MANAGEMENT SCIENCE
Vol. 38, No. 3, March 1992
*Printed in U.S.A.*

# INDEXING IN MODELING LANGUAGES FOR MATHEMATICAL PROGRAMMING*

ARTHUR M. GEOFFRION
*Anderson Graduate School of Management, University of California, Los Angeles, California* 90024

Indexing structures are of fundamental importance to modeling languages for mathematical programming as a device for mathematical abstraction, and because they facilitate achieving conciseness, stability, and error-resistance. The aim of this article is to stimulate discussion of such structures, especially the two most common kinds found in algebraic style languages: sets and relations. We offer a taxonomy of set-based and relation-based indexing structures, a suite of detailed examples illustrating this taxonomy, and a number of specific principles (some arguable and some not) for incorporating indexing structures into modeling languages. We also examine four modeling languages in detail with respect to their indexing capabilities: AMPL, GAMS, LINGO, and SML. By attempting to work all of the illustrative examples in each language, we are able to reach some conclusions concerning relative expressive power, economy of notation, obedience to our principles of "good" language design, ease of data handling, and other criteria.
(MODELING LANGUAGE; MODELING SYSTEMS; MATHEMATICAL PROGRAMMING; INDEXING STRUCTURES)

## 1. Introduction

Any algebraic modeling language aimed at mathematical programming applications of realistic size and complexity must support the indexing of constants, variables, expressions, and constraints. Indexing plays a critical role in language design, in the design of modeling language translators that produce optimizer-ready data files, and in the way users think about how to express applications in a particular language. Surprisingly, the rapid proliferation of modeling languages in recent years has brought little if any convergence with respect to indexing features or notation, discussion of the pros and cons of different design options, or comparative language studies. Bisschop and Kuip (1991a, b, c), are among the few papers in this vein. See also Chapter 3 of Maturana (1990), and Chapter 1 and Appendix A of Witzgall and McClain (1985).

The time has come to take a close look at exactly what one means by "indexing", to develop normative guidelines for its use, and to make a comparative analysis of modeling languages from the point of view of these guidelines.

We begin by developing in some detail "indexing structures" based on sets and relations. These are by far the most common kinds of indexing structures used by current languages. This culminates in a reasonably comprehensive taxonomy that is the basis for the remainder of the article. We illustrate it with a running example woven into the text, and provide normative guidelines for its use in the context of algebraic mathematical programming languages.

In the interest of keeping the focus on fundamentals, we largely avoid discussing and illustrating most of the mathematical uses to which indexing structures typically are put. In particular, we do not discuss mathematical operators, like summation, that commonly incorporate indexing, although the design of such operators should benefit from a deeper understanding of how indexing structures can and should be used. One happy by-product of confining attention to fundamentals is that this article is not limited to linear programming, or even to models intended for optimization. It is pertinent to a very broad

* Accepted by Thomas M. Liebling; received February 28, 1990. This paper has been with the author 7 months for 2 revisions.

0025-1909/92/3803/0325$01.25

class of models for economic, engineering, information systems, MS/OR, and other applications, although we shall not pursue the broader implications here.

Once the taxonomy of common indexing structures is complete and normative guidelines are in place, we use them as the basis for a comparative analysis of several contemporary model definition languages for mathematical programming: AMPL (Fourer, Gay, and Kernighan 1990), GAMS (Brooke, Kendrick, and Meeraus 1988), LINGO (Cunningham and Schrage 1990), and SML (Geoffrion 1990b, 1991b, 1992). AMPL is chosen for its modern design and possible future commercial importance for AT&T's mathematical programming systems. GAMS is chosen for its maturity, popularity, and venerable status as a pioneering modeling language. LINGO is chosen for the attention it is likely to receive as a companion to LINDO, which may be the most widely distributed of all linear programming packages. SML, whose intended scope goes far beyond mathematical programming applications, is chosen because of the author's interest in better understanding an important aspect of its design.

A more complete version of this article (Geoffrion 1991a), which is available from the author upon request, presents 25 small, formal examples spanning the taxonomy of indexing structures. Slightly earlier versions of these examples also appear in Geoffrion (1990a). All 25 examples have been attempted in all 4 languages, as documented in detail in Geoffrion (1991a). The results have been used to evaluate these languages in terms of relative expressive power, economy of notation, obedience to certain principles of "good" language design, ease of data handling, and other criteria. We present these results here.

It is our hope that others will undertake to work the suite of 25 formal examples in other languages, thereby deepening the understanding of modeling languages and adding to the store of materials supporting comparative language analysis. Such analysis is sorely needed, given the rapid pace at which new languages are being designed and offered in the marketplace. Indeed, as of this writing, work of this sort has already been completed or is under way for five additional languages.

We now elaborate on the concept and importance of indexing as a notational device for representing general structure, and on the concept and importance of general structure itself. This introduction then concludes with a summary of the organization of the balance of the paper.

Exhibit 1 presents part of a larger model used as a running example throughout much of this paper. It follows the common practice of describing a mathematical programming model—actually a model class—in five parts: indices, constants, variables, constraints, and objective function. We address primarily the first of these parts, which therefore is given in much greater detail than the other parts. Exhibit 1 is written in an ad hoc version of ordinary algebra rather than in any particular modeling language. As we shall see, it exemplifies bad as well as good modeling practice.

### *Importance of Indexing*

An obvious reason for the importance of indexing is that it is accepted universally as a standard part of ordinary algebraic notation, which is the notation in which most mathematical programming applications are first made rigorous. Consequently, it is supported in one form or another by the overwhelming majority of model definition languages for mathematical programming.

The ubiquitous use of indexing in ordinary algebraic notation is not an accident. It results from one of the greatest of all mathematical inventions, namely the use of symbols as abstractions for specific numbers, strings, and other mathematical objects. An index is a special kind of symbol that represents members of a set. Whereas nonindexed symbols enable parts of a mathematical model to be "value independent," indices and indexed

EXHIBIT 1

*Partial General Structure of a Production/Distribution Model*

| | |
|---|---|
| *Indices, Sets, and Relations* | |
| $w \in W$ | Warehouses [*W* contains between 10 and 12 members] |
| $p \in P$ | Products, ordered by date of introduction (thus, P is an ordered set) |
| $f \in F = \{``h", ``n"\}$ | Factories, where "*h*" denotes the original factory and "*n*" denotes the new one (notice the lack of dimension independence) |
| $W1 \subseteq W$ | Full-line warehouses |
| $W2 = W - W1$ | Partial-line warehouses |
| $W3 \subseteq W$ | Co-located warehouses (adjacent to a factory) |
| $W4 = W1 \cup W3$ | Warehouses that are either full-line or co-located |
| $W5 = W1 \cap W3$ | Warehouses that are both full-line and co-located |
| $P1 \subseteq P$ | Promotional products [P1 contains either 2 or 3 members] |
| P_OLDEST = $\{p \in P: \text{ord}(p) = 1\}$ | Oldest product |
| $W1P = W1 \times P$ | Stocking matrix for full-line warehouses |
| $W2P \subseteq W2 \times P$ | Stocking matrix for partial-line warehouses [W2P contains at least one (w,p) tuple for every $w \in (W - W1)$] |
| $FP \subseteq F \times P$ | Production possibilities for the factories |
| FP_HOME = $\{(f,p) \in FP: f = ``h"\}$ | Home factory production possibilities |
| W2PF = W2P*FP | Replenishment lanes for the partial-line warehouses |
| FP_OK = $\{(f,p) \in FP: PCOST_{f'p} \leq$ 1.2 Min $\{PCOST_{fp}$ over $(f', p) \in FP\}\}$ | Acceptable production possibilities |
| ⋮ | |
| *Constants* | |
| $PCOST_{fp}$ for $(f,p) \in FP$ | Unit production cost (\$/ton) |
| $PLIM_{fp}$ for $(f,p) \in$ FP_OK | Production limit (tons) |
| ⋮ | |
| *Variables* | |
| $PROD_{fp}$ for $(f,p) \in$ FP_OK | Production quantity (pounds) |
| ⋮ | |
| *Constraints* | |
| $PROD_{fp} \leq 2000\ PLIM_{fp}$ for $(f,p) \in$ FP_OK | Production limit constraints |
| ⋮ | |
| *Objective* (to be minimized) | |
| $\sum_{(f,p) \in \text{FP\_OK}} PCOST_{fp}\ PROD_{fp} + \cdots$ | |
| ⋮ | |

([ ] denotes additional definitional restrictions, * denotes the natural join operation, ord ( ) denotes the ordinal position of its index argument in the corresponding primitive index set.)

symbols enable them to be "dimension independent" ("population independent" would be a more accurate term, but may not be as suggestive).

A model fragment is represented in a value independent way if it is represented symbolically in such a manner that specific values are essentially absent for all of its value-bearing mathematical objects, although some objects may incorporate a rule for calculating their value. This pertains to values of all types: numeric, string, truth, etc. The partial model given in Exhibit 1 is value independent. But it would fail to be so if, for example, specific numerical values were given for some of the unit production costs.

A model fragment is represented in a dimension independent way if membership details are essentially absent for all cohesive groups of entities, constants, and other mathematical objects; that is, the identities of the particular mathematical objects in a group are not given, although rules governing membership may be. "Cohesive group" means a collection in which all members play a similar mathematical role.

The partial model given in Exhibit 1 is dimension independent except for index set F, whose two members are specified. One may wonder also about index set W, whose cardinality is bounded above and below, but it is dimension independent by the above definition because the identity of its members is not specified. The cohesive groups evident in Exhibit 1 are: W, P, F, W1-W5, P1, P_OLDEST, W1P, W2P, FP, FP_HOME, W2PF, FP_OK, PCOST, PLIM, PROD, the production limit constraints, and the objective function.

These two types of independence lift the burden of having to be specific about aspects that are relatively unimportant for many purposes.

Thus, indices and other symbols are essential representational devices for achieving mathematical abstraction. They enable working with an entire *class* of models of interest rather than with just one model instance that is specific as to all values and other details. They enable concentration on general structure without distracting detail.

*Importance of General Structure*

The above discussion views general structure as a class of specific model instances at a level of abstraction just high enough to achieve the value independence and dimension independence of all (or nearly all) model parts. Exhibit 1 gives part of a general structure. Appendix 1 of Geoffrion (1990a) or (1991a) gives many fully detailed, self-contained examples of general structure. It is worth reviewing some of the reasons why this concept, which depends to such a great extent on indexing structures, plays such a central role in mathematical programming.

The root reason is that most practical applications of mathematical programming involve many more than one model instance. Usually they all fall within what can usefully be viewed as a single model class. Although only model instances can be optimized, the natural focus of most model-related work is the model class, i.e., the general structure of all (or nearly all) pertinent model instances. When a modeling activity does require information about specific values or dimensions, such information can be supplied and understood in the context of the general structure. See, for example, the "databases" of Appendix 1 of Geoffrion (1990a) or (1991a).

There are at least three other reasons for the importance of general structure: it provides conciseness, stability, and the opportunity to avoid many kinds of errors.

General structure usually is much more concise than any of its model instances. It may fit on a page or two, whereas a model instance may require many pages. It sweeps away the confusing clutter of inessential detail.

General structure usually is far less volatile than its model instances. The product line can change, unit production costs can change, and so on, without any impact at all on the general structure of a well-designed model.

Conciseness and stability render a general structure much more useful for many purposes than any of the model instances that it represents. They

- facilitate mathematical analysis, including deciding on amenability to solution (e.g., optimization) by any particular solver
- facilitate auditing and verification
- facilitate understanding and communication with other modeling professionals and model sponsors
- enable a general structure and all work done in connection with it to be reused for multiple specific model instances.

Error resistance is important because real mathematical programming models—especially large ones—are notoriously error prone. But if the modeler is able to give sharp definition to the model class of interest, and if the modeling system is able to check whether model instances fall within the specified class, then more errors will be detected than if the model class of interest is less sharply defined. The crucial role that indexing plays in this regard has been recognized by Bisschop (1988), among others.

*Organization*

§2 explains the indexing structures to be treated in this article, namely primitive sets, derived sets, and relations. Within each of these, there is a basic option whether to specify entirely by formula or to permit user discretion. The most important kinds of formulas are discussed, and the main outcome is a taxonomy of indexing structures. Geoffrion (1990a) contains a slightly abridged version of this material.

§3 takes up some issues in the design of algebraic modeling languages. We argue for the presence of certain index-related features in modeling languages, and against the presence of others. Our positions are cast in the form of a number of principles, called *tenets* and *rules*, some of which are controversial.

§4 summarizes how four modeling languages—AMPL, GAMS, LINGO, and SML—can or cannot express each of the 25 formal examples of Geoffrion (1990a) or (1991a). Complete details are provided in Geoffrion (1991a).

The final section presents some conclusions concerning the absolute and relative success of each of the four languages studied. It also offers some suggestions for the modeling language research and development community.

## 2. A Taxonomy of Indexing Structures

As explained in §1, an indexing structure is an abstraction that facilitates representing a group of similar mathematical objects, exclusive of any values or other internal details that these objects might have. Indexing structures play a critical role in representing general structure as this term is defined in §1. They admit (usually) many possible specific instances, each of which gives full details about a group of mathematical objects.

Exhibit 2 summarizes the proposed taxonomy of indexing structures. We must now rationalize this taxonomy.

There are three major categories: primitive sets, derived sets, and relations. Sets and relations are by far the most commonly used kinds of indexing structures, although occasionally one encounters others, such as trees (Bisschop and Kuip 1991b, Hürlimann 1987), which will not be discussed here.

The term "set" is used in its standard mathematical sense, and so is an unordered collection of distinct objects. Without loss of generality, we assume henceforth that the objects constituting a set are distinct *identifiers* representing (i.e., in 1:1 correspondence with) external entities of some sort. Such identifiers need only be nominal in character, although some languages allow them to be more than nominal. In Exhibit 1, the sets are W, P, F, W1–W5, P1, and P_OLDEST.

Some modeling languages represent a set as an ordered collection of identifiers. Order is not necessary so long as identifiers are distinct, but it does add to the expressive power of sets as indexing structures (e.g., time periods are naturally ordered) and it does enable certain useful kinds of access mechanisms (e.g., taking predecessors of the index for a set of time periods enables time-lagged access). Certainly there is no loss of generality when an unordered collection is represented by an ordered one. In Exhibit 1, P is an ordered set.

The distinction between a primitive set and a derived one is that the latter is defined in terms of other (primitive or derived) sets—for example, by subsetting—whereas the former is defined ab initio. A derived set can be viewed as a special kind of relation, but we choose to retain the concept of a derived set because it is such an important and common special case. In Exhibit 1, the primitive sets are W, P, and F; the others are derived.

The term "relation" also is used in its standard mathematical sense; an $n$-ary relation on sets $S_1, \ldots, S_n$ is an unordered set of ordered $n$-tuples drawn from the $n$-fold Cartesian product $S_1 \times \cdots \times S_n$. Most of the relations used in this article are binary ($n = 2$).

EXHIBIT 2

*Taxonomy of Indexing Structures*

- PRIMITIVE SET
  - Define by "Formula"
  - User Input
    - Unqualified
    - Qualified
      - Bounded Set Size
      - Other
- DERIVED SET (unary relation)
  - Define by Formula
    - Elementary Set Operations
      - Difference
      - Union
      - Selection
    - Compound Set Operations
      - Intersection
      - Other
    - Ordinal Selection
    - Value-Driven Set Membership
    - Other
  - User Selection
    - Unqualified
      - From Previously Defined Set
      - From Newly Defined Set
    - Qualified
      - Bounded Set Size
      - Other
- RELATION
  - Define by Formula
    - Elementary Relational Algebra Operations
      - Cartesian Product
      - Difference
      - Union
      - Projection
      - Selection
    - Compound Relational Algebra Operations
      - Intersection
      - Natural Join
      - Other
    - Transitive Closure
    - Ordinal Selection
    - Ordinal Selection with Offsets
    - Value-Driven Set Membership
    - Other
  - User Selection
    - Unqualified
      - From Previously Defined Relation
      - From Newly Defined Relation
    - Qualified
      - Function
      - Inclusive
      - Irreflexive
      - Other
- OTHER

Each part of an $n$-tuple is called a "component". In Exhibit 1, the relations are W1P, W2P, FP, FP_HOME, W2PF, and FP_OK.

We view all relations as being derived from primitive sets, derived sets, and/or other relations. There is no need for the concept of a "primitive relation," since a unary primitive relation can be viewed as a primitive set, and an $n$-ary primitive relation with $n > 1$ can be viewed as being constructed from the Cartesian product of primitive sets.

It follows from our assumption concerning primitive sets and from the ways in which derived sets and relations are constructed that the members of a derived set are always identifiers first introduced by a primitive set, and that the same is true of each component of each tuple of a relation. Derived sets and relations can thereby inherit order from primitive sets (none of the modeling languages known to this writer use any other kind of order).

Within the three major categories of indexing structures, the taxonomy provides for two fundamentally different options:

(A) define by "formula" (in the broadest sense of the word; e.g., W2 and FP_HOME of Exhibit 1)

(B) user input or selection (e.g., W and W1 of Exhibit 1).

The first option fully specifies a particular indexing structure as part of a model's general structure, while the second specifies it only partially within general model structure. The second requires further specification as part of the detailed data by which a specific model instance is determined, while the first does not.

The types of formulas which are useful depend on the kind of indexing structure. We consider the most basic types of formula for each kind in turn. Then we take up option (B).

*Primitive Sets Defined by Formula*

There are a few types of commonly occurring primitive sets (identifiers included) which some modeling language designers have thought worth building into their languages via keywords or simple formulas.

For example, $1..N$ and $1{:}N$, where $N$ is an integer, are popular ways to denote the first $N$ positive integers as the identifiers of a primitive set.

Primitive index set F of Exhibit 1 is defined by enumeration, which can be viewed as a particularly simple kind of formula.

*Derived Sets Defined by Formula: Elementary and Compound Set Operations*

The most common way to define a set in terms of other sets is via binary operations like *difference*, *union*, and *intersection*. The first two are elementary operations, while the third can be defined in terms of the first. There is also an important elementary set operator that is unary rather than binary, that is, it operates on just one set. Called *selection*, we do not give a separate discussion here because it is a special case of the relation algebraic operator of the same name discussed in detail below. For a similar reason, we do not discuss here *value-driven set membership*.

See W2, W4, and W5 in Exhibit 1 for examples.

*Relations Defined by Formula: Relational Algebra*

The mathematics of defining relations by formula is well developed, largely as a result of the ascension of relational database theory. We have a choice of at least three main notational systems: relational algebra, tuple relational calculus, and domain relational calculus (e.g., Ullman 1982). It is a beautiful result that all three systems, different as they appear to be, are equivalent in expressive power.

We choose relational algebra in this article as the main notation for defining relations by formula. Relational algebra has five elementary operations: *Cartesian product*, *difference*, *union*, *projection*, and *selection*. They are independent in that none can be compounded from the other four. We assume that the reader is familiar with these operations (e.g., Ullman 1982). Among the important compound operations are *inter-*

*section* and *natural join*. The latter is built by composing Cartesian product, selection, and projection (e.g., p. 155 of Ullman 1982).

In Exhibit 1, W1P is defined by a Cartesian product, FP_HOME by selection, and W2PF by a natural join.

There are also formulas that cannot be represented equivalently by composing elementary operations. A well-known case is *transitive closure*.

Another possibility is *value-driven set membership*, which is commonly used by some languages but dangerous because it violates strong dimension independence (to be explained in §3). Unlike the other definitions by formula, it involves values other than identifier values. Exhibit 1's FP_OK provides an example. Neustadter (1989) discusses the merits and demerits of this kind of formula in detail.

## *An Ordinal Version of Relation Algebraic Selection*

Selection accepts or rejects particular tuples from a relation by a formula that evaluates to true or false for each tuple. In the context of indexing structures, the formula involves at least one comparison operator ("=", "< >", "<", "≤", ">", "≥") with constant or identifier operands, and such comparisons may be compounded using logical operators (AND, OR, NOT).

Since primitive sets can be ordered without loss of generality, and especially since many modeling languages presume them to be so, it is possible and desirable to add an additional kind of expressive power to selection formulas: allow all kinds of comparisons based on ordinal identifier position (instead of identifier value) in the primitive set that defines it. This can be done using notation that is very similar to what is commonly used in relation algebraic selection formulas. We call this the *ordinal selection* operation. Naturally, one may not compare the ordinal positions of identifiers defined by different primitive sets.

To be more precise about the definition of ordinal selection, one can say that all identifiers are effectively aliased to the integers corresponding to their ordinal positions in the primitive sets that originally define them. (Equivalently, for the purposes of ordinal selection, all primitive sets are viewed as having the consecutive positive integers starting with unity as aliases for their identifiers.) These aliases are used in place of actual identifiers in all comparisons of a selection formula, and the only constants allowed in a comparison are integers corresponding to ordinal identifier positions. For every comparison, both operands must refer to the same primitive set. The compounding of comparisons using logical operators is allowed as in ordinary selection.

Thus, ordinary selection is supplemented for derived sets and relations by replacing each identifier by its ordinal position in its defining primitive set, and by interpreting all constants as references to ordinal position. Exhibit 1's P_OLDEST provides an example.

There is a more expressive version of ordinal selection that we call *ordinal selection with offsets*. Offsets means that addition and subtraction of integers representing ordinal displacement are permitted. For example, if $i$ and $j$ are alias indices, then $i = j - 2$ or ord $(i) =$ ord $(j) - 2$ would mean that $i$ is the predecessor of the predecessor of $j$ in the defining primitive set. One should be very careful when using an offset, as it may require the ordering over a primitive set to have stronger properties than meet the eye; in terms of measurement theory, graduating from ordinal selection without offsets to ordinal selection with offsets is like graduating from an *ordinal scale* to a *difference scale* (cf. Clemence 1990).

## *User Selection*

Now consider the user-input-or-selection option rather than the define-by-formula option.

For the *Primitive Set/User Input* branch, the most common alternative is to input a primitive set explicitly, identifier by identifier. This can be *unqualified*, that is, with no constraints whatever; or it can be *qualified*, such as having to observe a limit on the number of set members. In Exhibit 1, *P* and *W*, respectively, represent these two possibilities.

The *Derived Sets/User Selection* branch also can be divided into *unqualified* and *qualified* options. We distinguish two suboptions of the unqualified option according to whether an unconstrained selection is made

(a) *From a Previously Defined Set* (e.g., W1 in Exhibit 1), or

(b) *From a Newly Defined Set* that is defined by formula specifically for the purpose of providing the menu for user selection. Usually the formula defines as tight a superset of the desired set as can be expressed conveniently.

A "previously defined set" can be any primitive or derived set, and the formula defining a "newly defined set" can incorporate any previously defined primitive or derived set. For the qualified option, suboptions can be defined according to the nature of the constraints that must be observed. One possibility is the *bounded set size* option (e.g., P1 in Exhibit 1).

We adopt a similar development of the *Relation/User Selection* branch. It divides into *unqualified* and *qualified* options, and the first of these subdivides further into *From a Previously Defined Relation* and *From a Newly Defined Relation* (e.g., FP in Exhibit 1), where "relation" includes sets as well as relations of arity greater than one. Some of the suboptions under the qualified option are *inclusive* (e.g., W2P in Exhibit 1), which requires that the selection must include certain designated members; *function*, which is a common special kind of relation; and *irreflexive*, which is another common special kind of relation.

This completes the discussion of Exhibit 2.

### *Illustrative Examples*

The taxonomy of Exhibit 2 has been partially illustrated by reference to Exhibit 1. It is illustrated much more systematically, in fact completely, by a suite of 25 formal examples given in Appendix 1 of Geoffrion (1990a) or (1991a). In each case, the general structure is presented, in plain but careful language, separately from a database that instantiates it. A second database is given in most cases that violates the general structure, as a test of the discriminating power of modeling languages.

Those examples were designed with these criteria in mind:

(1) span the taxonomy of indexing structures

(2) be simple yet obviously of practical relevance

(3) exercise the main indexing capabilities of the model definition languages of interest

(4) be nonredundant, but build on one another when possible

(5) bring out clearly the important points requiring discussion in a thorough study of indexing (e.g., six of them have general structures claimed in §3 to be so undesirable that no "good" modeling language should be able to represent them).

## 3. Issues in the Design of Modeling Languages

This section considers some of the design issues that arise for mathematical programming modeling languages in connection with indexing structures. Partly to express our views and partly to provoke discussion, we take a strong position on most of these issues in the form of "tenets" (when arguable) and "rules" (when indisputable). One of the conclusions which emerges, which some will find surprising, is that ordinary algebra (e.g., Exhibit 1) obeys only one of the tenets and none of the rules unless these are incorporated as explicit semantic restrictions.

### *Support for General Structure*

The first issue has to do with the degree of support a language provides for general model structure as distinct from specific model instances. We consider this to be an issue related to indexing structures because, as explained in §1, indexing structures play a critically important role in making it possible to express the kinds of general structures that arise in practical mathematical programming models. Since general structure by itself (i.e., without any instantiating data) is so useful, again as explained in §1, the first basic tenet is:

Tenet 1. *A good modeling language should be able to represent the general structure of a class of models separately from any specific model instance.*

General structure does not specify particular values nor enumerate the populations of various cohesive groups of objects. Instantiating data supply all of this detail, preferably as a low-redundancy supplement to general structure.

Tenet 1 is universally accepted in the neighboring field of database theory (e.g., Ullman 1982), and is gaining favor in the mathematical programming community.

*Strong Dimension Independence*

At the core of our second issue is a stronger version of the dimension independence concept explained in §1. Strong dimension independence means that not only are the identities of specific objects unknown at the level of general structure for all cohesive groups of entities, constants, and other mathematical objects, but these identities do not depend on the particular values of the constants or other value-bearing mathematical objects that must be supplied to instantiate the general structure. In other words, *group membership details are independent of instantiating data values.*

When strong dimension independence does not hold, general structures and hence the indexing structures that help represent them are unstable to the degree that group membership-determining values are unstable. Since exogenously supplied values often are the most volatile aspect of applied mathematical programming models, such structures are often unstable and hence less useful than alternative structures that do obey strong dimension independence.

To illustrate this kind of instability, consider the case where a constant $K$ helps to determine the membership of an index set $S$ that is used to index a group $G$ of constants. If the value of $K$ is volatile, then $S$ will inherit this volatility and changes in $K$ will lead to the deletion and creation of members of $G$'s population. Should one throw away a $G$ value when it corresponds to a member that is deleted as a result of a change in $K$? Perhaps it will be needed later when a compensating change occurs in the value of $K$. How should one acquire a $G$ value when a new member of $G$ is created as a result of a change in $K$? This could be vexing once the data development phase of a modeling project is finished.

The instability is exacerbated when $K$ is a variable rather than a constant, for the whole point of making something a variable rather than a constant is that its value is subject to discretionary change, possibly under algorithmic control. Matters get still worse when $K$ is a function.

If $S$ indexes a group of variables or constraints whose population is rendered unstable by such dependencies, then no conventional algorithm would be able to optimize over them because the optimization problem itself would become unconventional or even ill-posed.

The reader may think that the exacerbation discussed in the last two paragraphs can be avoided by forbidding $K$ to be anything but a constant, but this is a hopeless tactic in any practical sense because virtually any "constant" can be viewed as a variable or even as a function if one takes a broad enough view of it. There is only a tenuous line between the roles of constants and variables. In real applications, it is common for a particular numerical quantity to play each of these roles at different times. As an old saying puts it, "One person's constant is another person's variable." If a constant is allowed to help determine group membership, then safeguards are required to stop someone from deciding to treat it as a decision variable, thereby destabilizing group membership to the point of uselessness.

These observations strongly suggest that it is inadvisable to formulate general structures containing a group whose membership depends on the value of any constant, variable, or other value-bearing mathematical object unless that value is part of the general structure itself. A second exception can be made for terminal groups of expressions or constraints—"terminal" in the sense that such a group is dedicated in purpose and cannot be used to determine the membership of any other group. Thus, a second basic tenet is

Tenet 2. *A good modeling language should not be able to represent a general structure that violates strong dimension independence, except possibly for terminal groups of expressions or constraints.*

The partial general structure of Exhibit 1 violates this tenet. The root of the problem is the value-driven index relation FP_OK: its membership is determined by the values of the PCOST constants. Consequently, the members of the PLIM group of constants, the PROD group of variables, and an associated group of constraints are all dependent on the values of the PCOST constants. In terms of the discussion of a few paragraphs ago, one may make these identifications:

$$K \leftrightarrow \text{PCOST},$$

$$S \leftrightarrow \text{FP\_OK},$$

$$G \leftrightarrow \text{PLIM}.$$

Note that changes in PCOST require changes to be made in PLIM, an inconvenience that clearly is an artifact of the chosen representation for general structure (there is no inherent connection between PCOST and PLIM). One might argue similarly that PCOST changes induce unnecessary changes in group membership for PROD and for the associated constraints, although it could be counterargued that this is the deliberate intention of introducing FP_OK in the first place (i.e., to ensure that production only occurs for factories that are within 20% of the least cost factory for each product).

The real problem with using FP_OK to index PROD and the production limit constraints is that it causes the general structure to be unnecessarily *fragile*. One way to reveal this fragility is to consider a general structure change that would transform PCOST into a variable or even a function. As pointed out earlier, constants and variables often change identities in the course of real applications, so there is nothing bizarre about such a change in general structure. Suppose that the production processes have an adjustment variable $V_{\text{fp}}$ that influences $\text{PCOST}_{\text{fp}}$. Then $\text{PCOST}_{\text{fp}}$ becomes a function of $V_{\text{fp}}$, say $\text{PCOST}_{\text{fp}}(V_{\text{fp}})$. Worse, FP_OK becomes a function of $V$, say FP_OK($V$), thereby becoming intractable as an indexing structure for PLIM, PROD, and the associated constraints.

The most obvious remedy for PLIM, PROD, and the associated constraints is to adopt a new relational indexing structure that is either FP itself, or some subset of FP that can be shown to contain FP_OK for all $V$ of possible interest; moreover, a new set of constraints will be required to guarantee $\text{PROD}_{\text{fp}} = 0$ for $(\text{f}, \text{p}) \notin$ FP_OK (these constraints can be written in conventional form with the help of auxiliary 0-1 variables).

Notice that the remedy involves major changes in general structure that are, once again, an artifact of having chosen (in Exhibit 1) a representation of general structure that violates strong dimension independence. These difficulties, as well as the inconvenient dependence of PLIM on PCOST, could have been avoided easily by using an equivalent representation of general structure that obeys strong dimension independence. Both representations would lead to identical optimization problems assuming that a suitably capable "presolve" routine is used, thereby deflating any counterargument to the effect that optimization performance would suffer as a consequence of abiding by strong dimension independence.

Tenet 2 is violated by AMPL, GAMS, LINGO, and other modeling languages known to this writer, and so must be viewed as controversial. The thrust of private communications with several language developers has been that the practical advantages of setting this tenet aside may outweigh the arguments in its favor. See Neustadter (1989) for further discussion of the pros and cons of Tenet 2.

Like Tenet 1, this one also evokes related ideas in database theory. It is suggestive of a draconian enforcement of some high normal form found in relational database normalization theory (e.g., Chapter 7 of Ullman 1982), which aims to establish principles of database design that preclude instabilities (update anomalies) resulting from data dependencies.

*Proper Use of Identifiers*

Identifiers, as mentioned in §2, need only be nominal. We believe that they should not be any more than that, at least within the context of purely set- and relation-based indexing structures, although this view is not universal among modeling language designers.

Tenet 3. *In a good modeling language, identifiers should be arbitrary and play a purely nominal role, that is, they should not have a value used for computational purposes other than identification. Moreover, in keeping with Tenet* 1, *particular identifiers should not appear explicitly in general structure.*

This tenet seeks to preserve the fundamental mission of set- and relation-based indexing structures as a purely organizational device that enables dimension independence. An identifier may be a string composed from some alphabet of characters, or be an integer, but the properties which one normally associates with such domains (e.g., order or cardinal value) should not be recognized properties of identifiers in the context of indexing structures; that kind of overloading would be, we believe, undesirable. Any identifier whose value matters should be replaced by a nominal identifier together with an indexed value-bearing attribute. To violate Tenet 3 is to confuse the role of an indexing structure with the things being indexed, and to open the door to violations of Tenets 1 and 2.

It is not inconsistent with Tenet 3 for an identifier to be referred to implicitly in general structure in terms of its ordinal position in its defining primitive set.

Exhibit 1 violates the second part of this tenet because particular identifiers are given for F, and because one of these appears in the definition of FP_HOME.

*Domain Integrity*

In order for set union and set difference operations to make sense, one must be able to compare an identifier in one set operand with an identifier in the other set operand. A notion of equality is needed. Unfortunately, the arbitrariness of primitive set domains raises a difficulty. For example, different abbreviations might be used for the same external entity in two different primitive sets, leading to the erroneous conclusion upon comparison that the identifiers stand for different entities; or particular identifiers in distinct sets might happen to be identical and yet stand for quite different entities.

A resolution of this difficulty is

Rule 1. *Every modeling language should prohibit set union, set difference, and compounds involving these (such as intersection) unless both set operands are drawn from the same domain.*

A "domain" is simply a parent set. Ordinary algebra, which Exhibit 1 uses in an ad hoc way, does not enforce this rule (or any of the others given subsequently); however, Exhibit 1 itself happens to use set difference, union, and intersection correctly. This reveals a shortcoming of ordinary algebra as a modeling language. For example, nothing prevents one from appending a derived set W ∪ F to Exhibit 1 even though this could easily produce nonsensical results.

Domain compatibility difficulties arise also for elementary and compound relational algebra operations. In fact, the difficulties are even greater because relational algebra (and also tuple relational calculus and domain relational calculus) permits particular identifiers and constants to be used in formulas and permits "<" and "≤" type comparisons of identifiers. We now examine the difficulties in some detail.

The nature of the difficulty for the union and difference operations is the same as for derived sets, and its resolution is the same.

*Rule 2.* *Every modeling language should prohibit taking the union or difference of two relations (of the same arity), and compounds involving these (such as intersection), unless corresponding pairs of components are drawn from the same domain.*

There is no difficulty at all for Cartesian product or for projection.

The selection operation is where the main difficulties occur. We intend the discussion which follows to apply not only to relations, but also to derived sets (which are unary relations).

As mentioned in §2, selection accepts or rejects particular tuples from a relation by a comparison-based formula that evaluates to true or false for each tuple. Such formulas make some presumptions about underlying structure:

(1) If an "=" or "< >" comparison is made between two components of a tuple, or between a component of a tuple and a constant, then it is presumed that both operands are drawn from the same domain.

(2) If a "<" or "≤" or ">" or "≥" comparison is made between two components of a tuple, or between a component of a tuple and a constant, then it is presumed that both operands are drawn from the same ordered domain.

The first presumption calls for a resolution similar to Rule 2, namely

*Rule 3.* *Every modeling language should prohibit "=" and "< >" comparisons in a selection formula unless both operands are drawn from the same domain.*

The second presumption calls for a slightly stronger resolution, namely

*Rule 4.* *Every modeling language should prohibit "<", "≤", ">", and "≥" comparisons in a selection formula unless both operands are drawn from the same ordered domain.*

Note that a domain's order could derive either from identifier value or from ordinal identifier position in the defining primitive set. For example, a defining primitive set {3, 4, 1} would yield the order $1 < 3 < 4$ under the first approach, but $3 < 4 < 1$ under the second. Relational algebra, which provides the context for this discussion, adopts the first approach.

It warrants emphasis that Rules 3 and 4 pertain to ordinary selection, and not to ordinal selection.

An important addendum to this discussion comes about because Tenet 3 covers some of the same ground as Rules 3 and 4. The key observations are that "<", "≤", ">", and "≥" comparisons imply a "computational role" for the identifiers involved, and that making an "=" or "< >" comparison with a constant implies that the constant actually is an identifier. Thus, Tenet 3 yields restrictions on the use of ordinary selection.

*Tenet 3A.* *To be consistent with Tenet 3, a modeling language must restrict ordinary selection formulas in the context of indexing structures to prohibit all "<", "≤", ">", and "≥" comparisons, and to prohibit "=" and "< >" involving constants.*

Note that the only comparisons which Tenet 3A permits in the context of indexing structures are "=" and "< >" when both operands are identifiers (recall that, by Rule 3, both operands must be from the same domain, so a notion of equality is available that is independent of the particular identifiers).

Notice also that Rule 4 will be satisfied in the context of indexing structures if the first part of Tenet 3A is.

Rules 1–4 apply not only to indexing structures, but also to any other mathematical expressions using elementary set operations or relational algebra operations that a modeling language may allow.

Rules 3–4 and Tenet 3A severely restrict the use of selection formulas in the context of indexing structures, as well they should. We comment that ordinal selection (described in §2) rehabilitates ordinary selection for derived sets and relations simply by replacing each identifier by its ordinal position in its defining primitive set, and by interpreting all constants as references to ordinal position. This is fully consistent with Tenet 3.

The last six examples of Appendix 1 of Geoffrion (1990a) or (1991a) illustrate all of the tenets and rules given in this section.

## 4. Applying the Four Languages to the Formal Examples

Appendices 2–5 of Geoffrion (1991a) work the suite of 25 formal examples in AMPL, GAMS, LINGO, and SML. We omit all details, and give here only a summary of the results.

The examples can be grouped into three categories according to their main purpose:

(A) test the *expressive power* of a language under the constraint that no tenet or rule of §3 may be violated

(B) test the degree of *tenet obedience* of a language with respect to the views expressed by the tenets of §3 (these are arguable)

(C) test the degree of safety of a language with respect to the *domain integrity* requirements expressed by the rules of §3 (these are not arguable).

There are 23 examples in category (A), 4 in category (B), and 2 in category (C) (the total number exceeds 25 because there are "corrected" and "preferred" versions that we count separately).

See Exhibit 3 for a summary of the testing on category (A) examples. The possible outcomes are as follows each time one of these examples is attempted with one of the languages:

| | |
|---|---|
| *Perfect* | The language can represent the general structure perfectly while honoring all tenets and rules. |
| *Superset* | The language cannot represent the general structure perfectly while honoring all tenets and rules, but it can represent a superset of the given model class. |
| *Relative* | The language cannot represent the general structure perfectly while honoring all tenets and rules, but it can represent a nonsuperset relative of the given model class. |

In cases where the outcome is not perfect, and where a database with mistakes is available, it is also of interest to note how many of the mistakes the language specified against. This information is summarized briefly in Exhibit 3 by "$n/m$", which means that $n$ out of $m$ mistakes are specified against.

A comment is in order concerning the superset entries of Exhibit 3. The basic idea is that, if a language's expressive power is not sufficient to allow representing a given model class exactly, then the next best alternative is to identify a tight superset and to represent

EXHIBIT 3

*Summary of Language Performance Relative to Expressive Power* (*Geoffrion* 1991*a*)

| EXAMPLE | AMPL | GAMS | LINGO | SML |
|---|---|---|---|---|
| 1 | Perfect | Perfect | Perfect | Perfect |
| 2 | Perfect | Perfect | Perfect | Perfect |
| 3 | Perfect | Perfect | Perfect | Perfect |
| 4 | Perfect | Perfect | Perfect | Perfect |
| 5 | Superset 1/3* | Perfect | Perfect | Perfect |
| 6 | Perfect | Perfect | Perfect | Perfect |
| 7 | Perfect | Perfect | Perfect | Perfect |
| 8 | Perfect | Perfect | Perfect | Perfect |
| 9 | Perfect | Perfect | Perfect | Perfect |
| 10 | Perfect | Perfect | Perfect | Perfect |
| 11 | Perfect | Perfect | Perfect | Perfect |
| 12 | Perfect | Perfect | Perfect | Perfect |
| 13 | Perfect | Perfect | Perfect | Perfect |
| 14 | Perfect | Perfect | Superset 0/2 | Perfect |
| 15 | Perfect | Perfect | Perfect | Perfect |
| 16 | Superset 1/3* | Perfect | Perfect | Perfect |
| 17 | Perfect | Perfect | Perfect | Perfect |
| 18 | Perfect | Perfect | Perfect | Perfect |
| 19 | Superset 1/2 | Relative 1/2 | Superset 2/2 | Superset 1/2 |
| 20 Preferred | Perfect | Superset | Relative | Perfect |
| 22 Preferred | Superset 0/2* | Perfect | Perfect | Perfect |
| 23 Preferred | Perfect | Perfect | Perfect | Perfect |
| 24 Corrected | Perfect | Perfect | Perfect | Perfect |

* AMPL enhancements made in the summer of 1990 enable reclassification to "Perfect".

that in the language instead. A superset is preferable to a subset, or to any set that omits some of the model instances in the given model class, because a superset does not forbid any of the model instances in the given model class. Moreover, the superset should be as tight as possible in order not to include any more illegal (with respect to general structure) model instances than necessary. Ideally, one seeks the tightest possible superset, and usually this superset is obvious.

Exhibit 4 abstractly depicts the situation just discussed. Suppose that the class of model instances associated with a given general structure corresponds to all points inside an ellipse, and that a given modeling language can represent only a model class whose model instances correspond to the points inside a rectangle. Then the problem of finding a tight superset is the problem of finding a small rectangle containing the ellipse.

Exhibit 4 also indicates two points, one inside the ellipse and the other outside. These represent the model instances associated with the two databases that are given for each example. The first yields a legitimate model instance, but the second does not because it includes one or more mistakes that violate the associated general structure.

See Exhibit 5 for a summary of the testing on category (B) examples. The possible outcomes are as follows each time one of these examples is attempted with one of the languages:

*Obey* The language cannot represent the general structure accurately even if all tenets are ignored.

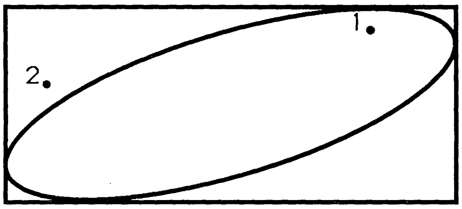
*Disobey* The language can represent the general structure accurately.

An "obey" outcome does not prove that a language always observes the tenet being tested. A "disobey" outcome, on the other hand, does prove that a language does not always obey the tenet being tested.

See Exhibit 6 for a summary of the testing on category (C) examples. The possible

EXHIBIT 4

*Abstract Portrayal of a Model Class (Ellipse) and a Superset Representation*



outcomes are as follows each time one of these examples is attempted with one of the languages:

*Safe* The language recognizes that the general structure is potentially ill-posed.

*Unsafe* The language fails to recognize that the general structure is potentially ill-posed.

## 5. Conclusion: How the Four Languages Compare

Some conclusions emerge from Exhibits 3, 5, and 6.

1. **Expressive Power** Exhibit 3 suggests that, when the languages are used in a manner consistent with all of the rules and tenets of §3, they rank as follows in terms of increasing expressive power with respect to general structure:

AMPL (19), LINGO (20), GAMS (21), SML (22).

The number in parentheses is the number of the 23 category (A) examples for which the result is "perfect." By looking at the pattern of nonperfect results in relation to the category tree, we can better understand the nature of the strengths and weaknesses of each language. AMPL's only weakness is in the area of ordinal selection, which is by design because it chooses to make its sets unordered. (*Note.* Enhancements to AMPL made during the summer of 1990 overcome this weakness, raising the score from 19 to 22.) LINGO lacks the ability to take transitive closures. GAMS' weaknesses are negligible. SML's one failure, shared by the other three languages, was on an index tree example (#19) which falls outside the scope of ordinary mathematical indexing with sets and tuples.

2. **Tenet Obedience** Exhibit 5 suggests that the languages rank as follows in terms of increasing obedience to Tenets 1–3 and 3A:

AMPL (0), LINGO (1), GAMS (2), and SML (4).

The number in parentheses is the number of the 4 category (B) examples for which the result is "Obey." What can be concluded in general about the four languages? We defer discussion of Tenet 1 to a later conclusion owing to its subtlety. AMPL disobeys Tenets

EXHIBIT 5

*Summary of Language Performance Relative to Tenet Obedience*
*(Geoffrion* 1991*a)*

| EXAMPLE | AMPL | GAMS | LINGO | SML |
|---|---|---|---|---|
| 20 | Disobey | Disobey | Disobey | Obey |
| 21 | Disobey | Obey | Disobey | Obey |
| 22 | Disobey | Obey | Obey | Obey |
| 23 | Disobey | Disobey | Disobey | Obey |

EXHIBIT 6

*Summary of Language Performance Relative to Domain Integrity (Geoffrion 1991a)*

| EXAMPLE | AMPL | GAMS | LINGO | SML |
|---|---|---|---|---|
| 24 | Unsafe | Safe | Unsafe | Safe |
| 25 | Unsafe | Safe | Unsafe | Safe |

2, 3, and 3A. For the other languages one must examine the structure of the languages themselves. LINGO also disobeys Tenets 2, 3, and 3A, but Tenet 3 is a near miss: it obeys the part of Tenet 3 about identifiers playing a purely nominal role, but one of the examples (#21) shows that it disobeys the part about precluding explicit reference to identifiers in the general structure. GAMS disobeys Tenet 2. But it obeys Tenets 3 and 3A in general unless a modeler uses the value-driven set membership construct (permissible under its rejection of Tenet 2) as a workaround. Clearly the author's own language, which always obeys all of the tenets, is at an unfair advantage here; the arguments leading to the tenets are susceptible to counterarguments by the designers of the other languages.

3. **Domain Integrity** Exhibit 6 suggests that the languages rank as follows in terms of increasing safety with respect to the domain integrity requirements stated in Rules 1–4:

AMPL/LINGO (0), GAMS/SML (2).

The number in parentheses is the number of the 2 category (C) examples for which the result is "Safe." Not only are GAMS and SML safe for these particular examples, but an examination of the structures of these languages also shows that they are safe in general. However, in the case of GAMS and LINGO, safety can be compromised for any of the rules by using the value-driven set membership construct condoned by these languages but condemned by Tenet 2.

Other conclusions emerge from looking at the details presented in Geoffrion (1991a) in more detail.

4. **Model Instances Are Easy** For category (A) examples, the languages seldom had any difficulty representing specific model instances; the major exception was the one example (#19) designed to illustrate a nonstandard indexing structure. Only LINGO had difficulty with any other specific model instances (namely, the preferred version of #20). The telling challenge for a modeling language is not whether it can represent model instances, but whether it can represent general structures of the type that arise in target applications.

5. **Schema Economy** Economy of schema notation is a common criterion for judging modeling languages. Readability considerations aside, more succinct notation is better than less succinct notation. A simple measure of notational economy is the average number of lines needed to represent the general structure for the category (A) examples, ignoring the fact that some languages achieve a perfect representation while others achieve only an imperfect one. The languages rank as follows by this measure, in order of increasing economy:

LINGO (6.7), GAMS (6.0), AMPL (4.3), and SML (4.0).

The number in parentheses is the average number of lines per example. For all examples in all languages, we have made an effort to make good use of a maximum line width of 65 characters, to use consistent naming conventions, and to achieve comparable standards of readability. We do not count more than two consecutive lines of comments. The developers of all four languages have had an opportunity to suggest improvements, and these have been adopted whenever appropriate.

6. **Schema Readability** Readability of schema notation is another common criterion for judging modeling languages. We refrain from offering any pronouncement as to how the languages compare by this criterion. This is not for lack of opinion, but rather because there appears to be no fair way to quantify the judgment. The organization of the appendices of Geoffrion (1991a) makes it easy for readers to form their own opinions by comparing all four languages side by side on identical examples.

7. **Ease of Data Handling** Turning now from schema to database, it is of interest to know how well the database lends itself to support by modern database technology. If one accepts the currently dominant relational paradigm, then all data should be in tables (relations), tables should have distinct names, columns within each table should have distinct names and definite domains, and the tables themselves should be normalized (Ullman 1982). Only SML was designed with this in mind and achieves it (Geoffrion 1990b), although the other languages probably can be adapted for use with a relational database system. The other languages do offer special features designed to economize on data entry when done by hand, and do exploit the visual simplicity of two-dimensional arrays.

8. **Structure / Data Separation** Tenet 1 deserves special discussion. The only examples that test it (#20 and #21) address whether it is possible to include specific identifiers in general structure. The results show that the answer is yes for AMPL, GAMS, and LINGO. The results also show that users have the option of deferring the mention of specific identifiers to the database. This raises the question of how strongly a language discourages the commingling of general structure and instantiating database. The answer is that the languages rank as follows, in order of least to greatest discouragement of commingling:

GAMS, LINGO, AMPL, SML.

GAMS actually requires commingling because its translator will not run, for example, unless all necessary data are given before each derived set or relation is declared. LINGO, a direct descendant of a language that totally commingles general structure and database, depends on its @FILE facility in order to separate general structure and database. This facility essentially provides the "include file" feature found in many standard programming languages. The approach is really one in which structure and data are thought of as being commingled, but can be separated if desired. AMPL's posture is strongly that structure and data ought not to be commingled, although exceptions are possible. SML's posture is the same, and no exceptions are possible.

The above observations and conclusions suggest some general points for language designers to keep in mind. Two are as follows.

- First, *pay less attention to providing language features that mimic the powers of conventional mathematical notation*. It is fine to provide modelers with features that they already know how to use, but avoid those which are dangerous because they are error-prone or lead to general structures that are less useful than those built from "safer" features. The consequences of excessive expressive power can be as adverse as the consequences of inadequate expressive power.
- Second, *pay more attention to providing language features that strengthen the distinction between general structure and model instance*. Almost any language can represent almost any model instance (not necessarily well). But it takes an exceptional language to represent a real life model class with accuracy, clarity, economy, and lack of unnecessary contamination by instance or solver details.

In closing, we reiterate that an avowed purpose of this article is to stimulate discussion of indexing structures. We hope that, over time, there will emerge:

(a) consensus on which indexing structure characteristics are good and bad in which contexts for algebraic mathematical programming modeling languages

(b) an improved taxonomy along the lines of Exhibit 2 that treats not only sets and relations, but also other useful kinds of indexing structures
(c) an enlarged set of formal examples like those of Geoffrion (1990a) or (1991a) that illustrate the improved taxonomy and the full indexing power of all leading languages
(d) solution sets like those of Appendices 2–5 of Geoffrion (1991a) for the enlarged set of examples, and also similar solution sets for other significant languages (the development of such solution sets is already taking place: for Lisp by Lin and Ramirez 1990, for LSM by Krishnan 1990, for MPL by Kristjansson 1990, for SQL by Ramirez 1990, and for ULP by Witzgall 1990)
(e) standards for indexing structure design and notation that offer improved functionality, greater error resistance, and fewer unnecessary obstacles to learning and use
(f) comparative language analyses, along the lines of the early part of this section, that will be useful to evaluators of the languages of today and to designers of the languages of tomorrow.[1]

[1] Partially supported by the National Science Foundation, the Office of Naval Research, Ketron Management Science, Inc., and Shell Development Company. The views expressed are those of the author and not of the sponsor.

I am pleased to acknowledge the helpful comments and assistance provided by Bob Fourer, Tony Hürlimann, Sergio Maturana, Alex Meeraus, Laurel Neustadter, Yao-Chuan Tsai, and Fernando Vicuña. In addition, I appreciate David Gay's help with AMPL and Linus Schrage's help with LINGO.

## References

BISSCHOP, J., "Language Requirements for A Priori Error Checking and Model Reduction in Large-Scale Programming," In G. Mitra (Ed.), *Mathematical Models for Decision Support*, NATO ASI Series F, Springer-Verlag, Berlin, 1988, 171–181.

——— AND C. KUIP, "Representation of Time in Mathematical Programming Modeling Languages," Dept. of Applied Mathematics, University of Twente, The Netherlands, February 1991a, 13 pp.

——— AND ———, "Hierarchical Sets in Mathematical Programming Modeling Languages," draft paper, Dept. of Applied Mathematics, University of Twente, The Netherlands, April 1991b, 19 pp.

——— AND ———, "Compound Sets in Mathematical Programming Modeling Languages," Dept. of Applied Mathematics, University of Twente, The Netherlands, June 1991c, 15 pp.

BROOKE, A., D. KENDRICK AND A. MEERAUS, *GAMS: A User's Guide*, The Scientific Press, Redwood City, CA, 1988.

CLEMENCE, R. D., JR., "A Type Calculus for Mathematical Programming Modeling Languages," Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, 1990.

CUNNINGHAM, K. AND L. SCHRAGE, "The LINGO Modeling Language," University of Chicago, September 1990, 113 pp.

FOURER, R., D. GAY AND B. KERNIGHAN, "AMPL: A Mathematical Programming Language," *Management Sci.*, 36, 5 (May 1990), 519–554.

GEOFFRION, A., "A Taxonomy of Indexing Structures for Mathematical Programming Modeling Languages," *Proc. Twenty-Third Annual Hawaii Internat. Conf. System Sciences. Vol.* III, (held in Kailua-Kona, January 2–5), IEEE Computer Society Press, Washington, 1990a, 463–473.

———, "SML: A Model Definition Language for Structured Modeling," Working Paper 360, Western Management Science Institute, UCLA, revised August 1990b, 129 pp.

———, "Indexing in Modeling Languages for Mathematical Programming," Western Management Science Institute, Working Paper 371, UCLA, revised July 1991a, 180 pp.

———, "FW/SM: A Prototype Structured Modeling Environment," *Management Sci.*, 37, 12 (December 1991b), 1513–1538.

———, "The SML Language for Structured Modeling," *Oper. Res.*, 40, 1 (January–February 1992).

———, S. MATURANA, L. NEUSTADTER, Y. TSAI AND F. VICUÑA, "User Documentation for FW/SM Release X90-09," Anderson Graduate School of Management, UCLA, December 1990, 100 pp.

HÜRLIMANN, T., *LPL: A Structured Language for Modeling Linear Programs*. Vol. 865, Ser. V, Peter Lang, Bern, Switzerland, 1987. (Later versions of LPL exist.)

KRISHNAN, R., "Indexing Examples in LSM/L↑," Informal Note, Decision Systems Research Institute, Carnegie Mellon University, November 1990, 33 pp.

KRISTJANSSON, B., Personal communication, Maximal Software, Arlington, VA, October 1990.

LIN, E. AND R. RAMIREZ, "Indexing in Modeling Languages for Mathematical Programming: A LISP Implementation of Indexing Structures," Dept. of Decision and Information Systems, Arizona State University, September 1990, 33 pp.

MATURANA, S., "A Translator Writing System for Algebraic Modeling Languages," Ph.D. Dissertation, Anderson Graduate School of Management, UCLA, 1990.

NEUSTADTER, L., "Value-Driven Sets in Modeling Languages: An Analysis," Anderson Graduate School of Management, UCLA, October 1989, 19 pp.

RAMIREZ, R., "Indexing in Modeling Languages for Mathematical Programming: A SQL Implementation of Indexing Structures Using ORACLE," ASUMMS Informal Note No. 4, Dept. of Decision and Information Systems, Arizona State University, November 1990, 35 pp.

ULLMAN, J. D., *Principles of Database Systems*, (2nd Ed.), Computer Science Press, Rockville, MD, 1982.

WITZGALL, C., Personal communication, National Institute of Standards and Technology, Gaithersburg, MD, May 1990.

——— AND M. MCCLAIN, "Problem and Data Specification for Linear Programs," NBSIR 85-3125, U.S. Department of Commerce, National Bureau of Standards, National Engineering Laboratory, Gaithersburg, MD, April 1985, 113 pp.