ARPA ORDER NO. 189-1

MEMORANDUM
RM-5290-ARPA
JULY 1967

DATALESS PROGRAMMING

R. M. Balzer

PREPARED FOR:
ADVANCED RESEARCH PROJECTS AGENCY

The RAND Corporation
SANTA MONICA • CALIFORNIA

ARPA ORDER NO. 189-1

MEMORANDUM
RM-5290-ARPA
JULY 1967

DATALESS PROGRAMMING

R. M. Balzer

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Any views or conclusions contained in this Memorandum should not be interpreted as representing the official opinion or policy of ARPA.

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA • 90406

Published by The RAND Corporation

PREFACE

This Memorandum presents the preliminary specifications of a new computer language designed to alleviate the problems of having to choose a data representation before coding a program. This study, part of the search for techniques to facilitate the man-computer interface, should be of particular interest to those concerned with the choice of languages to provide a proper programming environment for research and development applications.

SUMMARY

A programmer using existing programming languages almost always encounters difficulties because he has to choose a data representation before coding a problem. The Dataless Programming System is designed to alleviate these difficulties. A separation between the description of a process to manipulate data (a program) and the description of the data being manipulated (the data representation) is achieved by representing all data in a syntactically identical form and treating it in a homogeneous manner. Various facilities--generators, bugs, search expressions, and implied qualification of data references--are provided to support this separation of program from data representation.

The main body of this Memorandum describes the preliminary specifications of the Dataless Programming Language, a high-level algebraic language which is an extension of PL/1 and uses its syntax (with some exceptions). Separate sections of the Memorandum discuss: specifying hierarchical data references; maintaining the language's two types of pointers; defining individual members of a data collection; special features of the language; its restricted implementation; expected advantages and difficulties. A final section provides two Dataless Programming examples, with commentaries.

ACKNOWLEDGMENTS

I am indebted to J. C. Shaw and Dr. Allen Newell for their comments and suggestions on the topics contained in this Memorandum.

CONTENTS

PREFACE iii

SUMMARY v

ACKNOWLEDGMENTS vii

Section

I. INTRODUCTION 1

II. NOTATION 4

III. THE DATALESS PROGRAMMING LANGUAGE 7

IV. HIERARCHICAL DATA REFERENCES 19

V. MAINTAINING POINTERS IN A DYNAMIC ENVIRONMENT 25

VI. DATA DEFINITION 27

VII. FURTHER LANGUAGE FEATURES 30

VIII. IMPLEMENTATION 31

IX. EXPECTED ADVANTAGES AND DIFFICULTIES 32

X. DATALESS PROGRAMMING EXAMPLES 35

REFERENCES 43

I. INTRODUCTION

A programmer using existing programming languages typically codes a problem by 1) defining it, then 2) analyzing the processing requirements, and 3) on the basis of these requirements, choosing a data representation, and finally, 4) coding the problem. Almost always, difficulties arise because necessary processing not envisioned in the analysis phase makes the chosen data representation inappropriate because of a lack of space, efficiency, ease of use or some combination of these. The decision is then made to either live with these difficulties or change the data representation. Unfortunately, changing the data representation usually involves making extensive changes to the code already written. Furthermore, there is no assurance that this dilemma will not recur with the new data representation.

The Dataless Programming System is designed to help alleviate this problem. All data and function references are expressed in a single canonical form, so that the alteration of a data representation has no syntactic effect on the program, but only affects the internal processing associated with the data or function references--i.e., changing the data representation does not affect the program's source statements. Unless the change in the data representation causes a change in data values (e.g., a change in precision), the values produced by the program will be unaltered. (Naturally, the behavior of the program in terms of efficiency, speed, and space required may be affected by any change in data representation.)

Hence, the data representation can be specified after the program is written, and analysis of the problem definition and code production can be integrated into a single phase. The programmer should be able to construct his program in terms of the logical processing required without regard to either the representation of data or the method of accessing and updating. This concept we call "Dataless Programming."

Dataless Programming is more than a new language. It conceives of a program as the specification of a set of manipulations to be performed on a set of data values, and that this specification should be independent of the form in which these data values are represented. To achieve such independence, there must be a set of declarations that tell the programming system how to retrieve and store data values from the particular representation being used. The independence thus achieved will allow the programmer 1) to disregard, while specifying the program, the details of data processing, memory space requirements, and matching of data representation to the processing done on it; and 2) to handle them, instead, during the data declaration phase.

The Dataless Programming Language is the embodiment of the above concept. It will also help simplify the construction and debugging of programs by providing high-level data-handling and control facilities (insertion, deletion, generators, bugs, search expressions, and implied qualifications of data references) and a STATE-statement (enabling a wide range of monitoring and tracing facilities to be specified).

Throughout the design of the language, an effort has been made to allow the programmer to express his intent

more clearly through new program statements rather than in separate comments. That is, we have attempted to make the purpose of commonly occurring statement groups more apparent by combining them into an appropriately worded statement. Examples of this effort are the FOR-clause, search-expressions, and the extended IF-statement. The programmer can also increase the program's self-documentation through the use of mnemonic names for data items which are then defined as functions (as has been done in the programming examples in Sec. X below).

II. NOTATION

The syntax of the Dataless Programming Language will be described in PL/1 syntax notation,[†] the relevant parts of which are:

1) A notation variable, the name of a general class of elements in the programming language, which must consist of:

   a) Lower-case letters, decimal digits, and hyphens and must begin with a letter.

   b) A combination of lower-case and upper-case letters. There must be one portion in all lower-case letters and one portion in all upper-case letters, and the two portions must be separated by a hyphen.

Examples:

   a) "digit"--Denotes the occurrence of a digit, which may be 0 through 9 inclusive.

   b) "specification"--Denotes the occurrence of the notation variable named specification (explained below, p. 15).

   c) "DO-statement"--Denotes the occurrence of a DO statement. The upper-case letters are used for emphasis.

2) A notation constant denotes the literal occurrence of the characters represented. It consists either of all capital letters or of a special character:

```
DECLARE identifier FIXED;
```

This denotes the literal occurrence of the word DECLARE followed by the variable "identifier" followed in

[†]As defined in Ref. 1, pp. 11-17.

turn by the literal occurrence of the word FIXED and the semicolon.

3) The term "syntactical unit," used in subsequent rules, is defined as one of the following:

a) a single variable or constant,

or

b) any collection of variables, constants, syntax-language symbols, and reserved words surrounded by braces or brackets.

4) Braces ({}) are used to denote grouping, and the vertical stroke (|) indicates that a choice is to be made:

```
identifier {FIXED | FLOAT}
```

The above example indicates that the variable "identifier" must be followed by the literal occurrence of either the word FIXED or the word FLOAT.

5) Square brackets ([]) denote options. Anything enclosed in brackets may appear one time or may not appear at all:

```
CHARACTER·(length) [VARYING]
```

This denotes the literal occurrence of the word CHARACTER followed by the variable "length" enclosed in parentheses and optionally followed by the literal occurrence of the word VARYING.

6) Three dots (...) denote the occurrence of the immediately preceding syntactical unit one or more times in succession:

[digit] ...

The variable, "digit," may or may not occur since it is surrounded by brackets. If it does occur, it may be repeated one or more times.

III. THE DATALESS PROGRAMMING LANGUAGE

The Dataless Programming Language is a high-level algebraic language which is an extension of PL/1 and uses PL/1 syntax, except as noted. The language is based on a single canonical representation for all data and function references. This form is:

{collection-name|
function-name} [(expression [, expression] ...)]

If the name identifies a function name, the expressions, if any, are interpreted as parameters. If the name identifies a collection name, the semantic interpretation is that there exists a collection (or group) of data which has a common, unique name (the collection name). This collection has the property that, given a single integer index (computed from the expression, called the index expression, which follows the collection name), there exist algorithms which use the value of this index to operate on the data collection, performing the functions of accessing and updating the datum selected and inserting or deleting a datum from the collection. The system assumes that there exist such algorithms, determining what they are not from the form of the program's source statements but rather from explicit declarations supplied with the program. Hence, changing a collection from an array to a list does not change the source statements but merely changes a data declaration. Also, since function references are syntactically identical with data references, the user can change

a function reference to a data reference, or vice versa, by merely changing a declaration.

Separating the data definition and its implied data-handling routines from the common appearance of data or function references in the source code provides the main power of the language. This feature allows the user to program in a top-down fashion in terms of the logical pieces of information necessary for the required processing. Once the user has completed this programming, he can determine, for each collection, what data representation is suitable for the processing involving that collection. He may also decide that a piece of information should not be supplied by a data reference on a collection, but should be supplied by a function through a calculation or series of calculations on other information--either data or further function references. The user may use the language's full power in defining either a function or the data-handling routine for his data representations, and so is able to program each part of his problem in a top-down fashion.

Logically, since the data-handling routines operate with a single index, the data representations are ordered lists. The system allows the following data representations (for which it provides the data-handling routines):

1) ARRAY
2) LIST (forward links only)
3) DOUBLE LIST (list with both forward and backward links)
4) RING (forward links only)
5) DOUBLE RING (ring with both forward and backward links)

6) STRUCTURE

7) structures of any of the above (e.g., arrays of lists of structures).

In addition, any other ordered data representation can be used by providing the necessary data-handling algorithms. These algorithms can consist of a call to an external procedure and can be written in any language desired (including assembly language). Hence, within the restriction of ordered data representations, the representations allowed are completely open ended. This class is large and significant, covering most currently used representations. Clearly, however, some representations (e.g., colored pictures) lie outside this class; and we have no ideas on ways to incorporate these into the Dataless Programming Language.

Since all data manipulations of a collection are performed by the data-handling routines, these routines can be altered to provide a powerful monitoring capability. By suitably altering the update routine for a data collection, the system can be notified any time a member of that collection gets updated. Hence, the system can perform any desired action as a response to the occurrence of a particular state of the program variable, i.e., it can handle STATE-statements. The form for these STATE-statements is:

{ON | WHENEVER} (boolean-expression) statement;

For implementation purposes, the boolean-expression is restricted so that all variables which effect its truth

value must explicitly be part of the expression (enabling the system to determine which variables require checking), and the evaluation of the boolean-expression must not change the value of any of these variables (and care should be taken that it also does not change the value of any other program variables which affect the behavior of the program, so that its removal does not affect the program). Whenever one of these explicitly named variables is updated, the boolean-expression is evaluated and if true, the associated statement is executed. As an example, if the STATE-statement

```
ON (x = y | x = z) CALL print;
```

occurred, the data updating routines for x, y, and z would be altered so that this on-condition was checked whenever either x, y, or z was updated. If the on-condition was met, the indicated action would be performed. Thus, a wide range of trace and/or debugging features can be specified. Also, a variety of special case monitoring of the program, which is non-debugging--but part of the desired processing--can be incorporated in the language.

The language provides the following data-handling statements (small single letters will be used to denote data collection names):

1) DELETE region;
2) INSERT region {BEFORE | AFTER} x (index-expression) [AND MAKE CURRENT];
3) REPLACE region BY region;
4) ADD region TO x [AND MAKE CURRENT];

where a region is defined as

collection-name (index-expression)
[TO collection-name (index-expression)]

and specifies a contiguous inclusive set of collection members (the collection names must be the same and the indices in ascending order). (The AND MAKE CURRENT option is explained below in Sec. IV.) The REPLACE and ADD statements are defined in terms of the INSERT and DELETE statements. REPLACE is equivalent to DELETE followed by INSERT BEFORE, and ADD is equivalent to INSERT AFTER the last member of the collection. The above statements are defined for all data representations allowed in the language; e.g., insertion into an array is defined as increasing by 1 the index of all members with indices greater than or equal to the value (i) of the inserted index, and the storing of the value of the inserted datum as the new i^{th} member of the array. (The problems concerning the maintenance of data collections and of pointers to members of these collections are discussed in Sec. V below.)

Two facilities are provided for sequencing through a data collection. The first is the FOR-clause which sequences through a data collection searching for all members which satisfy a specified condition and, as each one is found, causes a single statement or a group of statements to be executed. Thus, one cycle of an "iteration" is performed for each member of the subset of the data collection which satisfies the specified condition. This facility essentially combines a DO-statement (to sequence through the

data collection) with an IF-statement inside the range of the DO-statement to test the given condition.

The second sequencing facility: 1) sets up a search through a data collection, in the same way as a FOR-clause; 2) searches for the first member of the data collection which satisfies the given condition; and 3) causes a placemarker to point to that member. This placemarker can be used for processing the selected member. When the next member of the sequence is desired, it is explicitly requested, causing 1) the data collection to be searched for the next member which satisfies the previously specified condition, and 2) the setting of the placemarker to point to the newly selected member. This facility generates, upon request, the next member of a collection which satisfies a given condition. Hence, the facility is called a "generator," the statement which sets up the sequencing a "GENERATE-statement," and the placemarkers "generator-variables." The generator method of sequencing differs considerably from the FOR-clause method in that 1) it is not associated with a particular statement or group of statements, and 2) is not automatic but occurs only on request. The same code is not necessarily used to process all members in the sequence, and the same code can be used for different data-collection sequences. These generators can operate independently of each other, and more than one can be sequencing through a data collection simultaneously (if two or more generators are sequencing through a data collection, and the processing done in connection with one of them alters the data collection, the set of selected members of the other generator(s) may

also be altered, as the current state of the data set is used in the determination of the next member to be selected.)

The FOR-clause is used to control the execution of a single statement or group of statements for selected members of a collection. Its syntax is:

```
FOR {ALL| EACH| EVERY} collection-name
    [iterative-specification]
```

where an iterative-specification is defined as

```
[IN THE DOMAIN specification [, specification]...]
    [SUCH THAT (boolean-expression)]
```

and a specification has the form

```
expression-1 [TO expression-2][BY expression-3]
    [WHILE (expression-4)]
```

The precise semantics of a specification are given in the PL/1 specifications [1], but basically correspond to the following: a region is specified (by expressions 1 and 2) through which an index is incremented in steps of size expression 3; and the WHILE-clause specifies a condition which, if not met, causes the termination of that specification. Notice however that contrary to the DO-clause, the index is not specified; it is not needed and is supplied automatically by the system.

The FOR-clause is associated with a single statement by inserting the FOR-clause before the statement's

terminating semicolon (e.g., x = y FOR ALL x SUCH THAT (x < 5);), or with a group of statements by preceding the statements by:

FOR-clause DO;

and following them by

END;

The FOR-clause causes execution of the associated statement or group of statements for all members of the collection in the specified domain for which the boolean-expression is true. The iterative specification has the same interpretation as in PL/1. If no iterative specification is given, then all the members of the data collection, taken in ascending order, are used as the domain.

Generators are used to give the programmer explicit control of the sequencing through a data set. A new variable type--generator-variable--is introduced for this facility and is used to obtain the successive members of the collection. These variables allow more than one generator to be operating on a data collection at once, and also allow one generator variable to sequence through two or more different data collections during program execution (but only one at a time). The syntax for generator statements is:

```
GENERATE THROUGH collection-name USING
     generator-variable [AND MAKE CURRENT]
     [iterative-specification];
```

This statement sets up the sequencing implied by the iterative-specification as defined in the FOR-clause (or a sequence of the successive members of the collection taken in ascending order if the iterative-specification is not specified) and establishes the generator-variable as a synonym for the first member of the collection selected by the sequence. (The AND MAKE CURRENT option is explained below in Sec. IV.)

The programmer explicitly instigates iteration by execution of the GET NEXT statement:

[GET] NEXT (generator-variable) [{AND| BUT DO NOT} MAKE CURRENT]; [OTHERWISE statement]

This causes the generator-variable to become a synonym for the next member of the collection selected by the GENERATOR-statement. If no other members exist for the generator and an OTHERWISE-statement (which can be a block or group) is present, it is executed as an ON-unit.† If not, a new

†ON-units and conditions are fully explained in Ref. 1, pp. 79-84, but basically consist of the following: When such conditions as fixed point overflow, end of file, or END_GENERATOR occur, the program execution is interrupted. If an ON-unit has been specified to handle a condition which has occurred, then it is executed. This group of code can take corrective action and return to the program, print out an error statement, or terminate the program. If no ON-unit has been specified, the software system takes some default action (usually terminating the program). ON-units can be specified for any occurrence of a condition or only for those occurrences related to a specified set of variables. Thus the action taken can be dependent on which variable was related to the occurrence of the condition.

condition--the END_GENERATOR condition--is raised and can be handled by an appropriate ON-unit. The maintenance of the correspondence between a generator-variable and a collection member in a dynamic environment is explained in Sec. V below.

Search-expressions cause a data collection to be searched for a member which satisfies a condition. Either this member or its index is the value of the expression. The syntax of a search-expression, which can be used anywhere an expression can, is

```
[INDEX OF] numeric-specification collection-name
    [IN THE DOMAIN specification] SUCH THAT
    (boolean-expression)
```

where specification is defined as in the FOR-clause, and numeric-specification has the format:

```
{FIRST | SECOND | LAST | {decimal-integer |
    (expression)} {ST | ND | RD | TH}}
```

Examples:

```
10TH
21ST
(x+3)RD
```

The numeric-specification specifies a value i, and the value of the search expression is the i^{th} member (or the index of the i^{th} member if INDEX OF is specified) of

the data collection which satisfies the boolean-expression with the order of iteration specified by the iterative-specification.

The statement following a statement containing a search-expression can begin with the keyword OTHERWISE. This statement is executed as an ON-unit[†] if and only if a value can not be found for the search expression.

If the OTHERWISE option is not specified and the search-expression does not produce a value, a new condition, SEARCH_FAILURE, is raised and can be handled by an appropriate ON-unit.

Many times a search-expression is used only to find out whether or not a member of a collection exists which satisfies a certain condition. To make the intent of this use of the search-expression more apparent, the syntax of the IF-statement has been extended as follows:

```
IF {boolean-expression | THERE [{DOES | DO} NOT]
     {EXIST | EXISTS} {A | AN | number | (expression)}
     collection-name SUCH THAT (boolean-expression)}
     THEN statement [ELSE statement]
```

The semantics of the IF-statement remain unchanged; i.e., the statement following the THEN is executed if and only if the condition specified between the IF and the THEN is true; and the statement following the ELSE, if present, is executed if and only if the condition is false.

[†] For a discussion of ON-units and conditions, see Ref. 1, pp. 79-84.

Examples of the extended IF-statement are:

```
IF THERE EXISTS A y SUCH THAT (y < 10) THEN GO TO z;
IF THERE DO NOT EXIST 10 y SUCH THAT (y > 0)
     THEN RETURN(0); ELSE RETURN(1);
```

IV. HIERARCHICAL DATA REFERENCES

For data representations which are hierarchical, references to the elements can be fully specified by using the following form, where n is the level of the element being referenced:

$$\text{Collection-name}_{n}\ (\text{index}_{n})\ \text{OF collection-name}_{n-1}\ (\text{index}_{n-1})\ \text{OF}\ldots\text{OF collection-name}_{1}\ (\text{index}_{1})$$

For example, consider a list, x, of arrays, y, where the i^{th} member of the j^{th} array is referenced by

y (i) OF x (j)

This notation assures unambiguous data references but is: notationally burdensome; restricts our reuse of the variables used in higher level indices; and necessitates referencing through each level of the hierarchy. Instead, it would be convenient to reference data relative to some member of a collection. Once a user has decided to talk about the j^{th} member of the collection x, he would like to be able to reference the i^{th} member of collection y by writing

y (i)

This facility is achieved by introducing a new statement which allows the programmer to specify that a particular member of a collection is to be made a reference

point for further data specifications. This is called making the member current. One (and only one) member of each collection can be made current. The form of this statement is:

```
MAKE collection-name (index-expression) CURRENT;
```

The specified member of the collection is made current.

Each FOR-clause saves the current member of the data collection being iterated through, and each iteration of the loop makes the newly selected member current. Normal termination of the loop (the iteration has been completed) causes the saved member to be made current again. If the loop is terminated by a transfer (GO TO), the current member remains current and the identity of the previous current member is lost.

Whenever a reference is incompletely specified, the system will supply the necessary current member (or members) of the missing collection (or collections) to complete the specification of the reference. This is done in the following way:

1) If the highest level collection in the reference is not completely specified, the current member of the collection at the next higher level is used to complete this part of the specification. Thus, in the previous example, a reference of "y(i)" will cause the system to complete the specification by supplying the current member of collection x.

2) If there are any missing collections which are intermediate in level between the highest and lowest collection specified in the collection, the index of the current member of those collections is used as the index of those collections to complete the specification. For example, given

a five-dimensional array (x1, x2, x3, x4, x5), the reference "x4(2) OF x1(10)" would be completed by the system and become "x4(2) OF x3(CURRENT) OF x2(CURRENT) OF x1(10)." (CURRENT is a function whose value is the index of the current member of the collection in whose index expression the function reference appears.)

Notice that the above completed specification is different from the reference "x4(2)" (which would be completed by rule 2) above to "x4(2) of x3"), and the current member of x3 does not necessarily have to be a subpart of x1(10).

Any data reference may require none, one, or both of the above rules to be completed. For example, "x4(2) OF x2(5)" requires both rules to be completed as "x4(2) OF x3(CURRENT) of x2(5) of x1". By use of these mechanisms, data references can be specified relative to a base determined by the current member of a data collection. Generator-variables can also be used for relative data specifications by specifying the generator-variable as the relative base. For example, if "gen" were a generator-variable sequencing through data collection x, then "y(i) OF gen" would refer to the i^{th} member of whichever member of x "gen" was set to. This method of hierarchical references is similar to the use of "bugs" in L6 [2] and to the concept of "current" in APL [3].

Since generator-variables can be set to an individual member of a collection (e.g., "GENERATE THROUGH x USING gen IN THE DOMAIN 5;" causes "gen" to be set to the fifth member of the collection x), the full power of the bug concept of L6 is available. To make this use of generator-variables easier, the following statement has been introduced:

SET generator-variable TO collection-name (index-exp);

This causes the specified generator-variable to be set to the specified member of the named collection (any attempt to execute a GET NEXT statement for this generator variable will raise the END_GENERATOR condition).

To extend the use of the current concept to generators, options have been included which allow the programmer to specify whether the members generated should or should not be made current. In the GET NEXT-statement, he can specify whether the members generated through the use of that particular GET NEXT-statement should be made current or not. If neither option is specified, then the GENERATE-statement is used for the determination. If the AND MAKE CURRENT option is specified, the generator members will be made current; otherwise, they will not.

Similarly, the programmer can specify in an INSERT or ADD-statement that the inserted or added member is to be made current. (If a region is inserted or added and the AND MAKE CURRENT option is selected, the last member of the inserted or added region is made current.)

To extend the programming facilities and/or for notational convenience, the following built-in functions have been defined--where a collection-member-expression can be a collection member reference, a search-expression, a generator-variable, or a function reference which returns a collection member as its value.

NEXT (collection-member-expression),
PREVIOUS (collection-member-expression), and
INDEX (collection-member-expression)--the value of the function is the next or previous member, respectively, of the collection specified in the expression from the specified member or the index of the specified member of the collection. An OTHERWISE-statement can follow a statement including these functions. This statement is executed if and only if the desired member does not exist. If an OTHERWISE-statement is not employed and the desired member does not exist, the NO_NEXT, NO_PREVIOUS, or NO_INDEX condition is, respectively, raised.

NUMBER (collection-name) returns the number of members in the specified data collection.

LAST and CURRENT return, respectively, the number of members in the specified collection and the index of the current member of the specified collection. These functions can only be used inside an index-expression, and the collection referenced is the one to which the index-expression applies.

NEW [(collection-name)] a new member of the specified collection is created; and, if initial values have been specified, is

initialized. This function can only be used in an ADD or INSERT-instructions; and if an operand is not specified, a new member of the collection being added to or inserted into is created. This function together with the ADD and INSERT-statement constitutes the collection-building capability of the language.

V. MAINTAINING POINTERS IN A DYNAMIC ENVIRONMENT

There are two types of pointers in the Dataless Programming Language: the generator-variables, and the current member of each data collection. Both types point at members of data collections, and these pointers must be maintained while the data collections are altered.

There are two types of alterations which can affect these pointers. The first is an insertion or deletion from the data collection, causing a change in the location of the members of that collection (an insertion or deletion from an array has this effect). The second is the deletion of a member being pointed to. These pointers can be maintained by keeping a list for each data collection of all pointers which point to members in that data collection. While moving or deleting a member (and its submembers), the system can check for pointers which refer to the affected members (and submembers), and take appropriate action. This action for movement is merely the repositioning of the appropriate pointers so that they point to the member's new location. In the case of deletion, the action taken is more complex. First, the affected pointers are set to a state called "undefined" (all pointers--generator-variables and the current member of all data collections--are initially set to this state), which will cause any subsequent attempt to reference the deleted member to raise the UNDEFINED_POINTER condition that can be handled by an appropriate ON-unit. Secondly, internal pointers to the next and previous members of the collection (if they exist) are associated with the

undefined pointer so that they can subsequently be used to move to these members. (If these members do not exist, an attempt to use these internal pointers will cause the NO_NEXT or NO_PREVIOUS condition to be raised.) These internal pointers are also maintained through any subsequent alterations of the data collection.

VI. DATA DEFINITION

The individual members of a collection that are not themselves collections can be any of the forms defined in PL/1 for data elements. Arrays, lists, and rings are defined by inserting the appropriate keyword (ARRAY, LIST, DOUBLE LIST, RING, or DOUBLE RING) after the collection-name in the PL/1 definition. Structures are defined as in PL/1. Thus an array of lists of structures would be defined as:

```
DECLARE
     1   department (20) ARRAY,
         2   people LIST,
             3   name CHARACTER (20) VARYING,
             3   man-number BINARY FIXED,
             3   projects LIST  CHARACTER (10);
```

For those data representations that cannot be defined using the system-defined representations, the definition is supplied by providing programs which handle the necessary manipulations on the data representation. The form of this definition is:

```
collection-name  ACCESSED BY program 1;
                 UPDATE USING program 2;
                 INSERT USING program 3;
                 DELETE USING program 4;
```

The programs used in the data definition can contain any of the features provided by the system. In addition, they can include, where applicable, references to the addresses of an operand and the contents of an address. Two system functions are provided for this purpose:

ADDRESS (variable)

and

VALUE (location, form)

where location specifies where the desired value is, and form specifies the transformation to be used to extract a value from this location. Typical forms are (with the IBM 360's in mind):

```
BF   =  sign + 31 numeric bits;
HBF  =  sign + 15 numeric bits;
QBF  =  sign + 7 numeric bits;
QL   =         8 numeric bits;
QB   =         8 logical bits.
```

These same forms could be specified in the left-hand side of an assignment statement to indicate the form in which a value should be stored:

```
y = VALUE (ADDRESS (x) + 5,HBF) +3;
```

the 16 bits at location (address (x) + 5) are treated as a sign + 15 numeric bits and are added to the number 3 and the result is stored in variable y

```
x-2 IN FORM BF = ADDRESS (y);
```

the location (address (y)) is stored in location (x-2) in form BF

When the system passes collections as parameters to a subroutine, it also passes the necessary data-handling routines (either the standard system routine or the user's routine) so that the subroutine can operate on any allowable data collection. Such routines are data-representation independent, and libraries of them should provide a flexible programming environment. One major problem not satisfactorily solved is finding a method that allows the user to change the form of the variables of a collection (such as from character strings to floating point numbers) and still use the same subroutine to process both forms. Present plans for handling this have the user:

1) declare those parameters which can be of different forms to be 'FREE',
2) perform form checking on all operations involving these FREE parameters,
3) utilize the correct routine to handle them properly.

The user would also have to specify in the calling program which parameters to the subroutine were FREE so that their forms could also be passed.

VII. FURTHER LANGUAGE FEATURES

In defining the Dataless Programming Language, several other features seemed desirable that were not currently available in algebraic languages in general--and specifically not in PL/1. These features are included here[†] (although they do not directly relate to the main goal of separating data description from program description):

1) a source-level execute command which causes the named statement or group of statements to be executed, and allows the passing of parameters for this execution;

2) a compare statement which stores the result of a comparison in a cell associated with the label of the compare statement, and which allows this cell to be subsequently interrogated and, if desired, modified;

3) the CASE function of LISP 1.5, which permits an expression to determine which of a series of expressions should be used as the value of the function.

[†] These and other features will be fully described in a related publication.

VIII. IMPLEMENTATION

Present plans call for an implementation of the Dataless Programming Language, in a restricted form, through an editing program utilizing Leavenworth's Syntax and Function Macros [4] (which puts the program into standard PL/1) plus a set of run-time routines. This method will impose certain restrictions (e.g., lists and rings must be implemented as arrays because the list processing capability of PL/1 is not available) and will be detrimental to program efficiency. But it will enable running programs written in Dataless Programming at a relatively small cost. With experience, the language can be improved, and plans made for a full implementation. Although most of the capabilities of Dataless Programming can be achieved by placing an editor between the programmer and the PL/1 compiler, this in no way alters the fact that Dataless Programming provides a radically different view of the programming environment than does PL/1.

IX. EXPECTED ADVANTAGES AND DIFFICULTIES

Programming should be simplified because it can be constructed top-down in terms of the logical processing required. The problem of data representation can be left until this programming has been completed; thus, a more rational decision can be made concerning an optimal representation. Because of this separation, the programmer should be able to think through his problem better.

The data-handling features incorporated to handle all data homogeneously put the language into a more canonical form, make it more mnemonic, improve its readability, and increase its self-documentation. This improved readability will undoubtedly help reduce the number of programming errors in the original program, and make finding and correcting the remaining errors easier. All this, in some sense, increases the "high-levelness" of the language.

Since Dataless Programming takes care of passing the necessary data-handling routines to subroutines or functions, libraries of routines which are data-representation independent (within the basic ordered-list restriction of the language) can be created for use in a general programming environment.

Also, since data representations can be easily changed, it is possible to determine experimentally which is best for the given program--i.e., with reference to the Dataless Programming system. Even so, this would provide one objective measure for different data representations from which criteria might be generalized and developed for the choice of representations.

The main difficulty with the system is the level of efficiency. Inefficiencies come from two sources: 1) the separation of the data-handling routines from the program and from the homogeneous manner in which all data is handled; 2) since data is treated homogeneously, the programmer cannot take advantage of the special properties of one representation in writing a program without having already decided what representation will be chosen. Hopefully, once the program is debugged there will be ways to remedy some of the inefficiencies.

There are several other difficulties and shortcomings of Dataless Programming, including the incomplete separation of data description from program description, and the lack of a method of extending the allowable set of data representations beyond ordered lists. Also, the language is now a hodgepodge of all the facilities considered desirable for either their processing capabilities or convenience. As such, it is not a smooth, polished, well-integrated system.

The essential considerations, however, are how easy and natural is it to learn and program in the Dataless Programming Language, and how much more powerful it is than existing languages.

A few words about the similarities and differences between Dataless Programming and COBOL: Both were designed to produce readable programs, both have a variety of data-handling capabilities, both use structures as a data representation, and both have a separate data definition section. The differences are, however, more significant. While COBOL is directed toward the description and processing of files of external data, Dataless Programming is directed

toward the description and processing of internal data. COBOL is designed for efficient business oriented applications, while Dataless Programming is designed for flexible scientific oriented applications. Finally, while COBOL provides facilities for processing only one type of collection (arrays), Dataless Programming provides--and is built around--a common set of facilities for processing any data collection which can be represented as an ordered set.

X. DATALESS PROGRAMMING EXAMPLES

In the following examples, the problem statement is followed by the uncommented program and by the set of comments pertaining to the program. The numbers at the start of each line of the programs are not part of the programs, but are used to associate comments with statements. Notice that almost all comments explain the functioning of the language statements; little commentary is needed to explain the processing--i.e., once the functioning of the various language statements has been mastered, the program itself becomes mnemonic because the intent of the programmer is usually apparent from the language statements themselves.

The first example is a problem which appeared in the article, "APL--A Language for Associative Data Handling in PL/1" [3]. Since APL and Dataless Programming share many similar features, the problem affords a means of comparison. This example was also chosen because it illustrates many of the facilities of Dataless Programming, including the use of functions as if they were data, the automatic passing of the current member of a collection, the use of a generator variable as a bug, and the use of search expressions (with the OTHERWISE option), iteration statements, and the collection-building facility.

The problem is to compute the starting and ending dates for a set of jobs. The input data consists of a set of entries, one for each job. Each entry consists of the job's jobname and its duration, followed by a list of all jobs that must be completed before this particular one can be started. It is assumed that these data are in a form suitable for PL/1 list-directed input.

```
1.            ON ENDFILE GO TO compute;
2.  read:     GET LIST (ident);
3.  locate:   MAKE 1ST job SUCH THAT ident = jobname CURRENT;
4.            OTHERWISE DO;
5.                ADD NEW TO job AND MAKE CURRENT;
6.                jobname = ident;
7.            END;
8.            GET LIST (duration);
9.            SET present_job TO job;
10. read_predecessor:  GET LIST (ident);
11.       IF ident = '' THEN GO TO read;
12.       EXECUTE locate;
13.       ADD INDEX (job) TO predecessor OF present_job;
14.       GO TO read_predecessor;
15. compute:
16.       FOR EACH job SUCH THAT (all_predecessors_finish_known
              ∈ finish_date_is_unknown) DO;
17.        start_date = max_predecessor_finish + 1;
18.        finish_date = start_date + duration;
19.        GO TO compute;
20.        END;
21.        IF THERE EXISTS A job SUCH THAT finish_date_is_
               unknown THEN
22.           PRINT LIST ('incorrect data');
23.       ELSE
24.           PRINT LIST ('satisfactory run');
25.       DECLARE
26.           ALL_predecessors_finish_known BIT (1)
27.               ACCESSED BY
```

```
28.                     IF THERE EXISTS A predecessor SUCH THAT
                            (finish_date_is_unknown (predecessor))
                            THEN RETURN ('0'B);
29.                     ELSE RETURN ('1'B);,
30.           finish_date_is_unknown BIT (1)
31.           ACCESSED BY
                  IF finish_date = 0 THEN RETURN ('1'B);
33.               ELSE RETURN ('0'B);,
34.   max_predecessor_finish BINARY FIXED
35.       ACCESSED BY
                  DO;
37.               x = 0;
38.               x = MAX (x, finish_date (predecessor)) FOR
                      EACH predecessor;
39.       END;,
40.   1 job LIST,
41.       2 jobname CHARACTER (20),
42.       2 start_date BINARY FIXED INITIAL (0),
43.       2 finish_date BINARY FIXED INITIAL (0),
44.       2 duration BINARY FIXED,
45.       2 predecessor LIST BINARY FIXED;
46.   DECLARE
47.       ident CHARACTER (20),
48.       present_job GENERATOR,
49.       x BINARY FIXED;
```

COMMENTS

1 Sets up an ON-unit that will cause transfer of control to the statement labeled "compute" when the end of the input file has been reached.

2 A PL/1 input statement that causes the next value in the input file to be assigned to the named variable ident.

3 Find and make current the job that has a jobname equal to the value of ident.

4-7 An OTHERWISE-clause. These statements are executed if and only if the search in statement 3 failed, i.e., if there did not already exist a job that had a jobname equal to the value of ident.

5 Creates and initializes a new job, adds it to the list of jobs and makes it current.

6 The jobname of the newly-created job is set equal to the value of ident.

8 The next value in the input file is assigned to the named variable duration (the reference "duration" is incomplete and is completed by using the current member of data collection job, hence the duration of the current job is the variable referred to).

9 The generator-variable present_job is set to refer to the current job.

10 The next value in the input file is assigned to the variable ident. This value is the name of a predecessor of the current job.

11 Test for the end of the list of predecessors. If the value of ident is NULL, the end of the list has been reached and control is transferred to the statement labeled read to process the entry of the next job.

12 This is an additional feature that causes the specified statement (the statement labeled locate together with its associated OTHERWISE-clause, i.e., statements 3-7) to be executed, after which execution continues with the next statement (statement 13). This group of executed statements (3-7) makes current the job whose name is the value of ident, and if this job does not already exist, creates and initializes it.

13 The index of the job which is a predecessor of the job referred to by present_job is added to the collection of such predecessors.

14 Get another predecessor.

15 The input phase has been completed; calculate the start and finish dates for the jobs.

16 Sequence through the collection of jobs, selecting those which satisfy the given condition. (The condition is specified in terms of two data items defined in the data declaration.) Notice how the details of finding the next job to be processed are removed from the program and can be stated in terms of the logical requirements of the process, i.e., to process a job the finish data of all its predecessors must be known and its own finish date must be unknown.

17 Calculate the start date of the current job (a job selected by statement 16 above) by use of a data item defined in the data declaration. Again, notice how the processing has been expressed in terms of the logical processing required.

18 The finish date of the current job is set equal to the start date of this job plus its duration.

19 Begin the search again.

21-24 Test for the satisfactory completion of the program and print conclusion. If any jobs have an unknown finish date, then the data used in the program must be incorrect.

25-29 Definition of all_predecessors_finish_known as a function.

28 The collection of predecessors of the current job is searched for one for which the finish date of the job whose index is the value of predecessor is unknown. If such a predecessor exists the finish date of all the predecessors of the current job is not known and the bit string '0'B is return.

30-33 Definition of finish_date_is_unknown as a function.

34-39 Definition of max_predecessor_finish as a function.

40-45 Definition of the structured collection named job.

46-49 Definition of the remaining variables.

The second example is a program that does a binary search through a collection of character strings in ascending order for a match with an input string. If a match is found, the index of the matching collection member is returned. Otherwise, the input string is added to the collection in the proper place and the index of this new member is returned. This routine can be used to build the ordered collection. Besides the restriction that the collection must be an ascendingly ordered collection of strings, the representation of the collection is free. The routine is able to handle different representations because the system passes the necessary data handling functions along with the collection.

```
1.  Binary_search:  PROCEDURE (input_symbol, collection)
        BINARY FIXED;
2.          IF NUMBER (collection) = 0 THEN DO;
3.              ADD input_symbol TO collection;
4.              RETURN (1);
5.          END;
6.          increment = initial_increment;
7.          MAKE collection (2 * increment) CURRENT;
8.  test:   COMPARE input_symbol AND collection;
9.          IF = IN test THEN RETURN (INDEX (collection));
10.         IF < IN test THEN MAKE collection (CURRENT-
                increment) CURRENT;
11.         ELSE MAKE collection (MIN (CURRENT + increment,
                LAST)) CURRENT;
12.         increment = increment /2;
13.         IF increment > = 1 THEN GO TO test;
14. add-new-item:
```

```
15.         IF < IN test THEN INSERT input_symbol BEFORE
                collection AND MAKE CURRENT;
16.         ELSE INSERT input_symbol AFTER collection
                AND MAKE CURRENT;
17.         RETURN (INDEX (collection));
18.         DECLARE
19.         initial_increment BINARY FIXED ACCESSED BY DO;
20.             increment = 2;
21.             DO WHILE (increment < NUMBER (collection));
22.                 increment = increment * 2;
23.             END;
24.         RETURN (increment/2);
25.         END;,
26.         collection CHARACTER (*) COLLECTION,
27.         increment BINARY FIXED,
28.         input_symbol CHARACTER (*);
29.  END binary_search;
```

COMMENTS

1 The data handling routines for the collection named "collection" are passed by the system to this routine.

2-5 If there are no members in the collection, the input symbol is added to the collection and the index of this new member is returned.

6 The value of increment is set to the value of initial_increment, which is a data item defined in the data declaration. Notice again how the details of the processing necessary to find this value are removed from the program.

7 The first guess of the binary search procedure is made current.

8 This statement is another additional feature. It compares the values of input_symbol and the current member of the collection, and places the results of this comparison in a cell that can be interrogated as shown in lines 9, 10, and 15.

9 If the result of the comparison in the compare statement labeled 'test' was equality (a match was found), the boolean-expression in this IF statement is satisfied and the index of the current member is returned.

10 If the result of the comparison in the compare statement labeled 'test' was that the first operand was less than the second operand, the next guess is the member whose index is (CURRENT-increment), and this member is made current.

11 Otherwise, the next guess is the member whose index is (CURRENT + increment); but a check is made not to go beyond the end of the collection, and this member is made current.

12 The size of the increment is cut in half.

13 Keep searching if the size of the increment is non-zero.

14 The search has been completed and the item has not been found; add it to the collection and return its index.

15 If the result of the last comparison in statement 'test' was that the input_symbol was less than the current member of collection, insert the input_symbol into the collection before the current member and make the inserted member current.

16 Otherwise insert the input_symbol after the current member of the collection and make the inserted member current.

17 Return the index of the inserted member (which is now current).

18-29 Declaration of the data items of the program.

19-25 Declaration of initial_increment as a function.

REFERENCES

1. IBM Operating Systems /360 PL/1: Language Specifications, IBM Form C28-6571-3, IBM Corporation, White Plains, New York, 1966.

2. Knowlton, K. C., "A Programmer's Description of L^6," Communications of the ACM, Vol. 9, August 1966.

3. Dodd, George G., "APL--A Language for Associative Data Handling in PL/1," AFIPS, Vol. 29, Proc. FJCC, Spartan Books, Washington, D.C., 1966.

4. Leavenworth, B. M., "Syntax Macros and Extended Translation," Communications of the ACM, Vol. 9, No. 11, November 1966.

5. Newell, A., Information Processing Language-V Manual, Prentice-Hall, Englewood Cliffs, New Jersey, 1961.

6. Abrahms, P., et al., "The LISP 2 Programming Language and System," AFIPS, Vol. 29, Proc. FJCC, Spartan Books, Washington, D.C., 1966.

7. Shaw, J. C., JOSS: A Designer's View of an Experimental On-Line Computing System, The RAND Corporation, P-2922, August 1964.

8. "Revised Report on the Algorithmic Language ALGOL-60," Communications of the ACM, Vol. 6, No. 1, 1963, pp. 1-17.