A MATHEMATICAL PROGRAMMING GENERATOR SYSTEM

Edward A. Stohr

October 1985

Center for Research on Information Systems
Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #96

GBA #85-41

An earlier version of this paper is contained in Stohr, Edward A., **'A Mathematical Programming System in APL'**, Discussion paper Number 348, The Center for Mathematical Studies in Economics and Mathematical Science, Northwestern University, Evanston, Illinois, 1979.

The revisions were carried out as part of a jointly-defined research project on expert systems with the IBM Corporation.

ABSTRACT

This paper describes a mathematical programming generator that interprets problem statements written in the algebraic notation found in journal articles and text-books and outputs statements in the 'MPS format' used by IBM's MPSX mathematical programming system. The system has been implemented in the APL programming language. Although originally designed for stand-alone use, it is currently being used as a component in an expert system that will help users formulate large linear programming models. The paper describes the syntax of the problem definition language and gives some illustrative examples. There are several unique features. First, the user can define objective function, constraint and right-hand-side coefficients as APL expressions. This leads to concise problem statements and also reduces data storage and processing requirements. Second, the system supports an integrated data base query language. Finally, there are a number of aids for model maintenance and sensitivity analysis. The last section of the paper describes the use of MPGEN in the expert system context.

A MATHEMATICAL PROGRAMMING GENERATOR SYSTEM

1. INTRODUCTION

The manual generation of the data for mathematical programming algorithms is a tedious task which lends itself well to automation. In fact, for the large linear programs often found in practice (which may have thousands of constraints and variables), it is hard to imagine that the required data could be generated by hand within a reasonable amount of time and with reasonable accuracy. To assist in this task a number of 'tableau generators' have been developed that can interpret the problem statement and produce the data required by the mathematical program, [4], [8], [12]. These are often combined with special report writers for the analysis and display of data and results, [7].

The Mathematical Program Generator System (MPGEN) described in this paper accepts a problem statement written in the algebraic notation found in journal articles and textbooks. This is interpreted and the data for the tableau is generated as 'data triples' in the form (i, j, v) where i is the row index, j the column index and v the associated tableau value. The conventions adopted in generating the data triples are those used by IBM's MPSX mathematical programming system, [9]. The generated data triples can be used by mathematical programming algorithms coded in APL, [6], or output to an external problem solver. In the latter case, the system also supplies correctly formatted commands to activate the relevant algorithms and report options.

MPGEN has been implemented on two different computers at New York University:

(1) A DEC-20 computer where it supplies inputs to the LINDO mathematical programming system, [15]

(2) An IBM 4341 where it interfaces with the MPSX mathematical programming package, [9].

The original motivation for MPGEN was to provide an input format that would allow a direct transcription from the mathematical statement of the problem to its tableau representation. In fact, the problem statement can, almost literally, be an abstract from the journal article, textbook or model builder's notebook in which the problem structure was originally defined. It is not necessary to know anything about APL in order to use MPGEN. However, a knowledge of APL syntax allows one to take advantage of many powerful features. The goal of the LPFORM expert system now being developed at NYU is to provide a front-end to MPGEN that will eliminate the need for users to understand any form of mathematical notation. Figure 1 shows the overall architecture of LPFORM. An important objective

of this paper is to explain the role of MPGEN in this expert system context.

Figure 1
Design of LPFORM System

The expert formulator (LPFORM) allows users to define the structure of the LP problem using a graphical interface, [11]. Special icons are used to represent real world objects such as physical locations, flows, resources, inventories and activities. Users can decompose their problem into successive layers of detail and utilize problem templates (e.g. transportation, product-mix and blending LPs) as components in a larger problem. The knowledge base in LPFORM uses all of these different kinds of information to perform a number of consistency checks and to generate an algebraic formulation suitable for input to MPGEN.

The sigma-notation syntax used by MPGEN provides a row-oriented viewpoint of a linear program. By way of contrast, OMNI, [8], and several other popular 'tableau-generators', have an 'activity' or column-oriented approach in which the formulation is organized in a column-by-column fashion. Since activities generally have only a few non-zero coefficients this can sometimes simplify the formulation task. However, column-oriented problem statements are longer than row-oriented ones and may be less easy to read and debug.

The idea of using algebraic notation appears also, for example, in [5] and [12]. One innovation here is that the variable coefficients, right-hand-side constants and variable indices can be specified as expressions in the APL host language. As the problem statement is interpreted, values are substituted for these expressions using the APL 'Execute' function, ⍎ (which executes APL instructions written as character strings). In effect, each value in the initial tableau for the mathematical program can be expressed as a complicated, real-valued function of the underlying data or as a logical function of its row and column position. Using APL in this way helps to provide a concise problem statement and also reduces the need to preprocess the data. As a result data storage requirements can be greatly reduced and (perhaps more importantly) less time and effort is involved in developing the initial model and in performing sensitivity analyses.

A second innovation in MPGEN is a database query language (similar to IBM's SQL, [1]) and data manipulation functions that assist in the computation of the data required by mathematical programs. The database interface allows the system to retrieve

and process the values of LP data coefficients at the time the problem is run.

This paper focuses on the problem definition language and especially on features of the implementation that help users form concise and readable problem statements. These same features simplify the task of building the artificial 'expert' in LPFORM. The syntax of the language is illustrated in Section 2 and described more fully in Section 3. Section 4 provides an example that uses the database query language. The way in which the APL language can help form concise problem statements is described in Section 5. Section 6 describes the user's interaction when running the system and outlines other features including the provisions made for data entry and display, revision of problem statements and storage of the output of the algorithms. Finally, Section 7 describes the interface with the LPFORM expert system.

2. ILLUSTRATIVE EXAMPLES

We first illustrate the use of the MPGEN system in the solution of a small linear programming problem:

$$\text{Minimize} \quad 3X + 6Y$$

$$\text{subject to:} \quad 4X + 2Y \geq 10$$
$$2X + 5Y \geq 13$$

This problem can be defined in MPGEN using the system editor as follows:

```
*SAMPLE 1 - EXTENSIVE ALGEBRAIC FORM
*
*VAR= X(I), I IN 1 THRU 2
*
MINIMIZE
  3X(1) + 6X(2)
*
  4X(1) + 2X(2) ≥ 10
*
  2X(1) + 5X(2) ≥ 13
*
```

Figure 2
Algebraic Format - Extensive Form

This is an algebraic form similar to that used by LINDO and a number of other linear programming generators. In Figure 2, statements beginning with an asterisk are COMMENTs. The VAR= statement is a VARIABLE DECLARATION which is used by the system to assign columns to variables. The THRU function is used to indicate that the index set for X is 1, 2. Obviously, the above problem definition is only useful for the particular problem data

shown. More generally, the standard form of a linear programming problem:

Minimize C_jX_j

subject to: $A_{ij}X_j \geq B_i, \quad 1 \leq i \leq M$

$X_j \geq 0$

can be defined in MPGEN using 'sigma' notation as follows:

```
*SAMPLE 2 - SIGMA NOTATION
*
DATA= M,N,A(MxN),B(M),C(N)
*
VAR= X(J), J IN 1 THRU N
*
MAXIMIZE
S C[J]X(J)
J IN 1 THRU N
*
FOR I IN 1 THRU M
S A[I;J]X(J) ≤ B[I]
J IN 1 THRU N
*
```

Figure 3
Standard Form of an LP in 'Sigma' Notation Format

The 'standard' form of an LP in Figure 3 obviously applies to any linear program; it is only necessary to define the appropriate cost and right-hand-side (RHS) coefficient vectors and the constraint coefficient matrix, A. Thus the algebraic notation has two great advantages: (1) it is a very compact notation which corresponds almost exactly to the format used in the statements of operations research models, (2) the problem definition is general in the sense that it is independent of the dimensions of the problem (number of variables and constraints involved).

The disadvantage of the standard format for a linear program is that it ignores any special structure which might apply to a particular class of problem. Thus, the user is required to construct the complete A matrix which can be a laborious and error-prone task. This also requires the input and storage of unnecessary data since, in practical problems, the matrix is generally quite sparse, As illustrated in the following sections, MPGEN allows each group of constraints to be defined separately allowing the matrix, A, to be built implicitly from its component parts thereby eliminating the need to build a massive tableau.

3. THE PROBLEM DEFINITION LANGUAGE

Before proceeding to some more comprehensive examples, we outline the rules for defining a linear programming problem.

The MPGEN system interprets the problem statement in one pass. The order in which the user must define the problem statement is indicated below:

(1) One or more DATA DECLARATION lines (optional)
(2) One VARIABLE DECLARATION for each decision variable in the problem (required)
(3) An OBJECTIVE GROUP (optional)
(4) One or more CONSTRAINT GROUPs

COMMENT and EXEC statements (see below) can be interspersed freely in the text of the problem statement. Comment lines start with an '*' in column 1.

Data Declaration Lines

The DATA= line in Figure 3 is a Data Declaration statement. It is used by the system to check that the required data (variables M, N, A, B, C) are present in the APL workspace and that the variables (A, B, C) have the indicated dimensions (i.e. A is an MxN matrix and B and C are M- and N-dimensional vectors). If either of these conditions is false, the system will ask the user either to input the required data or to halt the problem interpretation. Execution of this statement for a new problem also stores the values of all declared variables in a random access file. When the problem is run a second time these values are automatically restored to the workspace.

Variable Declaration Lines

Variable Declaration lines are used to identify the decision variables and to specify the columns associated with these variables in the tableau. In general, there must be one such line for each decision variable in the problem statement. A Variable Declaration line consists of the keyword 'VAR=' followed by the variable name complete with (dummy) index variables enclosed by parentheses. This is followed by one INDEX TERM for each index variable with a comma separating each such term. An Index Term has the form:

Index-variable-name IN Set-expression

The SET-EXPRESSION may be any APL expression that returns a positive integer scalar or vector result. This specifies the full range of values that the index will take on in the problem statement. In Figure 3 the Index Term is J IN 1 THRU N indicating that J takes on the values 1, 2, .., N. If more than one Index Term is present in a Variable Declaration Line the order in which they appear determines the order of the decision variable columns in the tableau with the right-most decision variable index varying

the fastest.

Objective and Constraint Groups

A GROUP consists of one or more lines of the problem statement. An OBJECTIVE GROUP consists of a line containing the word MAXIMIZE (or MINIMIZE), the OBJECTIVE DEFINITION LINE, and, (if the latter contains one or more summation symbols) the associated SUMMATION INDEX LINE.

A CONSTRAINT GROUP consists of one or more FOR INDEX LINEs, the CONSTRAINT DEFINITION LINE, and, if required, a SUMMATION INDEX LINE. A Group and its associated data triples form a basic unit of data in the MPGEN system. All Groups must be preceded and followed by one or more Comment lines. The user is free to write any desired descriptive material in the comment lines.

Objective and Constraint Definition Lines

An 'S' followed by a blank in an Objective or Constraint Definition line represents the algebraic symbol, ' '. The range for the summation is indicated in the following Summation Index Line. In Figure 3 we have: J IN 1 THRU N, meaning that J takes values in the SET 1, 2,.., N. The rows in the tableau for which a Constraint Definition is defined are given by one or more For Index Lines (see below).

Objective and Constraint Definition lines differ only in that the latter contain one of the relational operators $\leq$, =, or $\geq$, together with a RHS COEFFICIENT EXPRESSION. The Objective and Constraint Definitions contain one or more VARIABLE TERMS separated by '+' or '-' operators. A Variable Term has the following form:

```
[One or more    [Variable Coefficient    Varname ({Variable Index
Summation        Expression]                        Expression})
Symbols]
```

where the square brackets indicate optional components and the braces represent repetition. For example, the second constraint in Figure 2 has two Variable Terms (X(1) and 2X(2)), while the Constraint Definition in Figure 3 has only one (S C[J]X(J)). Note that the summation symbols 'S' apply to only one variable; if more than one decision variable appears in an Objective or Constraint Definition line, then each must have its own summation symbols and appear in a separate Variable Term. The 'Varname' in the definition of a variable term represents a user-chosen name for a linear programming decision variable Note that the indices for the decision variables in the above were enclosed in parentheses in the problem statement--X(1), X(2), X(3) in Figure 2 and X(J) in Figure 3. The MPGEN System uses the names in the Variable Declarations plus the parentheses to recognize decision variables when interpreting Constraint and Objective Definitions.

In the current implementation, a decision variable can be indexed by up to five VARIABLE INDEX EXPRESSIONs separated by commas. Each such expression can be a constant, a variable appearing in a Summation Index or For Index Line, or any non-parenthesized APL expression that returns a scalar result. During interpretation, the MPGEN system evaluates each Variable Index expression separately using the APL Execute function.

The VARIABLE COEFFICIENT EXPRESSION component of a Variable Term may be any valid APL expression that returns a scalar result. Examples are constants, variables and the complex expressions containing inner product operations illustrated in Section 6. Similar remarks apply to the RHS Coefficient Expressions appearing in constraint definition lines.

Summation Index Lines

If an objective or constraint definition line contains one or more summation signs, it must be followed by a Summation Index Line in which the corresponding index variables and the values they are to assume are defined by one or more Index Terms separated by commas. As above, the Set-Expression in the Index Term may be any APL expression which returns a positive integer scalar or vector result, e.g. 1 THRU N, in Figure 3. The result of the Set Expression defines the values taken on by the INDEX variable during the summation. Examples of Summation Index Line's with more than one Index Term are shown in Figure 6. There must be the same number of Index Terms as there are summation signs in the preceding line. The correspondence between summation signs and Index Terms is obtained from the order (from left-to-right) in which the latter appear in the Summation Index Line.

The indices in the Index Terms corresponding to a Variable Term in the preceding Objective or Constraint Expression are executed in oedometer order with the right most index varying the fastest. Note that the desired result of a Set Expression may depend on the value of a previously defined index. Thus, in the example in Section 5, we have K IN K, I IN IP[K;] where K is an APL vector of index values for K and IP is an APL matrix in which the rth row contains the index values for I when K=r. Since trailing zeroes in an index vector are ignored by MPGEN, it is possible for the number of values assumed by the I index to vary with the value of K. An alternative way to define an index, I, as a function of another index, K, is to use the notation I{K} and to store the associated values of I in APL vectors, I1, I2, ... For example, we could include the EXEC statements:

```
E I1 <- 1, 3, 5
E I2 <- 2, 4
```

at the beginning of the problem statement. Here, '<-', is the APL assignment function. Subsequent execution of the index expression, I IN I{K}, will assign the index values 2 and 4 to I when K=2.

For Index Lines

The rows in the tableau for which a Constraint Definition applies are defined by one or more preceding For Index Lines. These consist of the word 'FOR' followed by an Index Term as defined above. In Figure 3 there is one such statement specifying M constraint rows (I lies in the set 1, 2,..., M.) If there is more than one For Index Line the indices are evaluated in oedometer order with the index for the last line varying fastest. Again, the Set Expressions can be dependent on previously defined indices.

EXEC Statements

In addition to the statements which define the mathematical program itself the user may insert other commands in the problem statement by the use of the MPGEN EXEC statement. This has the form:

E valid APL expression.

An example of the use of the EXEC statement to assign values to data variables was given above. Other uses are to open, read and close files, to process the problem data and to erase data variables that are no longer required.

General

The data variables and constants that define a particular instance of the problem are imbedded in the Set Expressions, Variable Index Expressions and Variable and RHS Coefficient Expressions. With the exception of certain reserved names, the user may employ any valid APL variable names for the Decision Variables, Index Variables and APL data variables.

Any statement can be continued if necessary on a succeeding line prefaced by a colon as shown in Figure 4 below.

MPGEN can automatically recognize variables with upper and/or lower bounds and will flag them on the MPS problem statement to allow the solver to utilize a more efficient variant of the simplex method.

The problem data is contained in the Variable and RHS Coefficient expressions while the structure is represented by the Variable Index Expressions and the Set Expressions in the Variable Declaration, For Index and Summation Index Lines. The fact that all of these items are expressed using APL statements (including function calls where necessary) means that the problem statement is 'bound' to its data only during the interpretation process. This adds a new dimension of power and flexibility by allowing the user:

(1) to reduce data storage requirements

(2) to eliminate the preprocessing steps necessary to compute the Cj and Aij coefficients required by the mathematical programming algorithm.

(3) to perform sensitivity analyses by directly modifying the 'natural', disaggregated, unprocessed data elements of the real world problem.

The danger in allowing this freedom is that error detection may become more difficult because MPGEN does not check the syntax of the Set, Variable Index and Coefficient Expressions. This is done by the APL processor during the evaluation. This potential drawback is mitigated, if not eliminated, however by (1) 'trapping' any such error and providing an error message which displays the expression where the error occurred, and (2) by checking the presence and dimensions of all required data when the Data Declaration statement is parsed.

4. EXAMPLE OF DATABASE USAGE

The retrieval power of a modern relational database query language (such as IBM's SQL, [1]) can greatly facilitate the specification and maintenance of LP models. Such languages allow a concise and readable specification which is advantageous both for the initial development and subsequent documentation of LP models. Further, they allow the logical statement of the problem to be independent of particular values in the database and thus prevent the model from becoming out of date. The example in Figure 4 shows the use of the MPGEN query facility for a small problem. The example is based on the following relational schema:

```
Profits(Prodname, Factory, Profit)
Production(Prodname, Factory, Prodamt)
Resources(Prodname, Factory, Resname, Resamt)
Reslimits(Resname, Factory, Reslimit)
```

Here, Profits, Production, Resources and Reslimits are database relations (files) containing information on marginal profit contributions, amount of each product produced for the last 12 months, the resources used during the last 12 months in producing these products, and the limits on the available resources for the next time period. The problem is to determine the optimal product mix for the next time period.

The GET database function is a shorthand for 'SELECT ALL FOR'. It retrieves all columns for rows that satisfy the logical qualification FROM the indicated stored relation. The result is a relation with the same columns but fewer rows (in this case only the rows for the northern factory). The TABLE function transforms this relation into a numeric array with dimensions depending on the number of items in its left argument. The latter contains a list of column names from the original relation. The last column name

```
*SAMPLE 3 - DATABASE FACILITY
*
E PROF <- (PRODNAME,PROFIT)TABLE GET 'FACTORY=NORTHERN'
:     FROM PROFITS
E PRODUCED <- (PRODNAME,PRODAMT)TABLE GET 'FACTORY=NORTHERN'
:     FROM PRODUCTION
E USED <- (PRODNAME,RESNAME,RESAMT)TABLE GET 'FACTORY=NORTHERN'
:     FROM RESOURCES
E LIMIT <- (RESNAME,RESLIMIT)TABLE GET 'FACTORY=NORTHERN'
:     FROM RESLIMITS
E NUMPRODUCTS <-  NUMROWS PRODUCED
*
E NUMRESOURCES <-  NUMROWS LIMIT
*
DATA= NUMPRODUCTS, NUMRESOURCES, PROF, PRODUCED, USED, LIMIT
*
VAR= X(J), J IN 1 THRU NUMPRODUCTS
*
MAXIMIZE
S PROF[J] X(J)
J IN 1 THRU NUMPRODUCTS
*
FOR I IN 1 THRU NUMRESOURCES
S (USED[I;J] - PRODUCED[J]) X(J) ≤ LIMIT[I]
J IN 1 THRU NUMPRODUCTS
*
```

Figure 4
Database Example

specifies the column in which the numeric data is contained; preceding column names each specify a dimension of the resulting array. For example, the USED array in the above example, will have rows corresponding to the unique values of PRODNAME and columns corresponding to the unique values of RESNAME. The numeric values for the northern factory are obtained from the RESAMT column of the RESOURCES relation and placed into this 2-dimensional array. Default values of zero are provided if there is no data in the relation for particular product-resource combinations.

Continuing the above example, NUMROWS is an MPGEN function that returns the number of rows in an array. NUMPRODUCTS and NUMRESOURCES are computed for later use in specifying the index sets for the problem. The DATA statement checks that all the data are present and also saves it in the random access file (as JUNEPRODDAT, if JUNEPROD is the name of the problem statement). The arithmetic expression appearing in the constraint definition uses USED and PRODAMT values to compute technology coefficients corresponding to the amount of resource used per unit of product. Note that the problem statement will be valid again next month (even though the data may have changed) and that a record is automatically maintained of the data used to compute each month's production plan.

As another example, suppose that unit profit margins are calculated from the prices of the products (which are invariant over factory and market) and their standard costs (which depend on both the product and the factory where it is produced). Then the Profits relation might be replaced by the following two relations:

```
Prodprices(Prodname, Price)
Prodcosts(Prodname, Factory, Stdcost)
```

and the unit profits array above could be calculated as follows:

```
E PRICECOSTS <- GET 'FACTORY=NORTHERN' FROM
                    JOIN 'PRODPRICES*PRODCOSTS OVER PRODNAME'
E PROF <- ((PRODNAME,FACTORY,PRICE)TABLE PRICECOSTS)
                  - (PRODNAME,FACTORY,PRICE)TABLE PRICECOSTS)
```

where the temporary relation, Pricecosts, resulting from the relational join operation consists of a concatenation of the rows of Prodprices with those of Prodcosts where the Prodname values are equal:

```
Pricecosts(Prodname, Price, Factory, Stdcost)
```

5. USING APL TO FORM CONCISE PROBLEM STATEMENTS

There are many situations in which the power of the APL language can help develop a very concise problem statement. A few examples are described below.

Logical Conditions

Suppose that the index variable, I, appears in one of the For Index lines in a Constraint Group and that the RHS coefficient should equal one when I = 1 and should equal zero otherwise. Instead of generating a data vector for the RHS of zeroes and ones, the RHS Coefficient Expression can be stated simply as the logical expression, I = 1 (APL returns a '1' if a logical expression is true and a '0' if it is false).

Figure 5 is an example of the use of logical conditions to specify constraint coefficients in a network problem. The data is generated in the first few lines of the problem statement. The SHAPE function generates a matrix with 13 rows corresponding to the arcs and two columns specifying, respectively, the 'from' and 'to' nodes for the arcs. T(A) represents the amount to be transported on the Ath arc. The logical expression I = ARCS[A;1] will evaluate to 1 if arc, A, emanates from node, I. This will place a '1' in the Ith row and Ath column of the tableau. The other constraint coefficient is interpreted similarly. Finally, the last line in the problem statement erases all of the data.

Finally, consider the constraint set:

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_{ij} \leq b_i \ , \ 1 \leq i \leq n$$

```
* GENERALIZED NETWORK PROBLEM FORMULATION
* SAMPLE TRANSSHIPMENT PROBLEM (FROM SCHRAGE, PAGE 125)
*
E NODES <- 1 THRU 9
E ARCS <- 13 2 SHAPE 1 3 1 4 2 3 2 4 2 5 3 6 3 7 4 6 4 7 4 8 5
:    7 5 8 5 9
E COST <- 1 2 3 1 2 5 7 9 6 7 8 7 4
E EXOG <- 9 8 0 0 0 -3 -5 -4 -5
*
E NARCS <- NUMROWS ARCS
*
VAR=T(A), A IN 1 THRU NARCS
*
MINIMIZE
S COST[A] T(A)
A IN 1 THRU NARCS
*
FOR I IN NODES
S (I=ARCS[A;1]) T(A) - S (I=ARCS[A;2]) T(A) = EXOG[I]
A IN 1 THRU NARCS,     A IN 1 THRU NARCS
*
E )ERASE NODES ARCS NARCS COST EXOG NARCS
*
```

Figure 5
Use of Logical Conditions to Specify Coefficients

The index set for j varies with the value of i. It can be written in MPGEN as:

```
(~ (1 THRU N) = I) / 1 THRU N
```

where '~' is logical NOT and '/' is the APL 'compression' operator (see [6]).

Other APL Expressions and User Defined Functions

Any APL statement that returns a scalar result can be used as an objective function, constraint or RHS coefficient. As a non-trivial example, consider the following term which appears in the objective function of a problem in [2]:

$$(N_k[v_{ijk\ell} - \sum_{r \in R_\ell} P_{rj\ell} h_{rj\ell ik}])$$

This can be modelled in MPGEN as:

```
(N[K] +.× V[I;J;K;L] - P[;J;L] +.× H[;J;L;I;K])
```

Here the APL 'inner product' operator, +.×, is used to perform the multiplication of the p's and q's and the summation of the resulting products over r.

The model in [2] is concerned with the optimal assignment of programs and datasets to storage devices. The following constraint ensures that each program and dataset is assigned to only one device:

$$\sum_{u \in U_k(i)} \sum_{j \in J_\ell} x_{ujk\ell} - |U_k(i)| y_{ik\ell} = 0 \quad , \quad \forall \; i \in I_k^p, \; \ell \in L, \; k \in K$$

Here, for example, Uk(i) is the collection of data sets required by program i in usage class k. This is represented in MPGEN by:

```
FOR L IN L
FOR K IN K
FOR I IN IP[K;]
S S X(U,J,K,L) - (+/0<U[K;I;])Y(I,K,L)=0
U IN U[K;I;], J IN J[L;]
```

Here, U is a three dimensional array in which the positive elements in each vector, U[K;I;], represent the datasets used by program i in usage class k (recall that zeros are ignored in index expressions). The APL expression, + / 0 < U[K;I;], computes the cardinality of Uk(i). The expression, 0 < X, returns a vector of 0's and 1's where the 1's correspond to positive elements in X. This vector is then summed by the 'sum reduction' operator, +/ (usually pronounced 'plus over').

Finally, users who are knowledgeable in APL can code their own functions for inclusion in the problem statements. This can lead to extremely elegant and powerful model definitions.

Interactive Input of Data

If objective function, constraint or right-hand side coefficient expressions vary from run to run (perhaps for sensitivity testing purposes) this data may be input interactively during problem interpretation. The user simply types the word 'ASK' in place of the relevant Variable or RHS Coefficient Expression. As it interprets the problem MPGEN will prompt the user for the data items after displaying the current Objective or Constraint Definition Line and the appropriate index values.

6. INTERACTING WITH THE SYSTEM

There are two modes of interaction with the system. One is to sign-on to APL, load the MPGEN workspace, type 'RUNPROBLEM' and interact with the system via a menu-driven interface as shown in Figure 6. The second is to define the problems externally and to pass them as files to APL using a system command file. This is the method used by the LPFORM expert system as explained more fully in the next section.

In the interaction shown in Figure 6, the data dictionary of previously stored problem definitions is listed and the TRANSPORT problem is chosen and parsed. This generates data triples in MPS format. Before running the problem the results of the parse are

first displayed in both 'extensive' algebraic form and as a LP tableau. The final step shown in the example generates a file containing the problem statement in MPS format for input to the LINDO system.

```
      RUNPROBLEM

OPTION?  (OR TYPE HELP)  *** MPGEN ***
HELP

TO SELECT AN OPTION TYPE THE FIRST 3 LETTERS:

    NAM: SET PROBLEM NAME
    DBS: USE THE DATA BASE SYSTEM
    DAT: EDIT DATA
    DEF: DEFINE PROBLEM STATEMENT
    DIC: VIEW PROBLEM DICTIONARY
    PAR: PARSE PROBLEM STATEMENT
    SHO: SHOW RESULTS OF PARSE IN ALGEBRAIC FORM
    TAB: DISPLAY LP TABLEAU
    LIN: GENERATE LINDO STATEMENT
    APL: EXECUTE APL STATEMENTS
    STO: STOP EXECUTION

OPTION?  (OR TYPE HELP)  *** MPGEN ***
DIC
PROBLEM/DATA NAME

    AIRLINE
    AIRLINEDAT
    ALGEBRAIC
    DBASE1
    DBASE1DAT
    MULTDIV
    MULTDIVDAT
    TRANSARCA
    TRANSARCB
    TRANSPORDAT
    TRANSPORT

OPTION?  (OR TYPE HELP)  *** MPGEN ***
NAME

PROBLEM NAME?  TRANSPORT
```

Figure 6
Sample Interaction with the MPGEN System

```
OPTION?  (OR TYPE HELP)  *** MPGEN ***
PARSE

RESTORING VALUES FOR:  M N COST S D
-------- PARSING PROBLEM STATEMENT = TRANSPORT
* SAMPLE TRANSPORTATION PROBLEM FROM WAGNER PAGE 215

DATA= M, N, COST[MxN], S[M], D[N]
*
VAR= X(I,J), I IN 1 THRU M,  J IN 1 THRU N
*
MINIMIZE
S S COST[I;J] X(I,J)
I IN 1 THRU M, J IN 1 THRU N
*
FOR I IN 1 THRU M
S X(I,J) ≤ S[I]
J IN 1 THRU N
*
FOR J IN 1 THRU N
S X(I,J) ≥ D[J]
I IN 1 THRU M
*

LP TRIPLES (ROW,COL,VALUE) HAVE BEEN FORMED.
TABLEAU SIZE: ROWS = 6  COLS = 7  TRIPLES = 23

STORED VALUES IN FILE FOR: M N COST S D
ERASED VALUES FROM WS FOR: M N COST S D

OPTION?  (OR TYPE HELP)  *** MPGEN ***
SHOW

MINIMIZE
X11 + 2X12 + 2X21 + 3X22 + 3X31 + 4X32

X11 + X12 ≤ 1
X21 + X22 ≤ 2
X31 + X32 ≤ 3
X11 + X21 + X31 ≥ 2
X12 + X22 + X32 ≥ 3
```

Figure 6
Sample Interaction with the MPGEN System (cont'd)

```
OPTION?  (OR TYPE HELP)  *** MPGEN ***
TABLEAU

X11 X12 X21 X22 X31 X32 RHS

  1   2   2   3   3   4   0
  1   1   0   0   0   0   1
  0   0   1   1   0   0   2
  0   0   0   0   1   1   3
  1   0   1   0   1   0   2
  0   1   0   1   0   1   3

OPTION?  (OR TYPE HELP)  *** MPGEN ***
LINDO

-------------- START OF GENLINDO -----------------
TRANSPORT -  FROM APL WS AT   8/17/1986    1:06:23
--------------- END OF GENLINDO ------------------

OPTION?  (OR TYPE HELP)  *** MPGEN ***
STOP
```

Figure 6
Sample Interaction with the MPGEN System (cont'd)

Corrections and modifications to the problem statement are made using a special editor accessed by the DEF menu command. Similarly the data coefficients can be input and modified using the same editor via the DAT menu command. When a newly defined problem is parsed the problem definition is automatically added to the random access file. As each DATA= statement is executed, the system first checks if the declared data items are in the workspace. If they are, then their dimensions are checked. If a data item is missing, the user is asked to input its values (note that this is usually the most convenient method for initially entering problem data). The values of the data items are then stored in the random access file. As illustrated in Figure 6, if a previously defined problem is accessed its associated data is automatically retrieved from the random access file at the start of the parse step. The data (which may have been modified by EXEC statements in the problem statement) is automatically stored back into the file and erased from the workspace at the end of the parse. These defaults can be adjusted by the user to assist in generating and controlling many different versions of the same problem for exploration of management alternatives.

The DBS menu option allows the user to interact with the database system to define the schema and to input and maintain data in the relational database. Note that the database retrieval facility is always accessible from the MPGEN parser so no special steps need be taken by the user.

7. USE OF MPGEN WITH THE LPFORM EXPERT SYSTEM

As indicated in Figure 1, LPFORM generates algebraic statements in the MPGEN format and writes them to a file for subsequent processing by MPGEN. MPGEN then generates the MPS statement for input to MPSX or LINDO. As all of this is achieved in one step by use of a command file, the presence of MPGEN is transparent to the user. An exception to this occurs if the data items referred to in the symbolic statement are not defined. As explained above, MPGEN will then request that the user define them interactively before it proceeds with the parse.

The use of MPGEN in this way has greatly simplified the development of a working expert system. The major advantage has been that it is possible to avoid developing a complex arithmetic capability in PROLOG. As we have seen, the use of the APL language within the context of MPGEN is useful in forming very concise and general problem statements.

LPFORM is concerned with the generation of symbolic problem statements. There are two forms. In 'symbolic mode', the system makes no attempt to link the data coefficient symbols in the problem statement with data values (unless these are input directly by the user). In 'data mode', the symbols in the algebraic statement are logically linked to data values either explicitly, or through reference to external tables or through database retrieval statements.

Currently, the physical linkage of the symbols for sets and data coefficients is handled through the MPGEN system in a number of ways:

(1) If the problem statement is generated in symbolic mode, the assignment of data to the symbols is handled entirely through MPGEN. The simplest way is simply to input the data values when prompted during the parse of the problem statement. Alternatively, the data may be stored directly in the workspace by the user prior to running LPFORM.

(2) The expert system accepts data inputs directly from the user and translates these into EXEC statements at the beginning of the MPGEN problem statement. This is normally the way in which sets are defined.

(3) Users provide the names of the relevant external data tables during their interaction with LPFORM. LPFORM then maps the data items in the tables to the symbolic names in the problem statement and generates EXEC statements that cause MPGEN to access the file(s) containing the tables and to read them into appropriately named APL variables.

(4) Users provide database retrieval statements that are written directly by LPFORM as EXEC statements as in Example 3.

(5) Users provide arithmetic statements (or a mixture of data

retrieval and arithmetic statements) during their interaction with LPFORM. LPFORM simply passes these through for execution in MPGEN. Note that much can be done with only the rudimentary commands illustrated in Section 2.

A current research and development objective is to develop the expertese necessary to allow LPFORM to automatically generate the database retrieval and arithmetic expressions necessary to perform (4) and (5) automatically.

8. SUMMARY

The problem definition language described in this paper provides a convenient and concise means for defining linear and integer programming problems. Because of its labor saving characteristics, it allows the model builder to implement models more easily and to experiment with alternative formulations. This is most useful in a DSS environment since it allows one to quickly form a 'data base' of models in a wide range of areas such as cash management, capital budgeting, production planning and scheduling, transportation, facilities location, marketing and so on.

MPGEN's concise algebraic statements are suitable target outputs for current AI-based languages such as PROLOG and it is proving to be a very useful tool in the development of the LPFORM system for formulating LPs. In the long run, however, the objective is to integrate all the components shown in Figure 1 and to include additional components for reporting, analysing and explaining the results of model runs.

References

1. Astrahan, M. M., and Chamberlin, D. D., 'Implementation of a Structured English Query Language', Communications of the ACM, Vol. 18, No. 10, October, 1985, pp. 580-588.

2. Balachandran, V. and Edward A. Stohr, "Optimal Pricing of Computer Resources in a Competitive Environment," Working Paper No. 268, Center for Mathematical Studies in Economics and Management Science, Northwestern University, 1978.

3. Clocksin, W. F. and C. S. Mellish, Programming in Logic, Springer-Verlag, New York, 1981.

4. Creegan, J. P., 'DATAFORM: A Model Management System', Working Paper, Ketron, Inc., Arlington Va., 1985.

5. Fourer, R., Modelling Languages versus Matrix Generators for Linear Programming', ACM Transactions on Mathematical Software, Vol. 8, No. 2, June 1983.

6. Gilman, L. and A. J. Rose, APL: An Interactive Approach, John Wiley & Sons Inc., New YorK, 1984 (3rd edition).

7. Greenberg, Harvey J., 'A Functional Description of ANALYZE: A Computer-Assisted Analysis System for Linear Programming Models', ACM Transactions on Mathematical Software, Vol 9, No. 1, March 1983, pp. 18-56.

8. Haverly Systems Inc., OMNI Linear Programming System: User Manual and Operating Manual, Denville, N.J., 1977.

9. IBM Mathematical Programming Language Extended/370 (MPSX/370), Program Reference Manual, SH19-1095, IBM Corporation, Paris, France, 1975.

10. Ma, Paichun, 'An Intelligent Approach Towards Formulating Linear Programs', Unpublished Dissertation Proposal, Graduate School of Business Administration, New York University, 1985.

11. Ma, P., F. H. Murphy and E. A. Stohr, 'Design of a Graphics Interface for Linear Programs', Working Paper 111, Center for Research in Information Systems, Graduate School of Business Administration, New York University, 1985.

12. Meeraus, A., 'General Algebraic Modelling System (GAMS), User's Guide, Version 1.0, Development Research Center, World Bank, 1984.

13. Murphy, F. H. and E. A. Stohr, 'An Intelligent System for Formulating Linear Programs', International Journal of Decision Support Systems, Vol 3, No. 2, 1986.

14. Murphy, F. H. and E. A. Stohr, 'The Science and Art of Formulating Linear Programs', Working Paper 110, Center for Research in Information Systems, Graduate School of Business Administration, New York University, 1985.

15. Schrage, Linus, Linear, Integer, and Quadratic Programming with LINDO, The Scientific Press, Palo Alto, 1984.