
        P->UP = S;
        S = P;
    END;
    FILE(I) = S;
    /* USE DOWN_POINTERS TO CONNECT
    NEIGHBORING SQUARES. */
    DO
        I = 1 TO 8;
        P = FILE(I);
        S = NULL;
    DO
        J = 1 TO 8;
        P->DOWN = S;
        S = P;
        P = P->UP;
    END;
    END;
    /* INITIALIZE RANK_POINTERS. */
    P = FILE(1);
    DO
        I = 1 TO 8;
        RANK(I) = P;
    END;
    P = P->UP;
    /* USE RIGHT_POINTERS TO CONNECT
    NEIGHBORING SQUARES. */
    DO
        I = 1 TO 7;
        S = FILE(I);
        P = FILE(I + 1);
    DO
        J = 1 TO 8;
        S->RIGHT = P;
        S = S->UP;
        P = P->UP;
    END;
    END;
    P = FILE(8);
    DO
        I = 1 TO 8;
        P->RIGHT = NULL;
        P = P->UP;
    END;
    /* USE LEFT_POINTERS TO CONNECT
    NEIGHBORING SQUARES. */
    DO
        I = 1 TO 8;
        P = RANK(I);
        S = NULL;
    DO
        J = 1 TO 8;
        P->LEFT = S;
        S = P;
        P = P->RIGHT;
    END;
    END;
    /* USE RIGHT-UP-POINTERS TO CONNECT
    NEIGHBORING SQUARES. */
    DO
        I = 1 TO 7;
        S = FILE(I);
        P = FILE(I + 1);
        P = P->UP;
    DO
        J = 1 TO 7;
        S->R_UP = P;
        S = S->UP;
        P = P->UP;
    END;
    END;
    S = FILE(8);
    DO
        I = 1 TO 8;
        S->R_UP = NULL;
        S = S->UP;
    END;
    /* USE RIGHT-DOWN-POINTERS TO
    CONNECT NEIGHBORING SQUARES. */
    DO
        I = 1 TO 7;
        S = FILE(I);
        S->R_DOWN = NULL;
        S = S->UP;
        P = FILE(I + 1);
    DO
        J = 1 TO 7;
        S->R_DOWN = P;
        S = S->UP;
        P = P->UP;
    END;
    END;
    S = FILE(8);
    DO
        I = 1 TO 8;
        S->R_DOWN = NULL;
        S = S->UP;
    END;
    /* USE LEFT-UP-POINTERS TO CONNECT
    NEIGHBORING SQUARES. */
    S = FILE(1);
    DO
        I = 1 TO 8;
        S->L_UP = NULL;
        S = S->UP;
    END;

```

Figure 3.13D. Building a chessboard and setting chessmen on the board (Continued)

```

END;
DO
  I = 1 TO 7;
  S = FILE(I);
  S = S->UP;
  P = FILE(I + 1);
DO
  J = 1 TO 7;
  P->L_UP = S;
  P = P->UP;
  S = S->UP;
END;
END;
P->L_UP = NULL;
END;
/* USE LEFT-DOWN-POINTERS TO CONNECT
NEIGHBORING SQUARES. */
S = FILE(1);
DO
  I = 1 TO 8;
  S->L_DOWN = NULL;
  S = S->UP;
END;
DO
  I = 1 TO 7;
  S = FILE(I);
  P = FILE(I + 1);
  P->L_DOWN = NULL;
  P = P->UP;
DO
  J = 1 TO 7;
  P->L_DOWN = S;
END;
END;
P = P->UP;
S = S->UP;
END;
END;
/* ASSOCIATE INDIVIDUAL POINTERS
WITH EACH SQUARE. */
A1 = FILE(1); A2 = FILE(2);
A3 = FILE(3); A4 = FILE(4);
A5 = FILE(5); A6 = FILE(6);
A7 = FILE(7); A8 = FILE(8);
B1 = A1->UP; B2 = A2->UP;
B3 = A3->UP; B4 = A4->UP;
B5 = A5->UP; B6 = A6->UP;
B7 = A7->UP; B8 = A8->UP;
C1 = B1->UP; C2 = B2->UP;
C3 = B6->UP; C4 = B4->UP;
C5 = B5->UP; C6 = B6->UP;
C7 = B7->UP; C8 = B8->UP;
D1 = C1->UP; D2 = C2->UP;
D3 = C3->UP; D4 = C4->UP;
D5 = C5->UP; D6 = C6->UP;
D7 = C7->UP; D8 = C8->UP;
E1 = D1->UP; E2 = D2->UP;
E3 = D3->UP; E4 = D4->UP;
E5 = D5->UP; E6 = D6->UP;
E7 = D7->UP; E8 = D8->UP;
F1 = E1->UP; F2 = E2->UP;
F3 = E3->UP; F4 = E4->UP;
F5 = E5->UP; F6 = E6->UP;
F7 = E7->UP; F8 = E8->UP;
G1 = F1->UP; G2 = F2->UP;
G3 = F3->UP; G4 = F4->UP;
G5 = F5->UP; G6 = F6->UP;
G7 = F7->UP; G8 = F8->UP;
H1 = G1->UP; H2 = G2->UP;
H3 = G3->UP; H4 = G4->UP;
H5 = G5->UP; H6 = G6->UP;
H7 = G7->UP; H8 = G8->UP;
/* ASSOCIATE BOARD POINTER WITH A1
POINTER. */
BOARD = A1;
/* ASSIGN NAME TO NAME ELEMENT OF
EACH SQUARE. */
A1->NAME = 'A1'; A2->NAME = 'A2';
A3->NAME = 'A3'; A4->NAME = 'A4';
A5->NAME = 'A5'; A6->NAME = 'A6';
A7->NAME = 'A7'; A8->NAME = 'A8';
B1->NAME = 'B1'; B2->NAME = 'B2';
B3->NAME = 'B3'; B4->NAME = 'B4';
B5->NAME = 'B5'; B6->NAME = 'B6';
B7->NAME = 'B7'; B8->NAME = 'B8';
C1->NAME = 'C1'; C2->NAME = 'C2';
C3->NAME = 'C3'; C4->NAME = 'C4';
C5->NAME = 'C5'; C6->NAME = 'C6';
C7->NAME = 'C7'; C8->NAME = 'C8';
D1->NAME = 'D1'; D2->NAME = 'D2';
D3->NAME = 'D3'; D4->NAME = 'D4';
D5->NAME = 'D5'; D6->NAME = 'D6';
D7->NAME = 'D7'; D8->NAME = 'D8';
E1->NAME = 'E1'; E2->NAME = 'E2';
E3->NAME = 'E3'; E4->NAME = 'E4';
E5->NAME = 'E5'; E6->NAME = 'E6';
E7->NAME = 'E7'; E8->NAME = 'E8';
F1->NAME = 'F1'; F2->NAME = 'F2';
F3->NAME = 'F3'; F4->NAME = 'F4';
F5->NAME = 'F5'; F6->NAME = 'F6';
F7->NAME = 'F7'; F8->NAME = 'F8';
G1->NAME = 'G1'; G2->NAME = 'G2';
G3->NAME = 'G3'; G4->NAME = 'G4';
G5->NAME = 'G5'; G6->NAME = 'G6';
G7->NAME = 'G7'; G8->NAME = 'G8';
H1->NAME = 'H1'; H2->NAME = 'H2';
H3->NAME = 'H3'; H4->NAME = 'H4';
H5->NAME = 'H5'; H6->NAME = 'H6';
H7->NAME = 'H7'; H8->NAME = 'H8';
CALL SET_MEN;
SET_MEN:
PROCEDURE;
DECLARE
  1 SQUARE BASED(P),
  2 NAME CHARACTER(2),
  2 MAN CHARACTER(2),
  2 NEIGHBORING_SQUARES,
  3(UP,DOWN,LEFT,RIGHT,
R_UP,R_DOWN,L_UP,L_DOWN)POINTER,
(A1,A2,A3,A4,A5,A6,A7,A8,
B1,B2,B3,B4,B5,B6,B7,B8,
C1,C2,C3,C4,C5,C6,C7,C8,
D1,D2,D3,D4,D5,D6,D7,D8,
E1,E2,E3,E4,E5,E6,E7,E8,
F1,F2,F3,F4,F5,F6,F7,F8,
G1,G2,G3,G4,G5,G6,G7,G8,
H1,H2,H3,H4,H5,H6,H7,H8,
RANK(8)) EXTERNAL POINTER;
/* ASSIGN MEN TO INITIAL
POSITIONS. */
A1->MAN = 'WR'; A2->MAN = 'WN';
A3->MAN = 'WB'; A4->MAN = 'WQ';
A5->MAN = 'WK'; A6->MAN = 'WB';
A7->MAN = 'WN'; A8->MAN = 'WR';
P = RANK(2);
DO
  I = 1 TO 8;
  P->MAN = 'WP';
  P = P->RIGHT;

```

Figure 3.13D. Building a chessboard and setting chessmen on the board (Continued) •

```

END;
    P = RANK(7);
    DO
        I = 1 TO 8;
        P->MAN = 'BP';
        P = P->RIGHT;
    END;
    H1->MAN = 'BR'; H2->MAN = 'BN';
    H3->MAN = 'BB'; H4->MAN = 'BQ';
    H5->MAN = 'BK'; H6->MAN = 'BB';
    H7->MAN = 'BN'; H8->MAN = 'BR';
    /* BLANK MAN ELEMENTS OF REMAINING
    SQUARES. */
    DO
        I = 3 TO 6;
        P = RANK(I);
        DO
            J = 1 TO 8;
            P->MAN = ' ';
            P = P->RIGHT;
        END;
    END;
    END;
    SET_MEN;
END
    BUILD_BOARD;

PUT SKIP LIST('CHESS EXECUTED');
CLOSE FILE(SYSPRINT);
END CHESS;

```

Figure 3.13D. Building a chessboard and setting chessmen on the board

## REVIEW OF COMPLEX DATA LISTS

Chapter 3 shows how to create complex data lists by extending the simple list organization of Chapter 2. These extensions involve using other types of data in list components besides single characters, storing descriptive information about a list in the head of the list, and using additional pointer elements to obtain alternative orderings of list components besides a simple linear ordering.

Illustrations of complex data lists appear in Figure 3.14. L1 is a two-way circular list of eight components. The head of the list contains a count of the components, and each component possesses two pointers: one for forward linking, the other for backward linking. The list of available storage components, AVAIL, has 16 components. As illustrated, AVAIL need be linked in a forward direction only, because no intricate manipulations are performed on its components other than storing and retrieving available storage.

Complex data lists prove useful in modeling intricate systems that possess discrete arrangements of their parts. Such systems occur in many areas:

- Electrical and communication networks
- Industrial-process scheduling
- PERT and critical path analyses
- Military tactics and logistics
- Switching circuits
- Chemical structures
- Optimization of transportation routes and network flows
- Games and puzzles

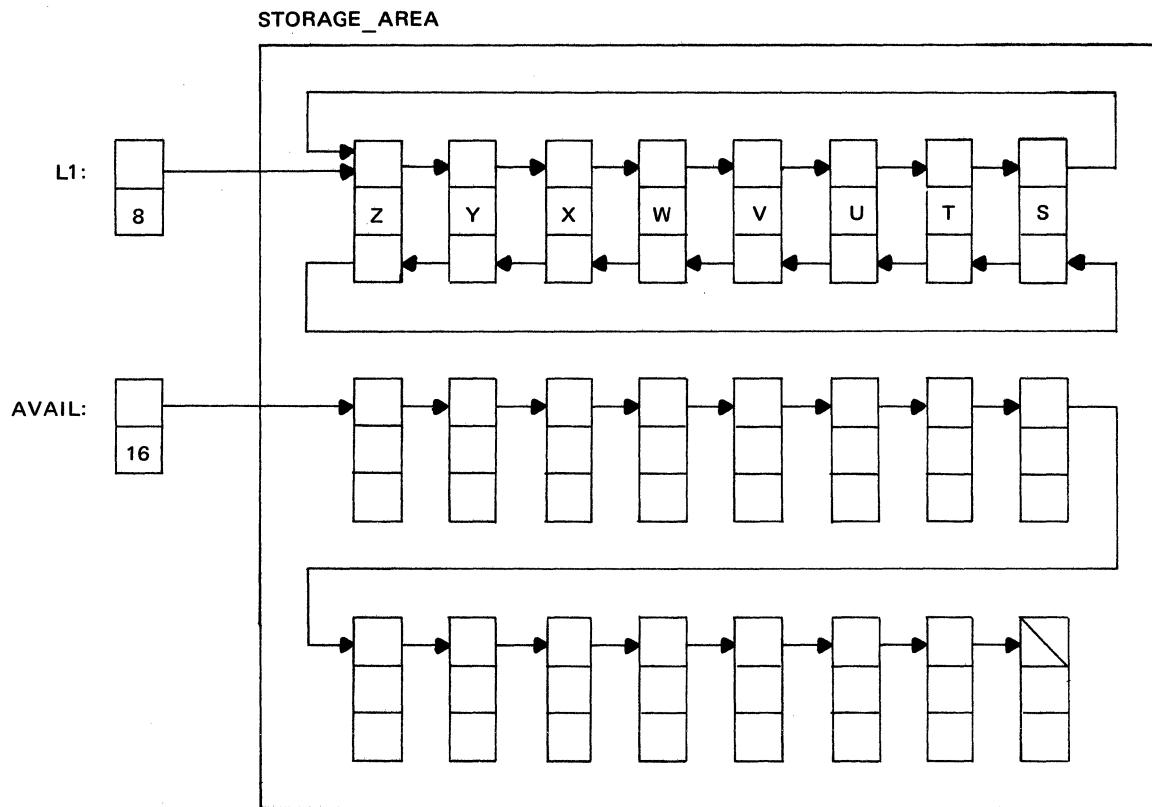


Figure 3.14. Complex data lists

## SUMMARY

1. Chapter 3 shows how the simple list organization used in Chapter 2 can be modified to create more complex data lists.

2. Modifications can include any or all of the essential elements of a simple data list: the head pointer, the component data item, and the component pointer.

3. Descriptive information about the list, such as the size of the list and its name, may be stored in the head of

the list along with the head pointer.

4. Each list component can contain data of any type and precision, including arrays, structures, and arrays of structures.

5. Each list component can contain more than one pointer element, so that components may be linked in circular and multidirectional fashion.

## Appendix 1: Review of Facilities for Subroutines and Functions in PL/I

This appendix reviews the main facilities for organizing and using subroutines and functions in PL/I. It deals with only those features that are used in this manual and does not attempt to cover all aspects of subroutines and functions.

### ORGANIZING SUBROUTINES

A subroutine is a procedure block whose **PROCEDURE** statement has the form:

```
entry-name: PROCEDURE [(parameter-1,parameter-2,  
... ,parameter-n)];
```

The entry name is a statement label that serves as the name of the subroutine. Program control enters the subroutine through its entry name when the entry name is referred to in an invoking **CALL** statement.

Each parameter in a **PROCEDURE** statement is a variable that is used within the subroutine and to which a value may be assigned by the **CALL** statement that invokes the subroutine. The parameters may also be used to return values to the procedure that contains the invoking **CALL** statement. Attributes for each parameter may be declared explicitly, implicitly, or contextually within the subroutine. It is also possible for the **PROCEDURE** statement in a subroutine to specify no parameters.

When an invoking **CALL** statement sends control to a subroutine, the subroutine becomes active and remains active until control encounters either a **RETURN** statement or the final **END** statement of the subroutine. At that point, control returns to the statement immediately following the invoking **CALL** statement.

When used in a subroutine, a **RETURN** statement has the form:

```
RETURN;
```

An arbitrary number of **RETURN** statements may appear in a subroutine.

Example:

```
SUM: PROCEDURE(VALUE1, VALUE2, TOTAL);  
  
    DECLARE (VALUE1, VALUE2) FIXED  
           DECIMAL(3),  
  
           TOTAL FIXED DECIMAL(4);  
  
    TOTAL = VALUE1 + VALUE2;  
  
    RETURN;  
  
END SUM;
```

The name of this subroutine is **SUM**, and it contains three parameters: **VALUE1**, **VALUE2**, and **TOTAL**. Parameters **VALUE1** and **VALUE2** receive values when the subroutine is invoked. The subroutine then adds the values and assigns them to **TOTAL**. The value of **TOTAL** also becomes available in the invoking procedure through an associated argument variable. Execution of the **RETURN** statement sends control to the statement immediately following the invoking statement. Had the **RETURN** statement been omitted in this example, the **END** statement would have returned control.

### USING SUBROUTINES

Execution of a subroutine requires that it be invoked by a **CALL** statement, which has the form:

```
CALL entry-name(argument-1,argument-2, . . . ,argument-n);
```

The entry name identifies the label attached to the **PROCEDURE** statement of the invoked subroutine. Each argument can be a constant, a variable, or an expression, the value of which is automatically assigned to the corresponding parameter specified in the **PROCEDURE** statement of the subroutine.

Within the invoking procedure, explicit declaration of the entry name for a subroutine occurs with the ENTRY attribute, which has the form:

```
ENTRY(parameter-attribute-list-1, . . . ,parameter-attribute-list-n);
```

Each parameter attribute list specifies the attributes of the corresponding parameter in the subroutine. The parameter attribute lists permit the invoking procedure to perform necessary conversions of argument values before they become associated with the corresponding parameter variables.

Example:

```
T_SUB:PROCEDURE OPTIONS(MAIN);
  DECLARE (A,B) FIXED DECIMAL(3),
  C FIXED DECIMAL(4),
  D CHARACTER(74),
  SUM ENTRY(FIXED DECIMAL(3),
  FIXED DECIMAL(3),
  FIXED DECIMAL(4));
  ON ENDFILE GO TO OVER;
START:
  GET EDIT(A,B,D)(F(3),F(3),A(74));
  CALL SUM(A,B,C);
  PUT EDIT(A,B,C)(F(6), F(6), F(7));
  PUT SKIP;
  GO TO START;
SUM:PROCEDURE(VALUE1, VALUE2, TOTAL);
  DECLARE(VALUE1, VALUE2)
  FIXED DECIMAL(3),
  TOTAL FIXED DECIMAL(4);
  TOTAL = VALUE1 + VALUE2;
  RETURN;
END SUM;
OVER:
END T_SUB;
```

This example shows how subroutine SUM may be invoked from the main procedure T-SUB. The program gets a card from the standard system-input file, SYSIN, and assigns the integer in columns 1 through 3 to variable A and the integer in columns 4 through 6 to variable B. Invocation of subroutine SUM then occurs with the CALL statement:

```
CALL SUM(A, B, C);
```

Arguments A, B, C become associated with parameters VALUE1, VALUE2, and TOTAL in the subroutine. The values of VALUE1 and VALUE2 become identical with the values of A and B. When the sum of VALUE1 and VALUE2 is assigned to TOTAL, the value of argument C becomes identical with the value of TOTAL.

Control returns from the subroutine to the PUT statement located immediately after the CALL statement.

T-SUB then prints the value of A, B, and C on a single line of the standard system-output file, SYSPRINT. These steps are repeated for each input card until the end of the SYSIN file is reached.

## ORGANIZING FUNCTIONS

A function is a procedure block whose PROCEDURE statement has the form:

```
entry-name: PROCEDURE(parameter-1,parameter-2,
. . . ,parameter-n) RETURNS(result-attribute-list);
```

This statement is similar to the PROCEDURE statement for subroutines except that it specifies the attributes of the value returned by the function. The result attribute list is optional; if it is not used, the attributes of the result are determined implicitly from the first letter of the function entry name (FIXED BINARY(15) for letters I through N and FLOAT DECIMAL(6) for all other alphabetic characters).

A RETURN statement returns control from the function and also specifies the result of the function:

```
RETURN(element-expression);
```

The expression, which can be a constant, a variable, or an operational expression, must be present and must specify a single value; it cannot represent an array or a structure value. A function can contain an arbitrary number of RETURN statements, each with a different element expression:

Example:

```
MULT: PROCEDURE(VALUE1, VALUE2) RETURNS
(FIXED DECIMAL(6));
```

```
  DECLARE (VALUE1, VALUE2) FIXED
  DECIMAL(3);
```

```
  RETURN (VALUE1 * VALUE2);
```

```
END MULT;
```

The function procedure MULT contains two parameters: VALUE1 and VALUE2. These variables receive values when the function is invoked. The RETURN statement evaluates the product of VALUE1 and VALUE2 and returns the result as the value of the function. The attributes of the result appear in the PROCEDURE statement as FIXED DECIMAL(6).

## USING FUNCTIONS

A function is not invoked by a statement but by a function reference, which has the form:

```
entry-name(argument-1, argument-2, . . ., argument-n)
```

A function reference can appear wherever an expression is permitted in PL/I, and the value of the reference is the result returned by the function procedure. When program control returns from the function procedure, it continues from the point of the invoking function reference. Each argument in a function reference can be a constant, a variable, or an expression (which itself may include a function reference). The value of each argument is automatically assigned to the corresponding parameter specified in the PROCEDURE statement of the function.

Explicit declaration of the entry name for a subroutine occurs with the ENTRY and RETURNS attributes, which have the form:

```
ENTRY(parameter-attribute-list1,  
...parameter-attribute-listn)
```

```
RETURNS(result-attribute-list)
```

Each parameter attribute list in the ENTRY attribute specifies the attributes of the corresponding parameter in the function. The result attribute list contains the attributes of the value returned by the function procedure.

Example:

```
T_FUNCT:PROCEDURE OPTIONS(MAIN);  
  DECLARE (A,B)FIXED DECIMAL(3),  
          C FIXED DECIMAL(6),  
          D CHARACTER (74),  
          MULT ENTRY(FIXED DECIMAL(3),  
                    FIXED DECIMAL(3))  
          RETURNS(FIXED DECIMAL(6));  
  ON ENDFILE GO TO OVER;  
START:  
  GET EDIT(A,B,D)(F(3),F(3),A(74));  
  C = MULT(A,B);  
  PUT EDIT(A,B,C)(F(6), F(6), F(9));  
  PUT SKIP;  
  GO TO START;  
MULT:PROCEDURE(VALUE1, VALUE2)  
  RETURNS (FIXED DECIMAL(6));  
  DECLARE(VALUE1, VALUE2)  
  FIXED DECIMAL(3);  
  RETURN (VALUE1 * VALUE2);  
  END MULT;  
OVER:  
END T_FUNCT;
```

This example shows how function MULT may be invoked from the main procedure T-FUNCT. The program gets a card from the standard system-input file SYSIN and assigns the integer in columns 1 through 3 to variable A and the integer in columns 4 through 6 to variable B. Function MULT is invoked by the reference MULT(A,B), which appears in the assignment statement:

```
C = MULT(A,B);
```

Arguments A and B become associated with parameters VALUE1 and VALUE2 in the function procedure. VALUE1 and VALUE2 receive the values of A and B, and the function returns the product of these values. The product is then assigned to variable C.

The program prints the values of A, B, and C on a single line of the standard system-output file, SYSPRINT. These steps are repeated for each input card until the end of the SYSIN file is reached.

## DUMMY ARGUMENTS

When an argument becomes associated with a parameter, the name of the argument, not its value, is passed to a subroutine or function. Some arguments, however, do not have names. For example, a constant has no name, nor does an operational expression. These arguments, therefore, cannot be associated directly with parameters. The compiler must provide storage for such arguments and create an internal name for each. These internal names are called *dummy arguments*. They are passed to the invoked subroutine or function in place of the original arguments that have no names.

Dummy arguments cannot be addressed directly in PL/I. Any changes, however, in the values of their associated parameters will be reflected in the dummy arguments, but the values of the original arguments will remain unchanged.

The compiler creates a dummy argument when:

1. An argument is a constant.
2. An argument is an expression involving operators.
3. An argument is enclosed in parentheses.
4. An argument is a function reference.
5. An argument is a variable whose data attributes are different from the data attributes declared for the associated parameter in an ENTRY attribute within the invoking block.

Changes to the value of a parameter will be reflected in the value of the original argument only if a dummy argument is not passed. When a dummy argument is not created, the argument name is passed directly to the invoked procedure, and the associated parameter becomes identical with the argument.

If an ENTRY attribute is not used in an invoking block to describe the attributes of the parameters in the invoked procedure, the compiler assumes that the arguments are compatible with their associated parameters. If they are not compatible, a specification interrupt may occur.

## RECURSIVE SUBROUTINES AND FUNCTIONS

PL/I allows an active subroutine or function to invoke another procedure and permits a sequence of such invocations to proceed to an arbitrary depth. Reinvocation of an already active procedure either from within itself or from within another active procedure may also occur and forms a *recursive invocation*. The reinvoked procedure is called a *recursive procedure* which must contain the attribute RECURSIVE in its PROCEDURE statement:

```
entry-name: PROCEDURE(parameter-1, . . .,parameter-n)
```

```
    RETURNS(result-attribute-list) RECURSIVE;
```

Recursion affects variables that possess automatic storage. Recursive invocation of a procedure causes the storage for each automatic variable within the procedure to be reallocated and also causes the previously allocated storage to be saved automatically in a push-down stack. Termination of each activation of a recursive procedure restores the most previously allocated storage for the automatic variables.

Recursion does not affect the storage for static variables, controlled variables, or based variables. The storage and values of such variables remain directly available at all levels of recursion.

Example:

```
FACTORIAL: PROCEDURE(N) RETURNS(FIXED  
    DECIMAL(6)) RECURSIVE;
```

```
    DECLARE N FIXED DECIMAL(2);
```

```
    IF N <= 1 THEN RETURN(1);
```

```
    RETURN(N* FACTORIAL(N-1));
```

```
END FACTORIAL;
```

FACTORIAL is a recursive function procedure that contains one parameter, N, whose value is a positive integer between zero and nine. The function computes the factorial value of N, which is the product of the integers from one to N and which is denoted mathematically by the

expression N!. The factorial of three (3!), for example, is six:

$$\text{FACTORIAL}(3) = 3! = 1 \cdot 2 \cdot 3 = 6$$

By mathematical convention, the factorial of zero is one (0! = 1). Note that the factorial function may be defined in terms of itself and is, therefore, recursive:

$$3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 = 6$$

This recursiveness is represented in the FACTORIAL function by the following expression:

$$N * \text{FACTORIAL}(N-1)$$

Since this expression appears within the FACTORIAL procedure, the reference FACTORIAL(N-1) causes recursive invocation of FACTORIAL.

Figure A 1.1 illustrates the flow of program control through recursive invocations of the FACTORIAL function. To simplify the presentation, the diagram duplicates the function procedure at each stage of recursion. It also shows the argument value passed by each invocation, the flow of control into and out of the function, and the value returned by each invoking reference. Since parameter N has the automatic storage class, storage is automatically allocated for N at each level of recursion, and its value at the previous level is saved for reuse when control returns to that level. Solid lines in the diagram denote flow of control from an invoking reference, and beaded lines represent flow of control from a RETURN statement.

The diagram begins at the top with execution of the assignment statement:

```
RESULT = FACTORIAL(3);
```

Before a value can be assigned to variable RESULT on the left, the expression on the right must be evaluated. This expression consists of a reference to the FACTORIAL function with an argument value of three. When control enters the function, storage is allocated for parameter N, and it receives the value three. Because N is greater than one, the following statement is executed:

```
RETURN(N*FACTORIAL(N-1));
```

This statement cannot return control until its expression is evaluated. However, the expression itself contains a reference to FACTORIAL that causes recursive invocation of the function with an argument value of two (3-1 = 2).



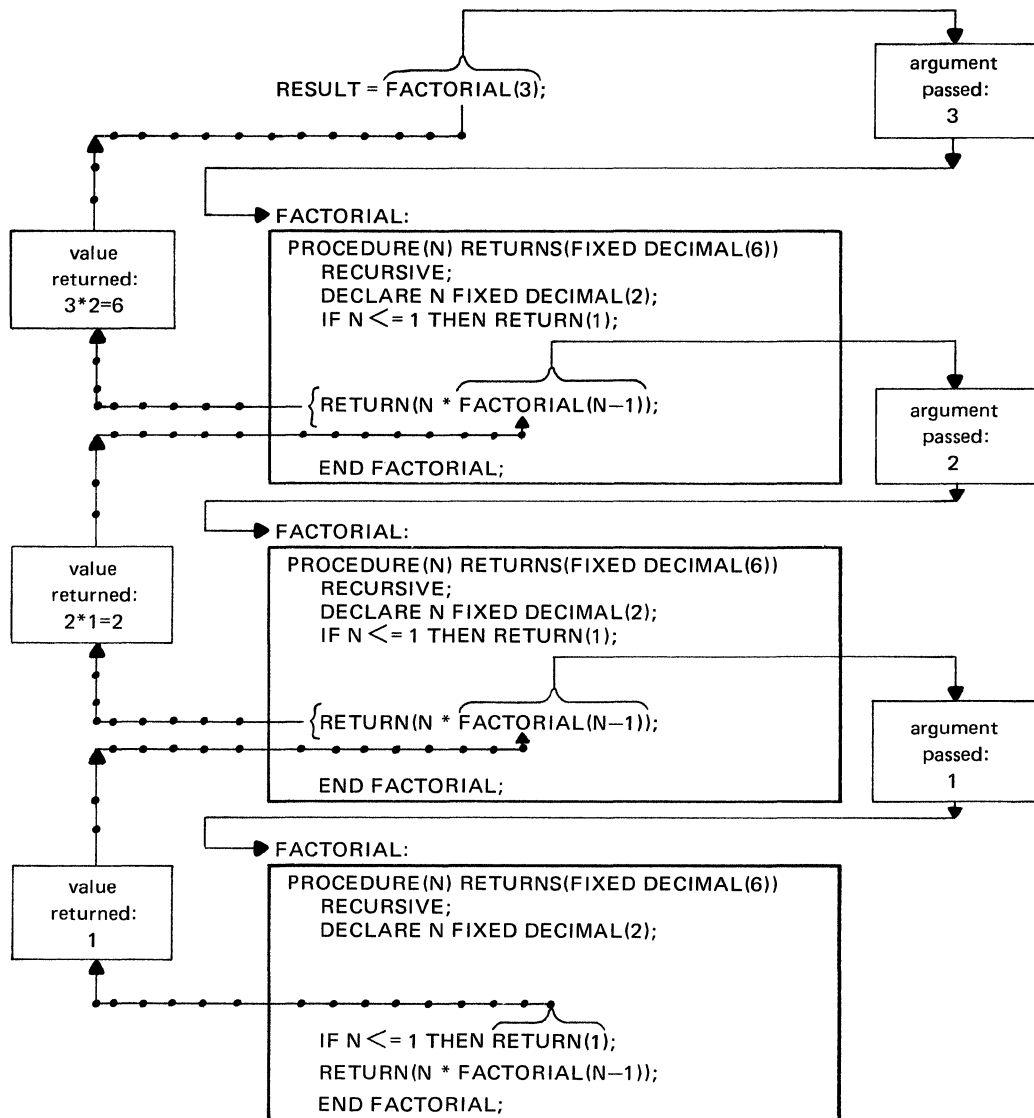


Figure A1.1. Computing FACTORIAL(3) recursively

This second invocation of FACTORIAL, which is represented by the second copy of the function in Figure A 1.1, causes the current value (3) of N to be saved and new storage to be allocated for N with a value of two. Again N has a value greater than one, which causes the previous RETURN statement to be reexecuted (in the second copy of the function):

RETURN(N\* FACTORIAL(N-1));

Return of control is suspended once more until the expression in the RETURN statement is evaluated. The evaluation causes a third invocation of FACTORIAL with an argument value of one ( $2-1 = 1$ ).

When control enters the function for the third time (represented by the third copy of FACTORIAL in Figure

A1.1, the current value (2) of N is saved, and new storage is allocated for N with a value of one. At this stage of recursion, the value of one for N causes the following statement to be executed:

RETURN(1);

Since this statement does not contain a reference to FACTORIAL, no further recursion occurs, and control returns a value of one to the previous point of invocation.

The previous point of invocation occurred within the second copy of the function and is associated with the statement:

RETURN(N\* FACTORIAL(N-1));

In this statement, N has a value of two, and the value returned to the reference FACTORIAL(N-1) is one. This statement, therefore, returns a value of two ( $2*1 = 2$ ) to the still previous point of invocation, which occurred within the first copy of the function and is also associated with the following RETURN statement:

```
RETURN(N * FACTORIAL(N-1));
```

This time the value of N is three, and the value returned to the function reference is two. Hence, this statement returns a value of six ( $3*2 = 6$ ) to the previous point of invocation, which occurred on the right side of the assignment statement:

```
RESULT = FACTORIAL(3);
```

At this point, the value of six for the function reference is assigned to variable RESULT.

Program T-FACT, shown in Figure A1.2, computes the factorial value of all the integers from zero through nine and prints the results as shown in Figure A1.3.

```
T_FACT:PROCEDURE OPTIONS(MAIN);
DECLARE
  N FIXED DECIMAL(2),
  RESULT FIXED DECIMAL(6),
  FACTORIAL ENTRY(FIXED DECIMAL(2))
    RETURNS (FIXED DECIMAL(6));
PUT LIST(' N    FACTORIAL OF N');
PUT SKIP(2);
DO N = 0 TO 9 BY 1;
  RESULT = FACTORIAL(N);
  PUT EDIT(N, RESULT)(F(2), X(5), F(6));
  PUT SKIP;
END;
FACTORIAL:PROCEDURE(N)
  RETURNS(FIXED DECIMAL(6))RECURSIVE;
DECLARE N FIXED DECIMAL(2);
IF N <= 1 THEN RETURN(1);
RETURN(N * FACTORIAL(N - 1));
END FACTORIAL;
END T_FACT;
```

Figure A1.2. Program T\_FACT

N	FACTORIAL OF N
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880

Figure A1.3. Printout produced by program T\_FACT

## SUMMARY OF FACILITIES FOR SUBROUTINES AND FUNCTIONS

This summary divides the facilities for subroutines and functions into three categories: statements, attributes, and invocations. The description of each facility uses standard PL/I syntax. Brackets [] denote optional items, and an ellipsis ... specifies optional repetition of the preceding item.

### Statements

```
entry-name: PROCEDURE [(parameter
  [,parameter] ...)]
  [RETURNS(result-attribute-list)] [RECURSIVE];
RETURN [(element-expression)];
```

### Attributes

```
ENTRY [(parameter-attribute-list [,parameter-
  attribute-list] ...)]
RETURNS (result-attribute-list)
```

### Invocations

#### Subroutine Invocation

```
CALL entry-name [(argument [,argument] ...)];
```

#### Function Invocation

```
entry-name [(argument [,argument] ...)]
```

## Appendix 2: Summary of List-Processing Facilities

The following summary divides the list-processing facilities into five categories: attributes, built-in functions, ON-conditions, statements, and miscellaneous features. Facilities within each category appear in alphabetic order.

When used in the format of each facility, brackets [] denote optional items; braces {} indicate that a choice must be made from the enclosed items, which are separated by an “or” symbol |; and an ellipsis ... specifies optional repetition of the preceding item.

### Attributes

AREA [(size-expression) | (\*)]  
BASED(element-pointer-variable)  
OFFSET (area-variable)  
POINTER

### Built-In Functions

ADDR(argument-variable)  
EMPTY  
NULL  
NULLO

### ON-Condition

AREA

### Statements

ALLOCATE based-variable  
    [IN(area-variable)]  
    [SET(pointer-variable)]  
    [,based-variable  
    [IN(area-variable)]  
    [SET(pointer-variable)]] ... ;  
FREE based-variable  
    [IN(area-variable)]  
    [,based-variable  
    [IN(area-variable)]] ... ;  
LOCATE based-variable  
    FILE(file-name)  
    SET(pointer-variable);  
READ FILE (file-name)  
    SET(pointer-variable);

### Miscellaneous Features

Pointer-qualification symbol:

->

REFER option:

element-variable REFER (element-variable)

## Bibliography

The following texts provide additional sources of information on list processing.

Barron, D. W. *Recursive Techniques in Programming*. New York: American Elsevier Publishing Company, Inc., 1968.

Berkeley, Edmund C., and Bobrow, D. G. (eds.) *The Programming Language LSIP: Its Operation and Applications*. Cambridge, Massachusetts: The M.I.T. Press, second printing, 1966.

Foster, J. M. *List Processing*. New York: American Elsevier Publishing Company, Inc., 1967.

Fox, L. (ed.) *Advances in Programming and Non-Numerical Computation*. Oxford, England: Pergamon Press, 1966.

McCarthy, John, Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I. *LIST 1.5 Programmer's Manual*. Cambridge, Massachusetts: The M.I.T. Press, second edition, 1965.

## Index

	<i>Page</i>		<i>Page</i>
\$EDIT subroutine . . . . .	72	FORM_TWO_WAY subroutine . . . . .	94
A_EXPAND subroutine . . . . .	75	FORM_TWO_WAY_CIRCULAR subroutine . . . . .	95
ADD_INT subroutine . . . . .	81	FORM_WATER subroutine . . . . .	102
ADDR built-in-function . . . . .	1	FREE statement . . . . .	5
ADDRESS_N function . . . . .	14	GET_DATA function . . . . .	17
ADDRESS_NEXT function . . . . .	15	GET_FD function . . . . .	26
ADDRESS_N2 function . . . . .	92	GET_LD function . . . . .	27
ALLOCATE statement . . . . .	4	GET_ND function . . . . .	26
AREA_OPEN subroutine . . . . .	12	GET_ND1 function . . . . .	28
AREA_OPEN2 subroutine . . . . .	91	GET_POINTER function . . . . .	17
AREA_ON-condition . . . . .	7	Head information . . . . .	89
Area variables . . . . .	5	IN option . . . . .	6
ASSIGN_LIST subroutine . . . . .	45	INSERT_FD subroutine . . . . .	22
ASSIGN_SUB subroutine . . . . .	45	INSERT_LD subroutine . . . . .	22
Assigning pointers . . . . .	2	INSERT_LIST subroutine . . . . .	42
AVAIL . . . . .	12	INSERT_ND subroutine . . . . .	21
Based attribute . . . . .	3	INSERT_ND1 subroutine . . . . .	24
Based storage . . . . .	4	INSERT_SUB subroutine . . . . .	41
Based variables . . . . .	2	LINK subroutine . . . . .	48
CATENATE subroutine . . . . .	51	Linking allocations . . . . .	7
CHES procedure . . . . .	107	LINKR subroutine . . . . .	70
COMPARE function . . . . .	56	LIST_TO_STRING subroutine . . . . .	62
COMPARER function . . . . .	72	NULL built-in-function . . . . .	2
Comparing pointers . . . . .	2	P_CNTRT subroutine . . . . .	79
Compile-time statements . . . . .	89	P_EXPAND subroutine . . . . .	77
Contextual pointers . . . . .	4	Pointer variables . . . . .	1
Creating a list . . . . .	11	Qualifying based variables . . . . .	3
Data lists . . . . .	9	RECURSIVE option . . . . .	60
DE_EDIT subroutine . . . . .	74	REMOVE_FD function . . . . .	34
DELETE_FD subroutine . . . . .	30	REMOVE_LD function . . . . .	34
DELETE_LD subroutine . . . . .	30	REMOVE_ND function . . . . .	33
DELETE_LIST subroutine . . . . .	44	REPLACE_FD subroutine . . . . .	36
DELETE_ND subroutine . . . . .	29	REPLACE_LD subroutine . . . . .	37
DELETE_ND1 subroutine . . . . .	32	REPLACE_ND subroutine . . . . .	36
DELETE_ND2 subroutine . . . . .	92	Restrictions on based variables . . . . .	3
DELETE_SUB subroutine . . . . .	43	REVERSE subroutine . . . . .	58
DELETER subroutine . . . . .	64	SET_DATA subroutine . . . . .	15
DGATHER procedure . . . . .	85	SET_POINTER subroutine . . . . .	16
EMPTY built-in-function . . . . .	7	SIZE function . . . . .	19
EQUAL function . . . . .	55	SIZER function . . . . .	66
EQUALR function . . . . .	71	SIZE1 function . . . . .	20
FIND_D function . . . . .	38	SORT subroutine . . . . .	59
FIND_LIST function . . . . .	54	SPLIT subroutine . . . . .	49
FINDR function . . . . .	69	STRING_TO_LIST subroutine . . . . .	61
FORM_BINARY_TREE subroutine . . . . .	99	SUB_INT subroutine . . . . .	83
FORM_CIRCULAR_TO_HEAD subroutine . . . . .	95	SWAP subroutine . . . . .	40
FORM_PYRAMID subroutine . . . . .	104		



**International Business Machines Corporation**  
**Data Processing Division**  
**112 East Post Road, White Plains, New York 10601**  
**(USA only)**

**IBM World Trade Corporation**  
**821 United Nations Plaza, New York, New York 10017**  
**(International)**

**READER'S COMMENT FORM**

Techniques for Processing Data Lists in PL/I

GF20-0018-0

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

---

**COMMENTS**

—  
fold

—  
fold

—  
fold

—  
fold

**YOUR COMMENTS PLEASE...**

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

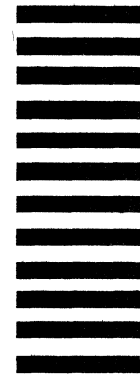
Please note that requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or the IBM branch office serving your locality.

fold

fold

FIRST CLASS  
PERMIT NO. 1359  
WHITE PLAINS, N. Y.

**BUSINESS REPLY MAIL**  
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES



POSTAGE WILL BE PAID BY...

IBM Corporation  
112 East Post Road  
White Plains, N. Y. 10601

Attention: Technical Publications

fold

fold



International Business Machines Corporation  
Data Processing Division  
112 East Post Road, White Plains, N. Y. 10601  
[USA Only]

IBM World Trade Corporation  
821 United Nations Plaza, New York, New York 10017  
[International]

IBM CORPORATION