The use of linear programming is impeded by the effort required to express a model as a matrix and to collect and handle data. An experimental interactive system called LPMODEL *simplifies the development of linear programming models. It frees the user from the necessity of expressing the model as a matrix.* LPMODEL *provides a nonprocedural language for constructing a model in terminology that is natural to the problem, using ordinary algebraic expressions. With this language, the user can express a model concisely by generic constraints which the system interprets in conjunction with a data base to generate a concrete model for optimization.*

The design of the system and its terminology and data base subsystems are discussed. An informal description is given of the modeling language which involves both ordinary arithmetic operations and symbolic operations with associated semantics. Experience with the system in agricultural modeling is described.

A system for constructing linear programming models

by S. Katz, L. J. Risman, and M. Rodeh

Linear programming has become a valuable aid to decision making in many fields; see, for example, References 1 and 2. However, the effort that is required to collect and organize data, to express a linear programming (LP) model as a matrix, and to input the matrix of coefficients to the computer impedes the use of this valuable tool. Several systems have been developed to help in the process of generating the matrix, as discussed later in the section on other modeling systems. The computer professional now has available powerful tools for developing mathematical programming models. While some progress has been made in meeting the needs of the less sophisticated user, the process of constructing a model is still complicated and slow. The required data have to be laboriously prepared, and the constraints and goal of the entity being modeled must be expressed in terms of tables or matrices.

The increased availability of interactive terminals and the general trend towards providing better programming tools for the user have made it natural to consider systems that concentrate on the *construction* and *development* of linear programming models rather than on their solution. Thus, the experimental system discussed here, called LPMODEL, makes linear programming more accessible to the user who is not an expert in computers or operations research. This interactive system simplifies the collection and organization of data. It provides a language, LPM (Linear Programming Modeling Language), which frees the user from the necessity of expressing a model as a matrix. LPMODEL generates input for a standard linear programming system which then actually performs the optimization of the model.

To achieve the goal of accessibility, principles that have been found useful in other applications have been employed. These include *modularity*—dividing the larger task into smaller subtasks that can be treated separately, *naturalness*—expressing the model in terms natural to the domain of the problem, and *abstraction*—expressing the nature of the problem independently of particular details such as numerical constants. Note that directly constructing a matrix using any programming language violates the above principles, in particular, naturalness.

The goal of this work is not to provide yet another method for solving linear programming problems, nor is it to provide heuristic guidelines for constructing a linear programming model. Rather, a declarative language is described which was designed to be sufficiently expressive yet simple to learn and use, and a system organization is presented which encourages the principles mentioned above. This paper can be viewed as a case study in special-purpose language and system design. Although the examples given below and the experience with the system to date have been in agricultural planning, the system is designed to be more generally applicable.

System design

Besides the general considerations of modularity and simplicity, the specific properties of linear programming models can be exploited in the system and language design.

model properties

A linear programming model consists of a set of linear constraints and a linear objective function to be minimized or maximized. The most basic property of such a model is that it is declarative: a problem is described and defined—no algorithm need be presented by the user for its solution, since that will be done in a standard way for all models. Another basic property is that the various constraints in a model are independent. Thus changing

one has no effect on the others, although it does, of course, affect the solution to the model.

In addition to these basic properties, there are some observations that hold for many real-life linear programming models:

- Linear programming variables that fulfill logically similar roles fall into natural groupings; e.g., on a farm natural groupings might be the crops or the tractors.
- Some of the constraints have a similar structure. Again consider a farm and assume that the water demand and supply are given on a monthly basis. For each month, a constraint is required to express the bounds on water supply in that month, yielding a set of constraints with similar structure.
- Some aspects of a model change more often than others. In many cases one conceptual model is applied to different sets of numerical values. In other cases some of the groups of variables or constraints are slightly changed, whereas the model as a whole retains its structure. We call this phenomenon *nonuniform stability*.

The above considerations led to a system for constructing linear programming models with components for the following tasks:

1. Defining a *terminology* which is natural for the problem domain
2. Creating and maintaining a *data base* for associating data values with identifiers that represent known quantities in the model
3. Expressing an *abstract model* independently of particular values of known data

abstract model

The abstract model uses the terminology that is defined separately. It includes two syntactically distinguishable types of identifiers: those that represent unknown quantities, to which the linear programming solution will ultimately assign values (the usual linear programming variables), and those which represent known quantities (the linear programming technological coefficients and other constants). In the data base, the association is made between those identifiers representing constants and their actual values. Thus, the terminology and the data base can be viewed as the *environment* of the abstract model. When the list of names represented by a term is provided by the terminology and the values are provided from the data base, the abstract model can be interpreted to yield a specific *concrete* model that is in a form ready for solution by a linear programming system.

Naturally, the development of the three components of a model is not really independent: as the constraints of the abstract model are being written, the need for additional terminology will arise,

and it will become clear exactly which data values are needed. The main advantage of separating the model into three parts is that if one part is changed, the others are often unaffected. For example, if the particular numerical values must be updated (e.g., the amount of land available for farming changes, or the costs of various items rise because of inflation), this can be done without affecting the terminology or the abstract model. The new components can then be recombined into a new concrete linear programming model. Similarly, if a new crop is added to the list of crops in the terminology, this need not influence the abstract model, although new numerical information usually will have to be added to the data base.

Another advantage of the system is that the data base can be updated regularly and used to generate reports independently of the linear programming context. This should help to alleviate the common difficulty that results wherein every time a linear programming solution is desired, a tremendous outburst of reporting and bookkeeping is required to gather the needed information. If this system is used as intended, the data can be collected and entered continuously, and running a particular linear programming model becomes a much less painful task.

In fact, independently of linear programming, data bases are already widely used for bookkeeping and administrative purposes. LPMODEL can be viewed as yet another way to use an already existing data base. In such a situation, special information-gathering exclusively for linear programming becomes unnecessary.

Defining the terminology

As explained above, it is generally convenient to define a terminology natural to the task at hand. In order to be precise about the nature of the subsystem for this task, which is called *terminology*, a few definitions are necessary.

A *name* is simply a sequence of letters, digits, and '_', which does not start with a digit and contains no blanks. There are two kinds of names: atoms and terms. An *atom* is a name which in itself represents some aspect of reality, and does not stand for any other name in the model. Examples could be *COTTON*, *PLUMS*, or *SALARY*. A *term*, however, is an abbreviation for a list of atoms. Only the terms are defined by the Terminology system. Any name that is not given a definition in Terminology is assumed to be an atom. Note that the same atom may appear in several terms. If a farm were being modeled, the definitions needed in Terminology might be:

```
MONTH ← MAY, JUNE, JULY
CROP ← COTTON, ONIONS, GREEN_PEP, WHEAT,
    PEAR, GRAPE
```

In more complex situations, it is convenient to "build up" a definition in stages, e.g.,

```
FIELD ← COTTON, ONIONS, GREEN_PEP, WHEAT
ORCHARD ← PEAR, GRAPE
CROP ← FIELD, ORCHARD
```

This would give *CROP* the same definition as previously, since after each term has been defined, no matter how this was done, it stands only for the associated list of atoms (which is called the "expanded" definition of the term). It is also legal to mix atoms with previously defined terms in defining new terms. Thus

```
CROP ← FIELD, PEAR, GRAPE
```

is yet another way to define the same list of crops as above.

Numerous editing aids and prompts are built into this system. Defining a new term is done as indicated above, by a left arrow. Other commands such as *PRINT*, *CHANGE*, *EXPAND*, or *LIST* allow the user to bring the terminology to a state where it reflects his terms of reference.

The data base

The data base subsystem is used to associate values with the identifiers that stand for constants in an abstract model. Again, we will first define somewhat more precisely what can be given a value in this system.

An *identifier* is either a single name (atom or term) or a series of names separated by periods. For example, *COT_MIN*, *ONIONS*, *LABOR.CROP.MONTH*, and *WATER.MONTH* are all identifiers. No repetitions of the same name are allowed within a single identifier.

A *primitive identifier* is an identifier comprised entirely of atoms, e.g., *LABOR.COTTON.MAY*. An identifier is interpreted by substituting for each of its component terms each of the atoms in its definitions. An identifier represents the collection of all the resulting primitive identifiers. From the example of a terminology defined in the previous section, *WATER.MONTH* represents the primitive identifiers *WATER.MAY*, *WATER.JUNE*, and *WATER.JULY*.

There is a "canonical order" among the primitive identifiers represented by an identifier. The first primitive identifier consists of the first atom from each term in the identifier; the next one leaves all unchanged except the right-most term (where the next atom is used), and so forth. Thus for *WATER.CROP.MONTH*, the

items would be present in the order corresponding to

WATER.COTTON.MAY, *WATER.COTTON.JUNE*,
WATER.COTTON.JULY, *WATER.ONIONS.MAY*,
WATER.ONIONS.JUNE, *WATER.ONIONS.JULY*,

and so forth. In all, *WATER.CROP.MONTH* represents the 18 primitive identifiers obtained from substituting the atoms from the terms *CROP* and *MONTH*.

As indicated above, identifiers can be used either as variables (when followed by a question mark) or as a way of referring to known values, without explicitly writing the numbers in the model, i.e., as constants. The data base system gives values to those identifiers used as constants by associating a value with each primitive identifier. If a primitive identifier used as a constant in the model does not have a value in the data base, a default value of zero will be given.

To assign values, an identifier is written followed by ← and a list of numbers, one for each primitive identifier it represents, in the canonical order.

prompting mode

The system has a prompting mode that is activated by not giving all the required values for an identifier (or not giving any values at all). This mode presents the next primitive identifier that needs a value and waits for the value to be entered. For example, if just *COT_MIN* ← is written by the user, since this is an atom, the system will respond

COT_MIN ←

and the user is expected to enter the required value. If *WATER.MONTH* ← is written, the system will reply

WATER -
  MAY ←

and after the user enters a number, the system will continue

  JUNE ←

scanning in this way all the primitive identifiers represented by *WATER.MONTH*. It is possible to pass back and forth from the prompting to the regular mode of inserting a series of values at once, or to define only relevant parts of an identifier.

If an identifier is written without the left arrow, the system will list the names and associated values of all the primitive identifiers it represents.

There are various editing options to add new values, update old ones, and display the present state of the data base. As was men-

tioned earlier, the data base can, and should, be used for obtaining reports on the state of the economic entity, independently of linear programming.

LPMODEL is designed so that, in principle, another data base system could be used for model construction instead of this special-purpose one. The only requirement is that it be able to supply answers to a series of requests for values that will come from the system. These requests are generated when the user asks the system to construct a concrete model from an abstract model so that a linear programming matrix can be built and used as input to a linear programming system for solution.

The abstract model

An abstract model is composed in the Linear Programming Modeling Language (LPM) by listing any number of constraints and exactly one objective function. The standard linear programming formulation has the form

$$\sum_{j=1}^{n} a_{ij} y_j \leq b_i \qquad 1 \leq i \leq m \qquad \text{(the constraints)} \tag{1}$$

$$\text{maximize} \sum_{j=1}^{n} c_j y_j \qquad \text{(the objective function)} \tag{2}$$

A potential user of LPMODEL, e.g., a farm manager, is not expected to think in terms of a_{ij}. He may conceive of a model verbally by statements such as these: (1) The total monthly water consumption for all crops must not exceed the monthly water allotment. (2) Maximize the total profit from all the crops.

The abstract mathematical notation is quite precise and concise. However, the statement of the problem in ordinary English is quite natural and easily understood. The language LPM endeavors to capture some of the conciseness of mathematical notation without losing the naturalness of ordinary language. In order to explain some of the features of the language, the transformation of the standard mathematical notation for a model to the notation of LMP is demonstrated below. The set of indices $\{j \mid 1 \leq j \leq n\}$ models some natural sets of objects such as a set $CROP$ of crops on a farm. Similarly, $\{i \mid 1 \leq i \leq m\}$ may model a set $MONTH$ of months. Then Equations 1 and 2 may be rewritten as

$$\sum_{j \in CROP} a_{ij} y_j \leq b_i \qquad i \in MONTH \tag{1A}$$

$$\text{maximize} \sum_{j \in CROP} c_j y_j \tag{2A}$$

Here $MONTH$ and $CROP$ are terms, and would be defined separately, as discussed in the earlier section on defining terminology.

The indices j and i are auxiliary variables. Replacing them by *CROP* and *MONTH*, respectively, does not cause any confusion provided that the multiplication is done element by element. Note that *CROP* and *MONTH* then do the double service of identifying the range of the indices and of acting as the index itself. (The case of a double summation over the same set of indices causes difficulties and may be avoided by renaming.) Equations 1A and 2A are then transformed to

$$\sum_{CROP} a_{CROP,MONTH} y_{CROP} \leq b_{MONTH} \tag{1B}$$

$$\text{maximize} \sum_{CROP} c_{CROP} y_{CROP} \tag{2B}$$

Observe that the letter y designates linear programming variables. In order to make more explicit the distinction between linear programming variables and identifiers that represent constants, and to allow more freedom in the choice of names, a special character *?* has been chosen to follow an identifier intended as a linear programming variable. To increase readability, descriptive names can be used, e.g., *WATER* instead of a, *WATER_BND* instead of b, and *PROFIT* for c. Equations 1B and 2B may be written linearly as follows:

```
∑ CROP: WATER.CROP.MONTH × CROP?
      ≤ WATER_BND.MONTH                    (1C)

maximize ∑ CROP: PROFIT.CROP × CROP?       (2C)
```

The dots within an identifier indicate that it is comprised of a series of names, as was explained in the section on the data base. The colon is used to separate the index of summation from the summand.

An explicit indication of the right boundary of the summand has been found to be helpful in avoiding misunderstanding of complex expressions. The square brackets were chosen to represent the '$\sum$' notation in LPMODEL. The left bracket may be preceded by the optional keyword *SUM*.

```
SUM [CROP: WATER.CROP.MONTH × CROP? ]
    ≤ WATER_BND.MONTH                      (1D)

maximize [CROP: PROFIT.CROP × CROP? ]      (2D)
```

In many practical cases, the index of summation is equal to the variable name mentioned in the summand (such as in $\sum_i a_{ij} y_i$) and may be omitted. Therefore, Equations 1D and 2D may be rewritten:

```
[ WATER.CROP.MONTH × CROP? ]
    ≤ WATER_BND.MONTH                      (1E)

maximize [ PROFIT.CROP × CROP? ]           (2E)
```

Note that the *generic constraint* (1E) represents a collection of constraints that are logically similar, as opposed to the original formulation where the constraints were arbitrarily numbered by $\{i \mid 1 \leq i \leq m\}$. A model consists of a number of such generic constraints and a single objective function.

The language LPM extends the concepts implicit in the above example. It allows inequalities or equalities between arithmetic expressions involving variables and constraints. The grammar of the language given in Appendix A enforces the linearity requirement that a variable may not be multiplied by a variable. A nonlinear expression is a syntactic error that is detected by the compiler and causes an error message to be printed. A more complex example of a generic constraint is

```
[MONTH: CROP: WATER.CROP.MONTH × CROP?]
    + [WATER.LIVESTOCK × LIVESTOCK?]
    ≤ WATER_BND + WATER.EXTERNAL_SUPPLY?
```

It should be noted that the arithmetic operations are permitted between constants, whose values are defined in the data base, and variables, whose values are undefined during the construction of the model. Numerical values are associated with variables only at the final stage in the solution of a model.

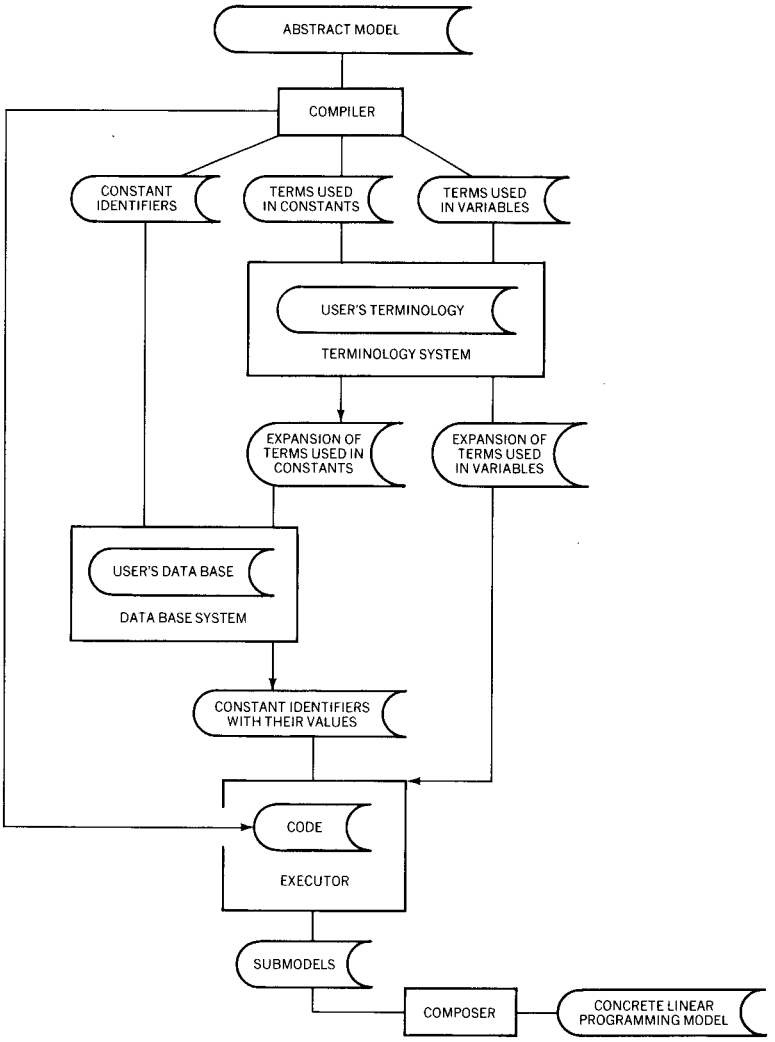
The semantics of arithmetic operations in LPM appear straightforward to the user. The syntax is intentionally quite close to the familiar notation for arithmetic expressions used in high school algebra. The treatment by the LPMODEL system of operations involving primitive constants is fairly standard, whereas expressions involving terms defined in Terminology lead to the creation of implicit loops.

However, arithmetic operations involving variables must be treated quite differently by the system. They represent essentially symbolic operations that are not executed arithmetically by LPMODEL but determine the constraints and objective function in a model.

Implementation

The goals that influenced the high-level system and language design also affected the implementation decisions that are not visible to the user. For example, the connections of the abstract model with the terminology and the data base are delayed to as late a stage of the processing as possible. This delay is again motivated by the greater stability of the abstract model, so that computation will not be unnecessarily repeated. In addition, the independence of the constraints is reflected strongly in the implementation.

Figure 1 Combining an abstract model, a terminology, and a data base



processing The processing may be summarized as follows (see Figure 1):

1. The abstract model is compiled alone, without any knowledge of the environment (i.e., the terminology or the data base). The result for each line is a section of code in a programming language. This code requires a terminology and data values as input, and in conjunction with some standard system functions, will produce a concrete submodel. In addition to the code itself, the data base references mentioned in every line of the model are produced, as well as two sequences of terms—those which are used in constants and those used in variables.

2. Next, the lists of atoms associated with the relevant variables are obtained from the Terminology system.
3. The lists of atoms associated with the terms used in data base references are found. These help determine exactly which data base values are required.
4. The data base is used to obtain the values for the requests from 1 using the information from 3.
5. The code generated in 1 is then executed for each line of the model, with the results of 2 and 4 as input, producing a collection of independent concrete submodels, one for each line.
6. Finally, the submodels are combined into one large concrete model, where all appearances of the same primitive variable are associated. Later, a system for solving concrete linear programming models may be applied.

Note that if a single line of the abstract model is changed, steps 1 through 5 above for all the *other* lines are unaffected and need not be repeated. If changes are made in the data base, steps 1 through 3 are unaffected, and a new terminology leaves 1 unaffected.

An experimental version of LPMODEL with the above design is presently implemented in APL on an IBM System/370 Model 168. Most of the system has also been implemented on a small computer, the IBM 5110. At present, the concrete model which is the result of LPMODEL is solved by the linear programming package in STATPACK of APL.[3] Large models cannot be handled by this package, and the intention is ultimately to connect the result of LPMODEL to MPSX/370 (Mathematical Programming System Extended/370).[4] The use of a small computer is being investigated so that the more frequent uses of the system can be done locally and inexpensively. These uses include updating the data base, defining terminology and abstract models, and obtaining reports. The actual execution of large linear programming models would still have to be done on a central computer.

Other modeling systems

Much recent work in mathematical programming systems has centered on the problem of model development and matrix generation.

A number of powerful matrix generating and report writing systems, such as the IBM MGRW (Matrix Generator and Report Writer),[5] are available. These systems facilitate the definition and manipulation of tables of data using a dictionary of terminology and generate a matrix to be input to MPSX/370 for optimization. Whereas MPSX/370 requires a matrix to be input by column, MGRW permits a matrix to be generated by row or by column.

Another approach is taken by the extended control language, ECL, of MPSX/370.[4] This system provides a convenient interface between the programming language PL/I and MPSX/370. The user can write PL/I programs that generate and modify matrices, place an MPSX/370 input "deck" in a PL/I structure, and invoke MPSX/370 for the solution. The user's PL/I program can access PL/I-based data files and data base systems.

Note that in the above systems, the user is required to conceive of a model as a matrix. Then, using either a special table-oriented language or a general-purpose language, the user describes the elements to be entered in each row and column of the matrix and in the right-hand side of the model. The user is assumed to be familiar with MPSX/370 and its input requirements. Similarly, the user who is willing and able to "build" a matrix representing a model using APL code can then continue in that system by invoking the linear programming package in STATPACK of APL.[3]

Progress toward meeting the needs of the less sophisticated user is exemplified in the following systems.

The GPLAN system developed at Purdue University is a network data base management system implemented in FORTRAN.[6] The query language user can ask the system to run linear programming for a matrix extracted from the data base.

MPOS, the multipurpose optimization system developed at Northwestern University,[7] accepts algebraic input of a model in standard algebraic form; e.g., the user expresses the constraint $2X + 3Y \leq 100$ directly. Constraints in algebraic form must be of a very limited sort, closely tied to the matrix formulation. There is no provision for generic constraints, and each individual concrete constraint must be input with its explicit numerical coefficients.

LMC is the linear modeling capability of the conversational modeling language developed at Yale University.[8] The LMC language permits the formulation of linear programming models and provides an interface with MPSX/370. Specification of a model is in two stages, "equations" and "parameterization." Constraints are specified in English-like statements in which the user describes the linear programming matrix in words. Numerical values of coefficients are then given in assignment statements, called the parameterization. No provision for an interface with an independent data base is discussed in Reference 8.

One of the referees has pointed out that a recent working paper by Fourer and Harrison[9] contains an independent proposal for a linear programming system with a high-level declarative language similar in its philosophy to LPMODEL, though quite different in form. Their proposed system has not been implemented. That paper also contains an extensive discussion of previous linear programming and matrix-generation systems.

Conclusion

So far several problems from agriculture have been successfully formulated in LPMODEL. Among these have been the question of how much of each kind of fish to raise in some fishponds, analyses of whether to dry up fishponds or uproot orchards so that the land could be used for raising cotton, and linear programming for entire farms. Both farm managers with and without previous linear programming experience have used LPMODEL. The users have adapted to the notation very rapidly (after only one or two introductory explanations), and the abstract models have helped the farmers perfect the statement of the real constraints on the farms. In these experiments, the typical size of the final matrix has been 10 to 50 variables and less than 100 constraints. Presently, using a workspace of 128K, the APL procedures that solve the linear programming cannot handle much larger matrices, but this restriction will not be present in future versions of LPMODEL.

Typically, after the initial formulation and solution of the model, the results are analyzed by the users, and this leads to further refinements. We identified four basic types of changes to the initial model:

1. Fine tuning—minor changes that affect only the data base, e.g., adjusting prices or production ceilings.
2. Simple extension—new restrictions that fit into existing generic constraints. For example, incorporating a new crop involves adding it to the list of crops in the terminology and inserting its relevant data into the data base, but the abstract model often need not be changed.
3. Restriction—midstream reevaluation due to changed circumstances from earlier versions of the model. Some variables of the original model become constants (since their values may no longer be changed), and some additional constraints are added to the model.
4. Reformulation—changes to a few constraints and addition of new constraints as some aspect of the model is seen to be an inaccurate description.

As expected, the numerical constants in the data base were modified most often, the terminology was changed next most often, and the abstract model was surprisingly stable.

LPMODEL as now implemented is an experimental realization of the design goals discussed above. Although all problems from agriculture that have been treated were easily expressible in the language, the desire for simplicity led us to somewhat restrict the expressive power of the language LPM. Extensions are being considered to strengthen the existing language by permitting more general subscript calculations involving the Terminology.

The planned interface of the language with other data base management systems, such as IMS, will also enhance the applicability of the system.

The use of this system for report generation and even cost accounting is also being considered. LPMODEL identifiers and expressions involving only constants (i.e., referring to known values in the data base) can be used to specify reports and tables to be printed. For example, *CROP.FERT.MONTH* would show fertilizer requirements for each crop in each month. For cost accounting, the user would enter LPMODEL expressions involving constants, which would be recomputed at will using the updated values in the data base.

Our experience with LPMODEL has demonstrated the merits of an approach to model construction that does not require the user to conceive of a model as a matrix, but rather permits him or her to formulate a model concisely using ordinary algebraic expressions and terminology that is natural to the problem at hand.

ACKNOWLEDGMENT

We wish to express our appreciation to Zvi Weiss and Ingrid Schwarz of the IBM Israel Scientific Center and to Ilan Amir of the Technion, Israel Institute of Technology, Department of Agricultural Engineering, for their valuable participation which made this work possible.

Appendix A: The syntax of the LPM language

The production rules below define a simple-precedence grammar for a sentence ⟨sent⟩ in the LPM language.

⟨name⟩→⟨input name⟩
⟨identifier⟩→⟨name⟩ | ⟨identifier⟩.⟨input name⟩
⟨const factor⟩→⟨number⟩ | ⟨identifier⟩ | ((const exp))|
[⟨const sum⟩
⟨const sum⟩→⟨const expl⟩] | ⟨name⟩:⟨const sum⟩
⟨const term1⟩→⟨const factor⟩ | ⟨const term1⟩ × ⟨const factor⟩|
⟨const term1⟩ ÷ ⟨const factor⟩
⟨const term⟩→⟨const term1⟩
⟨const expl⟩→ ⟨const term⟩|−⟨const term⟩|
⟨const expl⟩+⟨const term⟩|⟨const expl⟩−⟨const term⟩
⟨const exp⟩→⟨const exp1⟩
⟨variable factor⟩→⟨identifier⟩?|((variable exp))|
[⟨variable sum⟩
⟨variable sum⟩→⟨variable exp1⟩]|⟨name⟩:⟨variable sum⟩
⟨variable term1⟩→⟨variable factor⟩|
⟨variable term 1⟩×⟨const factor⟩|
⟨const term⟩×⟨variable factor⟩|
⟨variable term1⟩÷⟨const factor⟩

⟨variable term⟩→⟨variable term1⟩
⟨variable exp1⟩→⟨variable term⟩|−⟨variable term⟩|
⟨variable exp1⟩+⟨const term⟩|
⟨variable exp1⟩−⟨const term⟩|
⟨variable exp1⟩+⟨variable term⟩|
⟨variable exp1⟩−⟨variable term⟩|
⟨const exp1⟩+⟨variable term⟩|
⟨const exp1⟩−⟨variable term⟩
⟨variable exp⟩→⟨variable exp1⟩
⟨r⟩→≤|=|≥
⟨sent⟩→⟨const exp1⟩ ⟨r⟩ ⟨variable exp⟩|
⟨variable exp1⟩ ⟨r⟩ ⟨const exp⟩|
⟨variable exp1⟩ ⟨r⟩ ⟨variable exp⟩|
⟨const exp⟩|⟨variable exp⟩|*MAXIMIZE*⟨variable exp⟩|
MINIMIZE⟨variable exp⟩

Appendix B: Summary of a simplified model in LPMODEL

Terminology:

```
FIELD ← COTTON, ONION
CROP ← FIELD, PEAR, AVOCADO
MONTH ← MAY, JUNE, JULY
```

Data base entries (note that the entries not appearing, such as *WATER.ONION.MAY*, will be given the value zero when requested.):

```
LAND : 2700
FIELD_LAND : 1850
WATER_BND :
 MAY : 200000
 JUNE : 260000
 JULY : 270000

WATER :
 COTTON :
  MAY : 65
  JUNE : 80
  JULY : 90

 ONION :
  JUNE : 60

 PEAR :
  JUNE : 53
  JULY : 64

 AVOCADO :
  JUNE : 75
  JULY : 85

LABOR_TOT : 5850
LABOR :
 COTTON : 2.9
 ONION : 2.7
 PEAR : 1.0
 AVOCADO : 1.5

PROFIT:
 COTTON : 6453
 ONION : 6110
 PEAR : 4814
 AVOCADO : 8813

CEIL :
 COTTON : 2000
 ONION : 250
 PEAR : 500
 AVOCADO : 800
```

Abstract model:

```
SUM [CROP: CROP?]  ≤ LAND
                        (overall land constraint)

SUM [FIELD: FIELD?] ≤ FIELD_LAND
                        (land constraint for field crops)

SUM [CROP: WATER.CROP.MONTH × CROP?]
   ≤ WATER_BND.MONTH
                        (monthly water constraint)

SUM [CROP: LABOR.CROP × CROP?]
   ≤ LABOR_TOT
                        (labor constraint)

   CROP? ≤ CEIL.CROP
                        (production ceiling for each crop)

MAXIMIZE [CROP: PROFIT.CROP × CROP?]
                        (objective function)
```

This simplified model will create a matrix with four columns and 10 rows. If there were 20 crops, over 12 months, the matrix would have 20 columns and 35 rows.

CITED REFERENCES

1. S. I. Gass, *Linear Programming*, McGraw-Hill Book Co., Inc., New York (1974).
2. W. Orchard-Hayes, *Advanced Linear Programming Computing Techniques*, McGraw-Hill Book Co., Inc., New York (1968).
3. *IBM APL Statistical Library*, SH20-1841-1, IBM Corporation; available through IBM branch offices.
4. L. Slate and K. Spielberg, "The Extended Control Language of MPSX/370 and possible applications," *IBM Systems Journal* **17,** No. 1, 64–81 (1978).
5. *IBM MGRW, Matrix Generator and Report Writer, Primer*, GH19-5042-1, IBM Corporation; available through IBM branch offices.
6. R. Bonczek, C. Holsapple, and A. Whinston, "Mathematical programming within the context of a generalized data base management system," *R.A.I.R.O. Operations Research* **12,** No. 2, 117–139 (1978).
7. C. Cohen and J. Stein, *Multipurpose Optimization System, User's Guide*, Manual No. 320, Vogelback Computing Center, Northwestern University, Evanston, IL (1975).
8. R. Mills, R. Fetter, and R. Averill, "A computer language for mathematical program formulation," *Decision Sciences* **8,** No. 2, 427–444 (1977).
9. R. Fourer and M. J. Harrison, *A Modern Approach to Computer Systems for Linear Programming*, Working Paper 988-78, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA (1978).

The authors are located at the IBM Israel Scientific Center, Computer Science Building, Technion City, Haifa, Israel.

Reprint Order No. G321-5135.