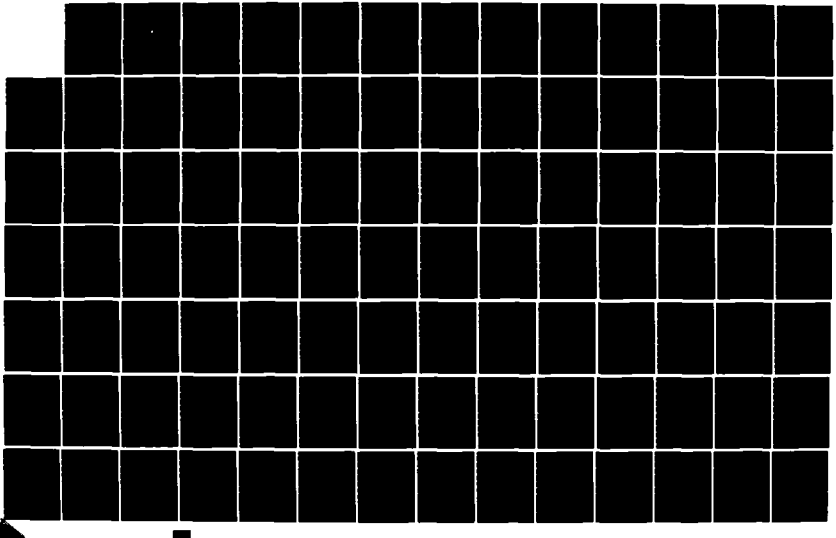
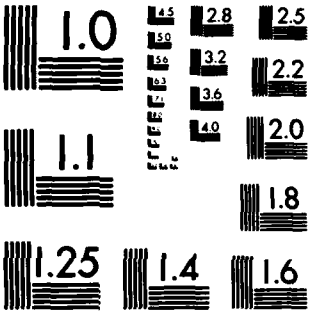
AD-A150 536 LEXICON: A STRUCTURED MODELING SYSTEM FOR OPTIMIZATION (U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R D CLEMENCE JUN 84 1/2

UNCLASSIFIED F/G 9/2 NL





NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

LEXICON: A STRUCTURED MODELING SYSTEM FOR OPTIMIZATION

by

Robert D. Clemence Jr.

June 1984

Thesis Advisor: G. Bradley

Approved for public release; distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A150536 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LEXICON: A Structured Modeling System for Optimization	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1984
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert D. Clemence Jr.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943	12. REPORT DATE June 1984
	13. NUMBER OF PAGES 97
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Linear Programming, Matrix Generators, Modeling Languages, Problem Generators, Structured Modeling

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Linear Programming (LP) is used infrequently for routine decision-making. Even in situations where LP is an extremely attractive tool, there is too much cost, frustration, delay and risk incurred in conversion of a mathematical hypothesis into a valid LP solution. This report outlines an entirely new approach to specifying and generating LP's which departs fundamentally from classical methods in an ambitious attempt to mitigate their most

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

#20 - ABSTRACT - (CONTINUED)

onerous disadvantages. These ideas are implemented and tested in a new modeling language and software system called LEXICON. Using LEXICON, a model is conceived, formulated, specified, expressed, internally documented, verified and directly executed in a single form. The LEXICON language is derived from a modeling form proposed by Geoffrion. The software engineering of the LEXICON system admits expansion of the language, portability, and linkage with contemporary real-time LP solvers.

include: Problem Generators
Matrix Generators.

Approved for public release; distribution unlimited.

LEXICON: A Structured Modeling System
for Optimization

by

Robert D. Clemence Jr.
Captain, United States Army
B.S., Lehigh University, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL

June 1984

Author: Robert D. Clemence Jr.

Approved by: _____
Thesis Advisor

GERALD G. BROWN
Second Reader

_____
Chairman, Department of Operations Research

_____
Dean of Information and Policy Sciences

ABSTRACT

Linear Programming (LP) is used infrequently for routine decision-making. Even in situations where LP is an extremely attractive tool, there is too much cost, frustration, delay and risk incurred in conversion of a mathematical hypothesis into a valid LP solution. This report outlines an entirely new approach to specifying and generating LP's which departs fundamentally from classical methods in an ambitious attempt to mitigate their most onerous disadvantages. These ideas are implemented and tested in a new modeling language and software system called LEXICON. Using LEXICON, a model is conceived, formulated, specified, expressed, internally documented, verified and directly executed in a single form. The LEXICON language is derived from a modeling form proposed by Geoffrion. The software engineering of the LEXICON system admits expansion of the language, portability, and linkage with contemporary real-time LP solvers.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I am grateful to Professor Arthur M. Geoffrion of the University of California, Los Angeles, for providing me the opportunity to contribute to his research. I would also like to express my appreciation to Professors Gerald G. Brown and Daniel R. Dolk. Their first-hand experiences with matrix generators and modeling languages were invaluable during the development of the LEXICON system. A special acknowledgement is due Professor Gordon H. Bradley. The ambitious goals I set for this work would not have been fully realized without his insight and tutelage in software engineering. Finally, I wish to thank my wife, Marilyn, for her patience, endurance, and support during this effort.

I. INTRODUCTION

The goal of this research is to make linear programming (LP) models much more reliable and responsive to the needs of decision makers. Although models can be critical in decision-making [Ref. 1], the actual usage of linear programming remains surprisingly low, e.g., [Ref. 2]. A principal reason for this paradox is the time lag between the formulation of a model and the production of correct, optimized results.

Practical linear programming problems are conceived in a form understandable by people, but solved in a form convenient for a computer algorithm. Fourer classifies these two representations as modeler's form and algorithm form [Ref. 3: pg. 143]. Modeler's form is written by a person and is usually expressed in symbolic notation as variables, constraints, and objectives. Algorithm form, on the other hand, is read by a machine and is typically a variable-by-variable listing of the non-zero coefficients of the problem. Whereas modeler's form is conceptual and defines an entire class of LPs, algorithm form describes a specific instance of the problem. Before an LP model can produce solutions for use by decision makers, it must be translated from its conceptual description to a computationally efficient form.

The method of translation selected greatly influences the timeliness of these solutions.

The predominant approach to translation is to divide the task between the modeler and the computer by writing a computer program called a matrix generator. Although this method is a vast improvement over manual translation, there are substantial difficulties inherent in its use. Matrix generators are written in either general purpose programming languages or special programming languages designed for creating algorithm forms. Since these languages do not have the expressive power of pure mathematical notation, the relationship between a modeler's form and its matrix generator form is always abstract and often obscure. Thus, a matrix generator requires both internal documentation, inherent in any computer program, as well as extensive external documentation of how it represents the modeler's form.

Perhaps the greatest difficulty inherent in the creation of this intermediate form is the need to verify the matrix generator program. A matrix generator is especially difficult to debug. Because its output is not intended to be read by humans, and can be voluminous for a large LP, manual inspection of the list of coefficients produced by the matrix generator is neither an efficient nor a reliable means of verification. Instead, matrix generators are verified indirectly through a series of manual and automated procedures.

Fourer describes this process in detail [Ref. 3: pg. 148] and concludes that

> ...even an erroneous MG [matrix generator] can look correct to a person, can generate output that passes many diagnostic tests, and can represent an LP that has a plausible solution. Thus there is normally a non-negligible risk, in the use of an MG, that the wrong LP will be generated, solved, and analyzed. [Ref. 3: pp. 148-149]

Hence, considerable human time and computer time can be spent to achieve a less than reliable matrix generator.

Along with documentation and identification, there is the unavoidable problem of modification. Whenever the modeler's form is changed to correct deficiencies in the model or to test a new hypothesis, the corresponding matrix generator must be revised. A change to a modeler's form can be instituted in a matter of hours; revising, verifying, and documenting this change in the matrix generator may require days or weeks. In a planning environment, where model structure is frequently altered and only a few production runs are made with each version, modification of the matrix generator can become an onerous and persistent task.

A simpler alternative to matrix generators is to create an executable modeler's form. This approach dispenses with intermediate forms altogether and makes the computer, not the modeler, responsible for the veracity of the modeler's-form-to-algorithm-form translation. The concept is straightforward: the modeler writes his modeler's form in a computer language designed for modeling; the computer reads this

symbolic description of the LP, along with the corresponding parameter data, and produces the prescribed algorithm form.

The executable model approach requires two components: a modeling language and a modeling language translator. A modeling language is a declarative language that expresses the modeler's form in a notation that the computer can interpret [Ref. 3: pg. 144]. As such, it must satisfy two conflicting sets of requirements. First, it must be convenient for people: it must be easy to learn, easy to use, and as powerful and flexible as the algebraic notation it is intended to replace. Second, it must be understandable to a machine: it must have an unambiguous syntax and a notation compatible with ordinary computer hardware. The modeling language translator is a compiler: it parses the language, interprets its expressions, and converts them from their higher-level modeler's form into the lower-level form required by the solution algorithm.

Executable linear programming models are in their infancy. Research to date has emphasized the development of modeling languages which resemble, as much as possible, common modeler's forms. These languages enable the modeler to formulate in his personal style and then convert his work into an executable model. However, the requirement to conceive in one form and express in another form persists [Ref. 3: pg. 158].

A yet untried alternative is to change the way the modeler views his problem so that formulation in a modeling language is both direct and natural. Geoffrion has proposed a theory

of model building, called structured modeling, that supports this approach [Ref. 4]. The merit of his idea is contingent on the existence of a modeling language translator capable of executing his hypothesized modeler's form.

The purpose of our research is to evaluate the feasibility of such a translator by designing and building an operational prototype modeling system based on Geoffrion's theory. This paper reports the details of that implementation, dubbed LEXICON, and the modeling language we have developed. Although structured modeling is applicable to models which are not LPs, our software has been designed specifically for practical linear programming problems.

Using the LEXICON language, a modeler can conceive his model in an immediately executable and internally documented form. This symbolic description can be read by a computer, processed into algorithm form, solved and its solution returned for analysis without manual intervention. Thus, LEXICON increases the value of linear programming to decision makers by allowing modifications to planning models to be made in minutes rather than days.

Our research is presented in five chapters. Chapter II presents an overview of the theory of structured modeling. Chapter III introduces the modeling language. Chapter IV describes the modeling language translator and principal system components. Chapter V discusses some of the practical aspects of the modeling language and the LEXICON system. Chapter VI presents our research conclusions.

II. OVERVIEW OF STRUCTURED MODELING

This chapter is a synopsis of a new theory of model building and model expression proposed by Geoffrion [Ref. 4]. The definitions, concepts and constructs presented provide the terms of reference needed to understand the modeling language and the illustrative example contained in Chapter III.

A. DEFINITIONS AND CONCEPTS

A structured model is composed of a finite collection of elements, partitioned into mutually exclusive and exhaustive element sets called genera, which in turn are organized into conceptual units called modules. Structured models are comprised of five types of elements: primitive entities (pe), compound entities (ce), attributes (a or va), functions (f), and tests (t). A primitive entity is an assertion of the existence of a physical thing or concept about which statements can be made. Each model must have at least one element of this type. A compound entity is also an existential assertion. It establishes a relation between other entities already defined that does not require a value. An attribute element has two parts: a tuple of entity elements and a unique value associated with that tuple. This value has a specified range and may be non-numeric. The fourth element type, the function, is identical to the mathematical

relation of the same name. It is defined by a domain of entity element tuples and a rule that associates a unique value in some range. The last element type, the test, is a function whose range is the logical values true-false.

Each element of the model is associated with a unique genus (singular of genera) which is composed of elements of only one type. We can thus unambiguously refer to the type of a genus.

All genera, except primitive entity genera, are defined in terms of other genera. This defining relationship among genera provides the framework of a structured model and is abstracted in the notion of a calling sequence. The calling sequence of a genus notes all the other genera that its definition depends on directly. This establishes a relationship "calls" (or "is partially defined by") among genera. For example, the calling sequence of an attribute genera "calls" the genera that it further describes.

Primitive entity genera (and optionally, other genera) introduce a named index. Each genus with more than one element has, through its calling sequence, an ordered set of the named indices. Each element of a genus is identified with a unique tuple of specific index values.

In a structured model the genera are ordered so that genera only call previously specified genera. This "define before use" or "no forward referencing" ordering is referred to in [Ref. 4: c. 2.3] as acyclic-preserving. In general, there are many orderings satisfying this condition.

Genera contiguous within an acyclic-preserving ordering can be grouped into a higher conceptual unit called a module. Each genus is related to a unique module, this relationship is called "part of." Contiguous modules and genera may also be grouped into a module by this relationship. Except for one distinguished module, called the model module, each module and genus must be "part of" one other module. This relationship may not contain any cycles.

The modules and genera together with the relationships which both order and connect them may be viewed as forming an arborescence with the model module as root, the genera as leaves and the other modules as interior nodes. Since in general there are many alternate groupings of genera and modules and there are many alternate acyclic-preserving orderings, the modeler has a wide latitude in developing his model.

The function performed by a structured model module is to organize physical things, specifications and relationships among parts of the system being modeled into a conceptual structure which facilitates understanding for both the modeler and those who will use the model. An important feature of a module is that modules are themselves made up of still smaller modules. Thus, the modules of a structured model form a hierarchy. At the top level is the entire model and at the bottom level are the genera and their element sets.

B. PRIMARY CONSTRUCTS

A single model instance may be obtained by enumerating all elements and their calling sequences, all attribute values, all function and test rules, and the acyclic-preserving and module relationships. In the sense of this paper, a model with these aforementioned characteristics is termed completely-specified.

A class of similar models is obtained by separating the mathematical/conceptual formulation from the elemental detail necessary for computation. The symbolic description of this class is the modeler's form of structured modeling, called a master dictionary. The parameter data, called elemental detail, necessary to manipulate the model is contained in a dictionary extension called an element section.

A master dictionary has one module paragraph for each module and one genus paragraph for each genus. It displays all modules and genera in a linear fashion in a way which is acyclic-preserving. Each paragraph contains

1. the name of the module or genus;
2. an interpretation of the module or genus in natural language; and
3. if a genus, the element type and other elemental information about the genus which may be given without enumeration of the individual elements.

The element section supplements the information provided in the genus paragraphs. The amount of detail provided for

each paragraph depends on the genus type and the degree of specification desired. The final level of specification may, for example, be complete except for the values of selected attribute elements. This state is referred to as A-partial specification. The unspecified elements, the 'variables' of conventional models, are referred to as variable attributes (va). Notation and syntax for the modeling system developed in this thesis are presented in the next chapter.

C. ANALYTIC USES OF A STRUCTURED MODEL

Models are used in a wide variety of contexts for differing purposes. Many of the ways models are used can be summarized by four analytic modes: evaluation, retrieval, satisfaction and optimization. The definitions which appear below presume an A-partially or completely specified model [Ref. 4: c. 3.1].

1. Evaluation: the process of determining the values of all function and test elements. If the model is A-partially specified, trial values must be given for the variable attributes. Evaluation is often called 'simulation.'
2. Retrieval: the process of answering queries about the element detail and conceptual structure of the model that do not require evaluation.
3. Satisfaction: the process of specifying values for all variable attributes such that evaluation results

in all test elements being true-valued. The variable attribute element values specified, called a feasible solution, may constitute an end in itself or a prelude to optimization.

4. Optimization: the process by which a feasible solution is found that maximizes (or minimizes) the value of a selected real-valued function element.

III. LEXICON STRUCTURED MODELING NOTATION

A. GENERAL

LEXICON is a prototype structured modeling system for linear optimization problems. It accepts as input user-specified master dictionary and element section files; translates this external representation of the LP into an algorithm form compatible with the Brown and Graves XS mathematical programming optimizer [Ref. 5]; and submits the problem for solution. After an optimized result has been obtained, the modeling system reports the values of the objective function and the decision variables in a labelled format. In addition to its primary role, LEXICON offers two other procedures useful during model development. A verification sub-program is available to locate and identify structural and grammatical inconsistencies in a master dictionary. This file can also be manipulated by a video display sub-program to create various file perspectives.

This chapter presents the prototype modeling language designed for the LEXICON modeling system. Its syntax is similar to one hypothesized for computational implementation by Geoffrion, but does not include all the executable features he envisions. Our design, however, does permit these exceptions to be implemented at a later date if warranted by computational experience. In order to achieve an executable

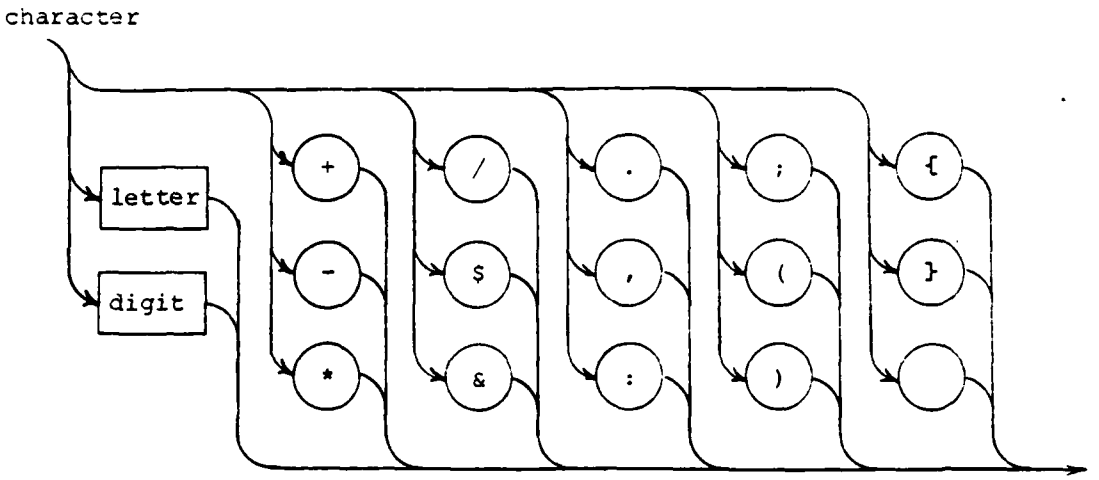
model, three types of simplifying restrictions have been imposed on the modeling language. We will distinguish these as design, implementation, and prototype restrictions, although to some extent they all overlap. Design restrictions define the functional limitations of the software. Implementation restrictions represent capability that was designed for but not implemented. Prototype restrictions represent specific restrictions in the prototype (like dimensions on arrays, lengths of names, etc.) that can be easily changed.

Section A deals with the primitive tokens of the modeling language. It also introduces the 'railroad diagram' used to document these building blocks and the more complex expressions discussed in later sections. Sections B and C present the syntax used in the master dictionary. Section D covers the makeup of the element section. Finally, Section E presents a structured model master dictionary and element section for a textbook modeling problem.

B. LANGUAGE BUILDING BLOCKS

1. Character Set

The character set consists of digits, upper case letters, lower case letters and a subset of the special characters available in the EBCDIC character code, e.g., [Ref. 6]. The full character set is defined by the first flow chart in Figure 3.1. This 'railroad diagram' is traversed by entering at the top left-hand corner and exiting



Note: The circle containing no symbol is a blank.

Figure 3.1 Character Set

from the bottom right. The name above the entry shows the language token which the diagram defines. The boxes and circles within the chart are like turnstiles: they may only be passed through in the direction indicated and then only upon 'payment' of the required items. The payment required for circles and rounded boxes is the characters displayed inside them. Rectangular boxes require an entity defined elsewhere. The paths between the turnstiles should be treated like railroad lines. Shunting backwards past a sharp corner in the path is not permitted [Ref. 7: pg. 19].

2. Constants

The language recognizes both integer and real number constants. Real numbers may be expressed in either

decimal or scientific notation. Figure 3.2 defines their form. Sign conventions are defined by another construct.

3. Symbolic Names

Six types of symbolic names are used to build more complex expressions. Figure 3.3 shows the conventions used for genus and module names. Each paragraph name in the master dictionary must be unique. Formats for domain indices, index function names and an intermediate construct, the basic generic name, are defined in Figure 3.4. The identifier, the sixth name type, is shown in Figure 3.5. Table 3.1 lists the names which are language keywords. Details on keyword usage are provided later. Within the above limitations, the intention is to allow meaningful names (e.g., mnemonics, abbreviations, etc.) to be used when describing and documenting the model. We suggest, for example, that genus names beginning with the characters 'T:' be used for test genera and that the '$' character be included in cost or profit-related genus names.

C. MASTER DICTIONARY CONSTRUCTS

Each paragraph of a master dictionary is composed of an ordered collection of statements. Module paragraphs have a constant form. There are several forms of the genus paragraph, differing according to the type of elements comprising the genus. This section defines the seven statement types used when forming paragraphs.

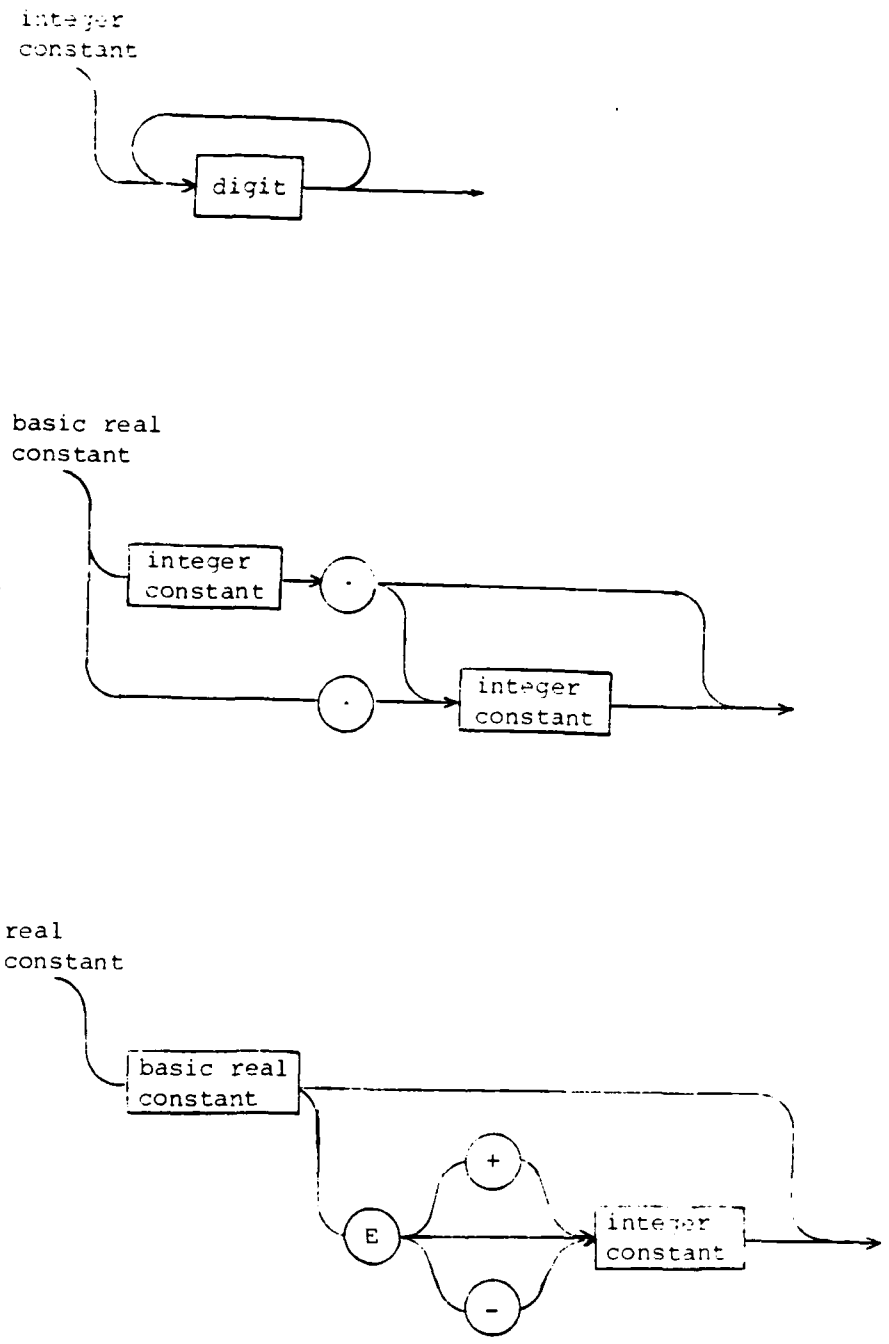
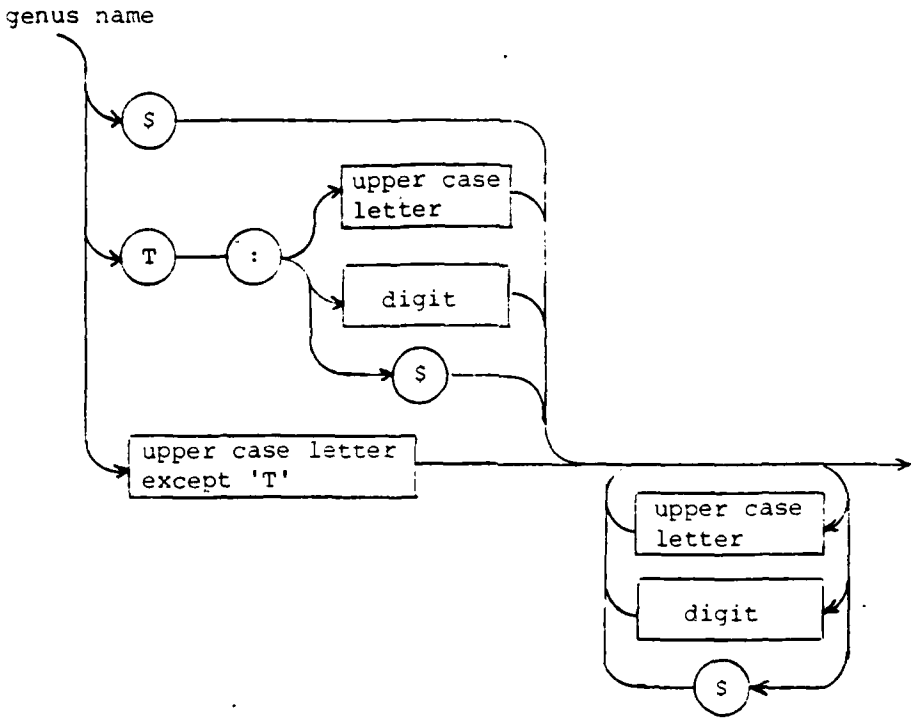
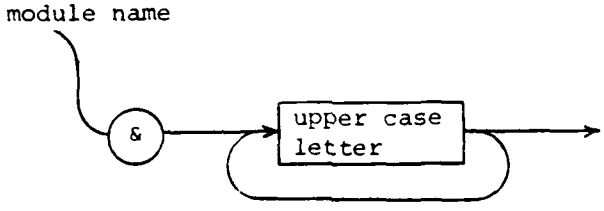


Figure 3.2 Integer and Real Constants. Design Restrictions: IBM 3033 Single-word Integer, Double-word Real

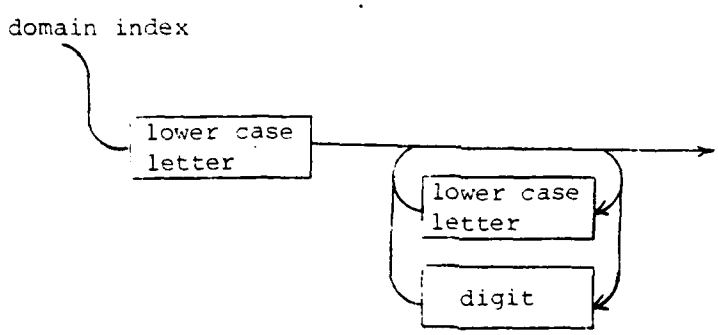


Prototype Restriction: No More Than 10 Characters

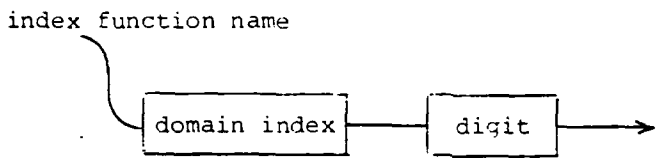


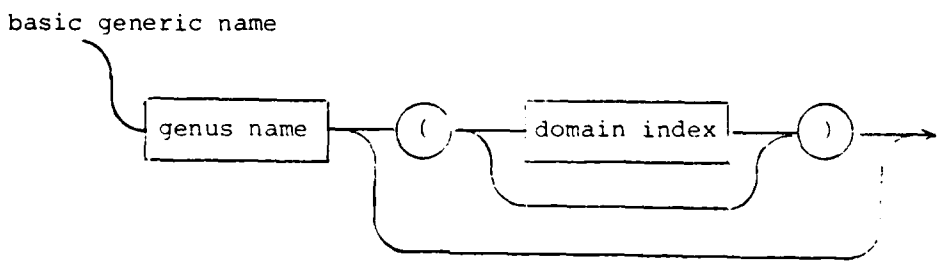
Prototype Restriction: No More Than 11 Characters

Figure 3.3 Genus and Module Names



Prototype Restriction: No More Than 5 Characters

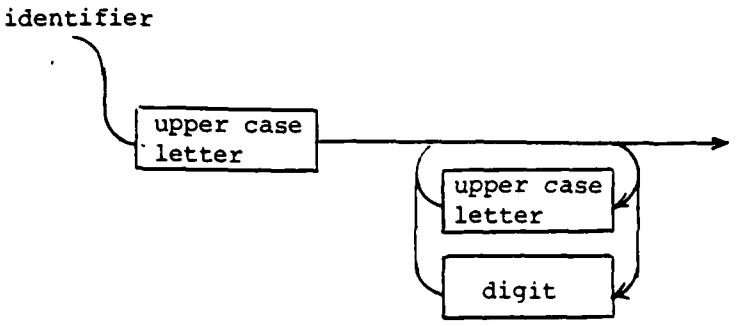




Prototype Restriction: No More Than 10 Domain Indices

Note: generic names without a domain index are unindexed.

Figure 3.4 Domain Index, Index Function Name and Basic Generic Name Constructs



Prototype Restriction: No More Than 6 Characters

Figure 3.5 Identifier

Table 3.1 LANGUAGE KEYWORDS

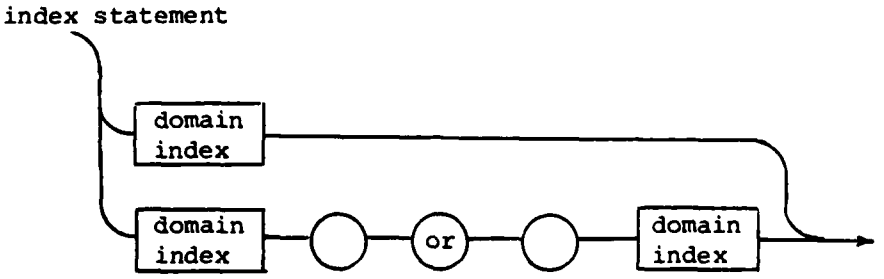
	Usage	Words
1.	System Arithmetic Functions	ABS, MAX, MIN, MLX, SUM
2.	Relational Operators	LT, LE, EQ, GE, GT, NE
3.	Set Operator	SELECT
4.	Data Descriptors	&ATR, &FNX, &SET
5.	Translator Reserved Word	CHECK (not available to user)

1. Interpretation

Interpretation is non-executable, natural language text intended as a comment to increase the documentation content of the paragraph. It is the only paragraph statement used by both modules and genera. Module interpretations should convey the sense in which each module unifies its constituent modules and/or genera. Genus interpretations describe the typical element in the genus.

2. Index Statement

The index statement defined in Figure 3.6, introduces a distinct domain index for use with the elements of its genus paragraph. The values of the domain index are specified by an ordered list of identifiers contained in the element section. Each primitive entity genus introduces a domain index. These and other genera that introduce domain indices for modeling convenience, are said to be directly



Implementation Restriction: Only One Alias Allowed

Figure 3.6 Index Statement

indexed. All other genera that may contain more than one element are said to be indirectly indexed by one or more domain indices derived from the components of their calling sequence statements. This procedure is explained later. Genera that must be singletons (contain only one element) are said to be unindexed.

Domain indices, whether introduced directly or indirectly derived from a calling sequence statement, provide the index tuple component of the basic generic name used to denote a typical element of a genus. When each domain index is given a specific identifier value, the basic generic name provides a unique element name. The singleton elements of unindexed genera are uniquely identified by their genus name alone.

The index statement is executable and allows a domain index alias to be introduced if required by the model. For example, if a network were described in a structured model, each node could be an element of a NODE genus indexed by the domain index 'i' or its specified alias 'j'. If the arcs connecting the network nodes are elements of a genus called LINK, then LINK(i,j) clearly identifies a typical arc, without introducing a separate genus and domain index to distinguish tail nodes from head nodes.

3. Calling Sequence Statement

The calling sequence statement gives the generic (typical) calling sequence for an element in its genus

paragraph. The generic calling sequence for the typical arc, LINK(i,j), would refer to a typical tail node, NODE(i), and a typical head node, NODE(j), in order to identify its position in a network. Thus, the calling sequence statement of the LINK genus would contain two NODE components and would be written as '(NODE(i),NODE(j))'.

Figure 3.7 shows that the calling sequence statement is composed of one or more genus components. The sequence of a called component, given in Figure 3.8, is founded on the basic generic name of the genus called, supplemented by special subscript expressions.

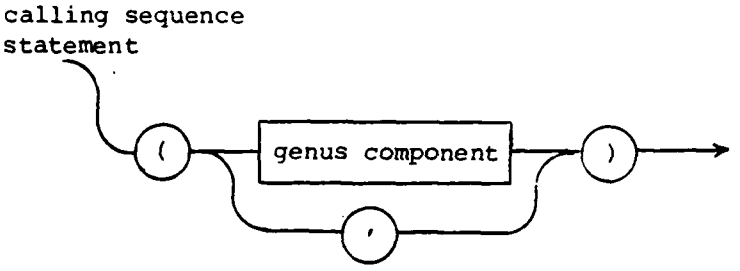
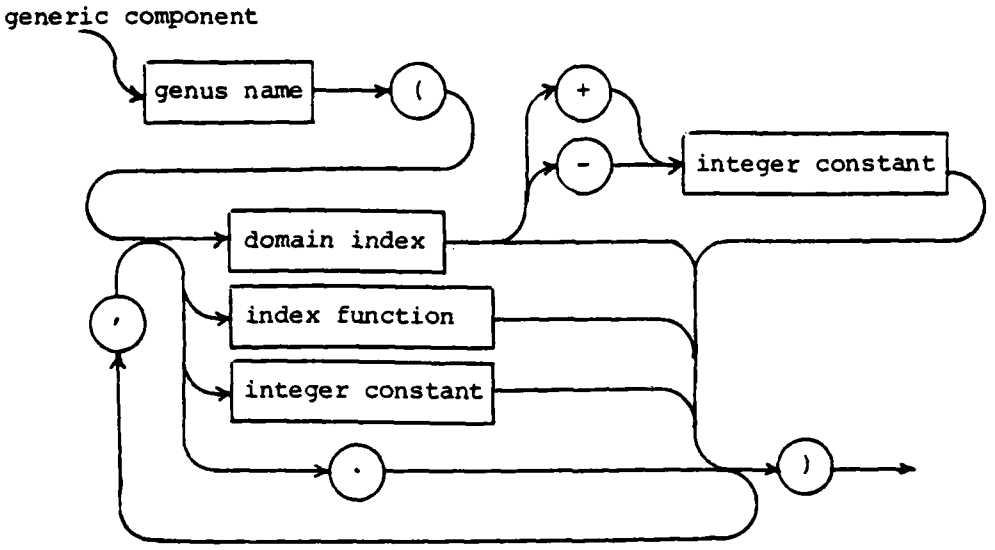


Figure 3.7 Calling Sequence Statement

These domain index substitutes allow each component the expressive power necessary to represent specific as well as multiple elements of its genus. In general, these expressions either:

a) modify the index;

b) assign it a fixed value;



	Option chosen instead of domain index 'i'	Meaning
1)	i ± N	Select the Nth value after (if "+"), or before (if "-") the value of i. If N is omitted, select all values from i to the end (if "+") or from i to the beginning (if "-") of its index set.
2)	N (an integer)	Select the Nth value of i.
3)	a dot	Select all values of i.
4)	index function	Select a single value of i according to a rule that depends on the values of independent indices which form the index function argument.

Restriction: No more than 10 domain indices or domain index replacements

Figure 3.8 Generic Component Syntax

c) annihilate it by referencing all its identifiers; or

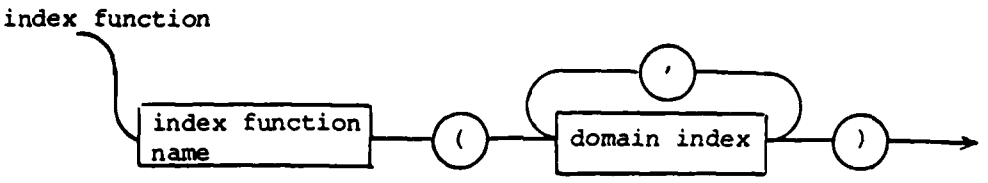
d) render its value dependent on one or more other (independent) indices.

A complete discussion of the index functional dependence introduced by option (d) must be preceded by an explanation of exactly how domain indices are determined for genera that do not introduce one through an index statement. The domain indices of indirectly-indexed genera are specified by a minimum covering of the unreplaced domain indices, those modified by option (a) and the independent indices of those replaced by option (d) for all components of the calling sequence statement. Consider the calling sequence statement for the calling genus, 'B':

(A(p,j,3),C(i,k(j)),D(·,ℓ+2))

The domain indices of 'B' are: 'p', 'j', 'i' and 'ℓ.' Since the order assumed is left-to-right in order of appearance in the calling sequence statement, the generic name of genus 'B' would be written 'B(p,j,i,ℓ).'

The term 'k(j)' in component 'C(i,k(j))' is one instance of the syntax defined in Figure 3.9 for index functions. Here, 'j' is the independent index. 'k' is the index whose value is functionally dependent on the value assigned to 'j.' In general, a single identifier of the index range of 'k' is selected according to a rule that depends on



Prototype Restriction: No more than 2 domain indices

Figure 3.9 Index Function Syntax

the values of a subset of the calling genus' domain indices. The exact mapping is specified in the element section of the master dictionary. Since an index function may be invoked more than once in a single generic calling sequence and also for more than one calling sequence statement, the notation specifies that the name of the index being rendered dependent be followed by a digit if more than one dependency is needed in the model. The LEXICON prototype allows up to nine distinct functional dependencies to be defined on any domain index.

The calling sequence statement is executable documentation for the physical relationships and dependencies the model describes. The LEXICON system is designed to verify the pre-definition of each genus and domain index in the statement, to create the internal representation necessary to support index functions introduced and to determine

a default index set from the calling sequence statement if an index set statement is omitted from the genus paragraph.

4. Index Set Statement

The values of the index tuples of a generic name constitute a genus index set. The syntax for an index set name is given in Figure 3.10. Each genus element has a unique index tuple in its genus index set. The members of the set are ordered according to the order of appearance of the identifiers if there is just one domain index. If there are multiple indices, the set members are ordered lexicographically according to the order of domain index appearance in the generic name; this is called the index-induced order [Ref. 4: c. 2.5].

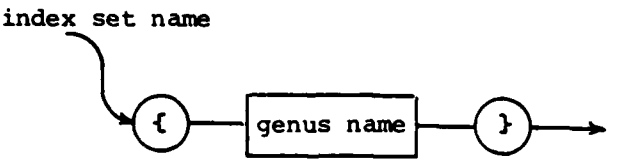


Figure 3.10 Index Set Name

The index set statement is an executable field which limits the size of the index set allowed by the generic name of its genus paragraph. There are several statement forms, differing according to whether the genus is directly-indexed, indirectly-indexed or unindexed. If

the genus is directly indexed, an integer entry documents the cardinality of the direct index set it introduces. Omission implies that the size of the set will be determined by enumerating its identifiers in the element section. The requirement that unindexed genera specify the integer '1' as their index set statements is an implementation restriction.

The index set statement of an indirectly-indexed genus either refers to a previously specified genus index set that possesses the same domain indices in the same order, or provides a rule by which its indirect index set may be constructed from other genus index sets. Omission of an index set statement asserts that the genus is indexed by its default index set. A default index set consists of all domain index tuples for which a generic calling sequence is 'well-defined.' A calling sequence is well-defined if each component in the sequence is completed as a valid element name.

The LEXICON design allows an indirect index set to be specified in one of two ways. The keyword 'SELECT,' followed by an index set name argument, asserts that the set is a subset of the identifier-tuples formed by the Cartesian product of the domain indices of the named sets. The Cartesian product of two sets I and J is the set (denoted by $I \times J$) of all pairs (i,j) such that i is a member of I and j is a member of J [Ref. 8: pg. 287]. In the sense of this paper, I and J are the direct index sets that range the

domain indices i and j. K-tuples (k > 2) for an index set with 'k' domain indices can be formed by recursively applying this definition. The 'SELECT' option requires that the user specify the identifier-tuple of each element of the subset in the element section.

The second way to define an indirect index set is as a Cartesian product of Cartesian product derived indirect and/or direct sets. If this latter option is preceded by 'SELECT,' the intention is to define a subset of this aggregation. The syntax for the index set statement is shown in Figure 3.11.

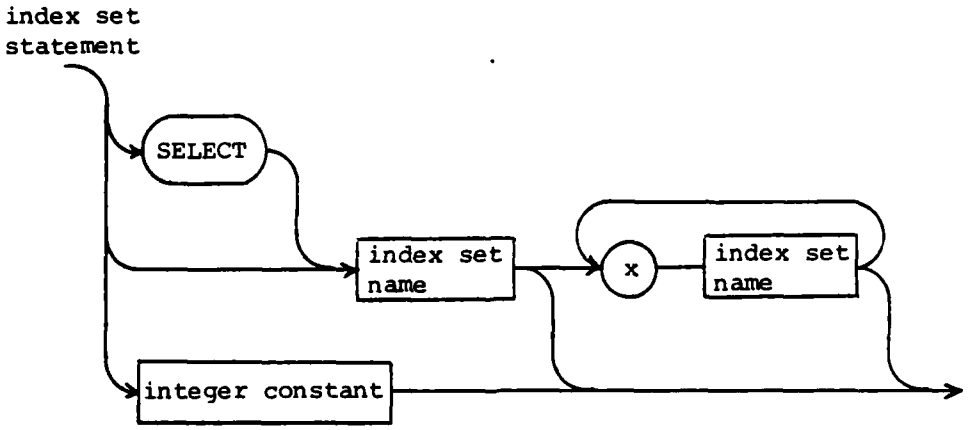


Figure 3.11 Index Set Statement

5. Genus Type Statement

This executable field specifies the type of elements introduced by a genus paragraph. The two-character codes used are shown in Figure 3.12.

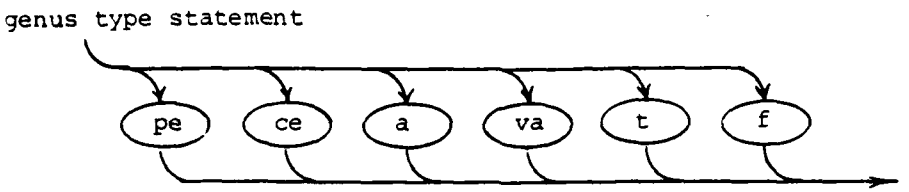


Figure 3.12 Genus Type Statement

6. Range Statement

The range statement implemented by the LEXICON system is a non-executable, optional field. It is intended as a comment to increase the information content of attribute paragraphs. Since the LEXICON design limits attribute element values to the real number system, we suggest using this field to record the sign, magnitude, or other criteria useful in validating the model's data.

7. Rule Statement

Rule statements are introduced by function and test genus paragraphs. A function genus rule statement is a real expression that evaluates to a real number for each of its elements. For a test genus, the rule statement is formed as a relational expression which values its elements as 'true' or 'false' (see Figure 3.15).

The choice of an executable, yet natural, syntax for the rule statement was a major design issue. The language developed is an amalgam of conventional mathematical notation

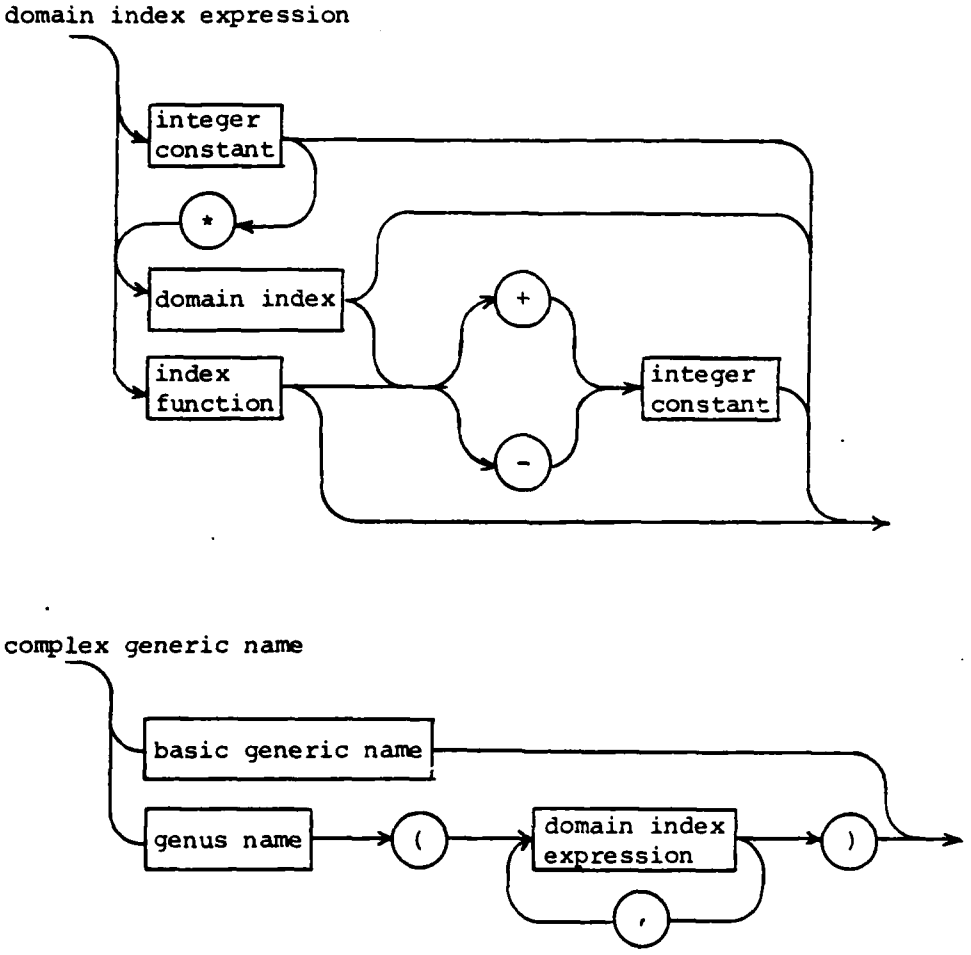
and the syntax used by FORTRAN arithmetic expressions. This combination provides technically trained people with a familiar method of specifying arithmetic expressions supplemented by powerful LEXICON system functions. These functions enable useful operations such as 'sum', or 'chose the smallest valued element' to be iterated over indexed expressions or applied to collections of elements. Rule statement syntax and system function capabilities are detailed below.

a. Real Expressions

The real primary, the basic unit of a real expression, is composed of real constants, system function calls and complex generic names. Generic names, in this context, can be thought of as the array elements used in FORTRAN. The generic name format, and the format of an intermediate construct, the domain index expression, are illustrated in Figure 3.13. Discussion of system function calls and their formats is deferred until later. Figure 3.14 defines the syntax of the real primary and the real expression. These last diagrams are adapted from charts produced by Day for FORTRAN [Ref. 7: pg. 60].

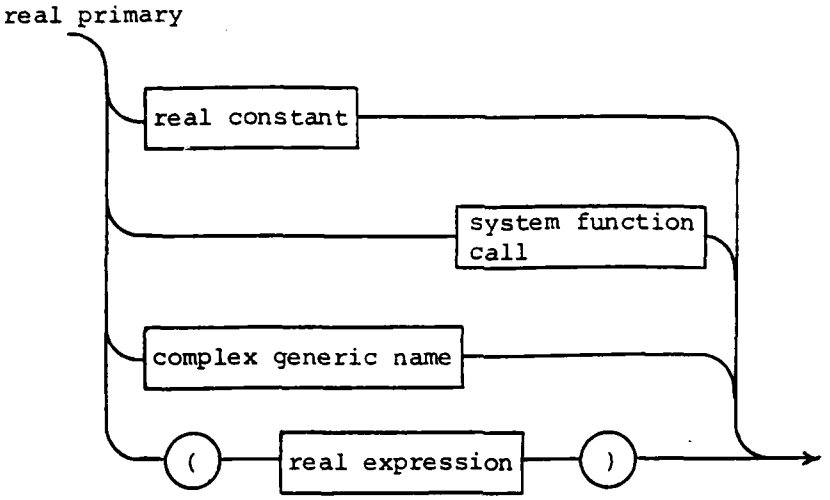
b. Relational Expressions

Test genus rule statements are relational expressions made up of two real expressions, separated by a relational operator. The six relational operators and the syntax for a relational expression are shown in Figure 3.15.



Prototype Restriction: No more than 10 domain index expressions

Figure 3.13 Complex Generic Name Syntax



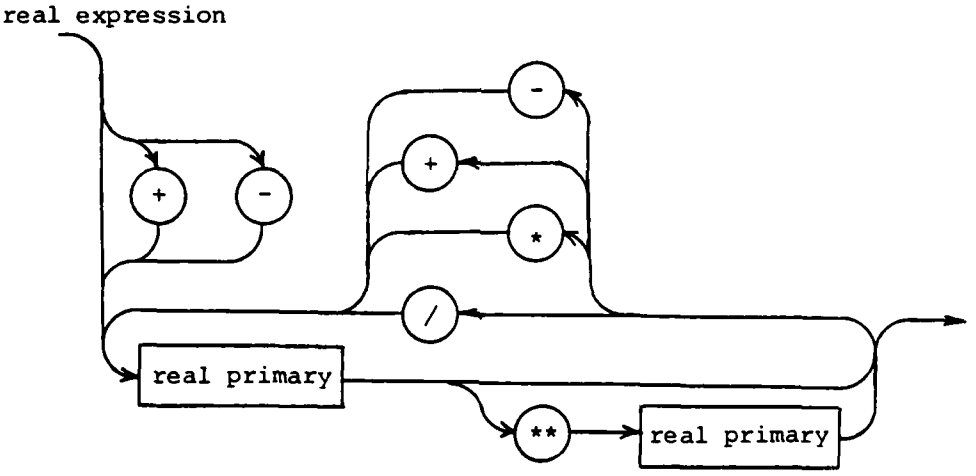


Figure 3.14 Real Expression Syntax

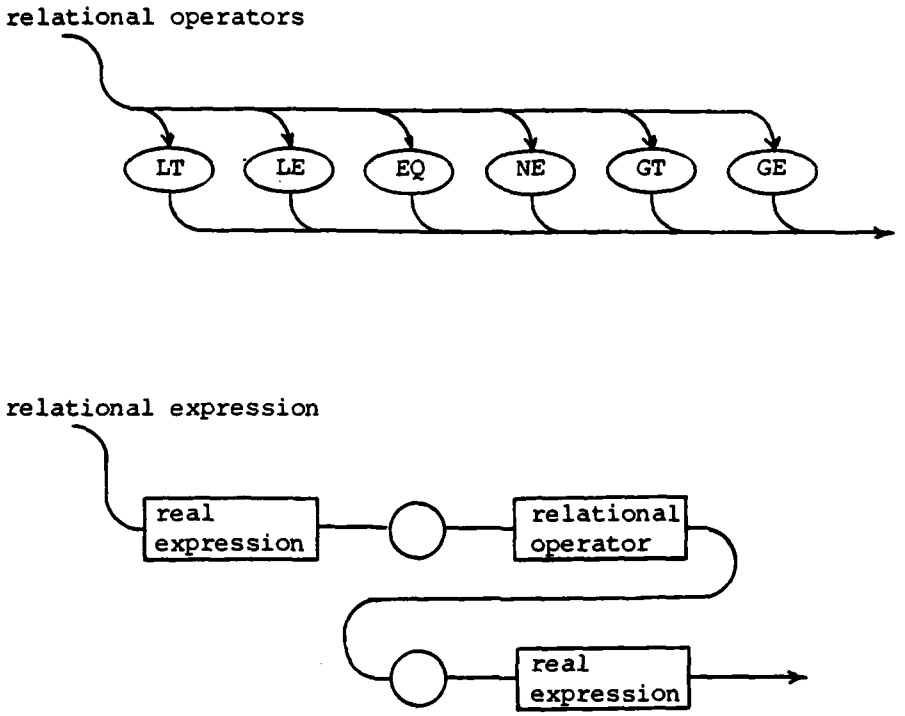


Figure 3.15 Relational Expression Syntax

c. System Function Calls

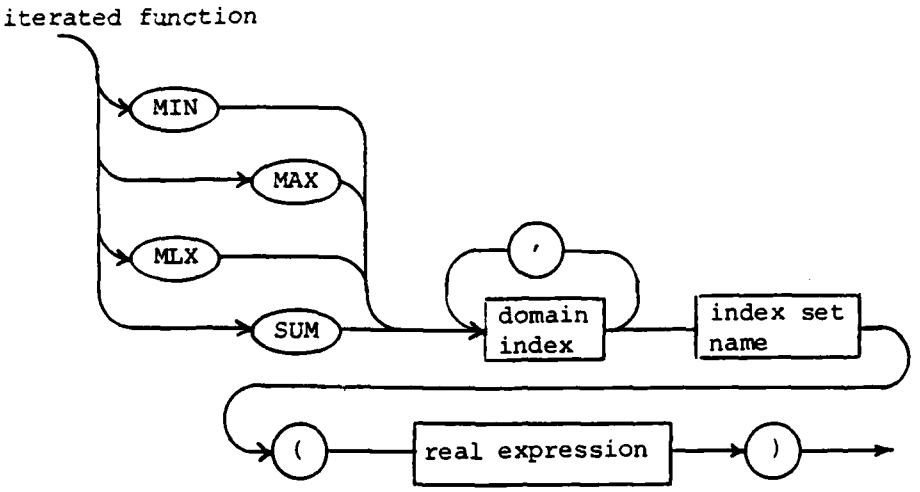
The prototype provides seven intrinsic functions for computation. Three static functions, "MIN', 'MAX' and 'ABS', operate over fixed collections of arithmetic expressions. They provide the capability to choose the smallest value or the largest value from a collection and to compute the absolute value of a single arithmetic expression. The remaining four system functions perform their defined

operations by iterating an arithmetic expression using specified domain indices from a chosen genus index set. Iterated functions provide the capability to find both largest and smallest values ('MIN', 'MAX') and to compute the sum or product ('SUM', 'MLX') of real expressions over single or multiple domain indices. The syntax of each system function type is given in Figure 3.16.

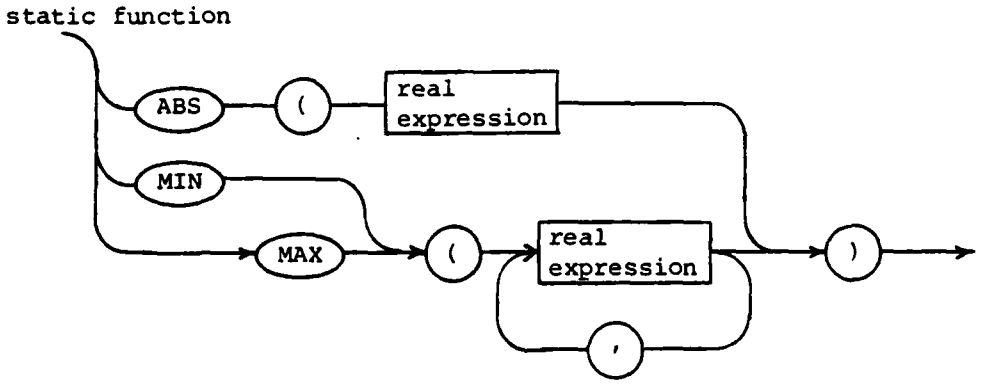
D. MASTER DICTIONARY FORMAT

The master dictionary is an ordered list of module and genus paragraphs. For each module, it is required that its parts be contiguous in the acyclic-preserving ordering. The modular structure is specified by indenting the list of module and genera paragraphs: for each module all the module and/or genera paragraphs that are "part of" it are indented (to the right) an equal amount.

The indentations and presentation sequence used are acyclic and are restricted such that each module/genus definition is completed before any other module/genus definition which calls or depends upon that definition (i.e., module/genus definitions are both acyclic and individually contiguous). This indentation and sequence restriction may be viewed as a pre-order presentation of the modular structure. This restriction can always be satisfied, usually in many ways, and endows the language with desirable properties not completely discussed in the present paper. Figures 3.17



Prototype Restriction: No more than ten domain indices



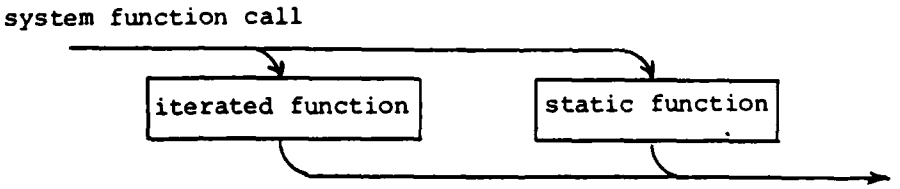


Figure 3.16 System Function Call Syntax

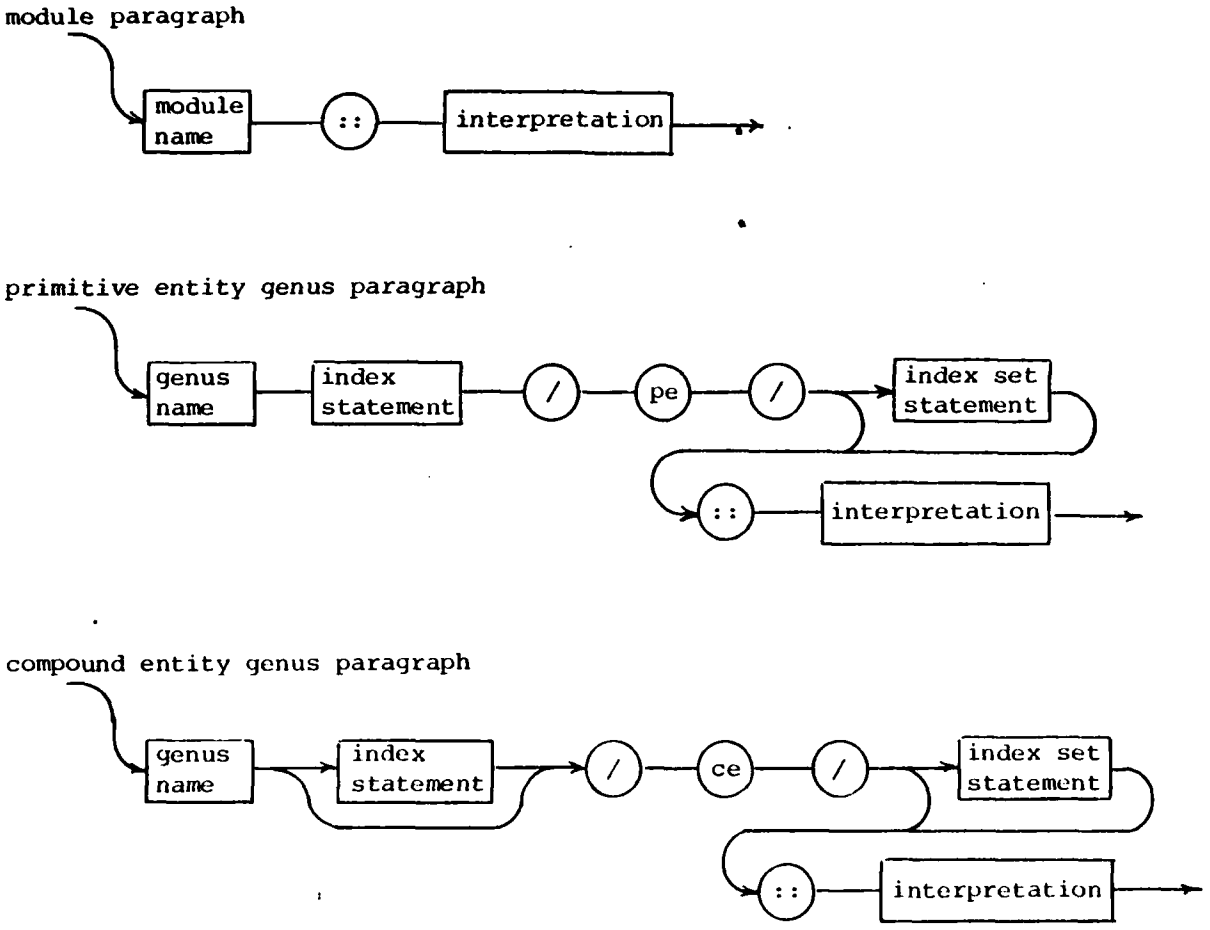


Figure 3.17 Syntax for Module, Primitive Entity Genus, and Compound Entity Genus Paragraphs

and 3.18 illustrate the format of a module paragraph and the four formats used by genus paragraphs.

LEXICON is designed to accept a master dictionary, prepared as a 72-column text file, as input. Although it requires no particular indentation scheme, the software expects that level-1 (the entire model is level-0) modules and genera begin in column 1, and that the indentation used is consistent within levels of the model. If a paragraph cannot be completed in one line, a continuation line may be initialized by a continuation symbol, '..', indented at the same depth as the first line of the paragraph. This prototype allows no more than nine continuation lines per paragraph. One blank line may be left between adjacent paragraphs, if desired.

E. ELEMENT SECTION FORMAT

The element section fixes a concrete instance of the model contained in the master dictionary by providing values for genus index sets, index functions and the elements of attribute genera. Index set members are specified as tuples of identifiers. Index function mappings and attribute elements are specified by an identifier-tuple and a value. The value of an index function mapping is an identifier from the direct index set of the index rendered dependent. The value of an attribute element is a real number. Each genus paragraph in the model which creates a new index set,

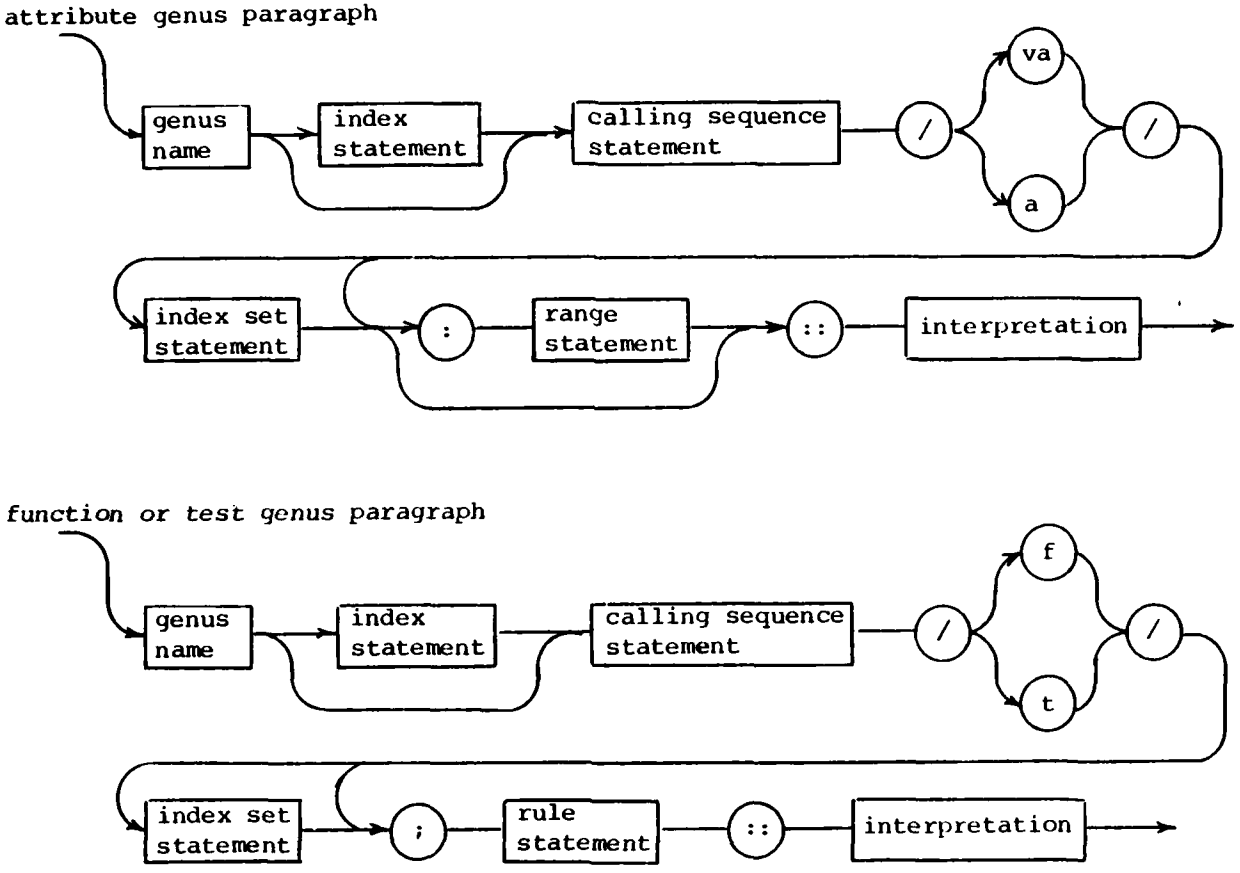


Figure 3.18 Syntax for Attribute, Function, and Test Genus Paragraphs

creates a new index function, or is an attribute genus, induces a separate data requirement.

LEXICON is designed to accept this data from a 72-column disk file, partitioned by descriptor statements into descriptor data blocks of different entry formats. Each block contributes values to the members of an index set, to the mappings of an index function or to the elements of an attribute genus. A block must appear in the same order as its parent paragraph in the master dictionary.

The descriptor statement provides an execution-time format for the data block it prefaces. It consists of a descriptor and the name of the object being specified. The descriptors implemented to preface index set, index function and attribute genus names are '&SET', '&FNX', and '&ATR', respectively. Descriptor statement syntax is defined in Figure 3.19.

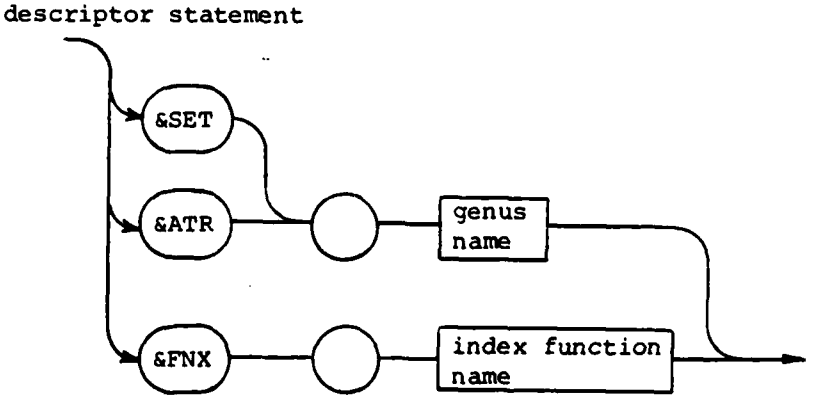
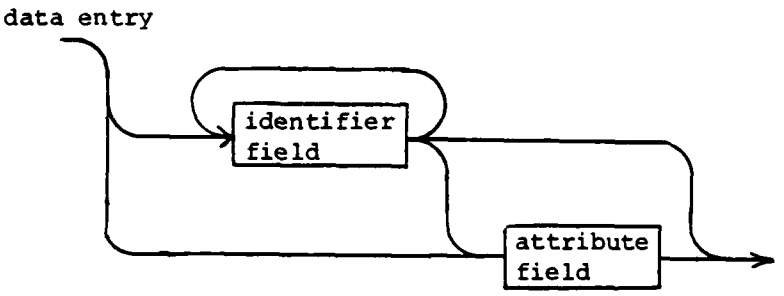


Figure 3.19 Descriptor Statement

LEXICON checks each data block to ensure that the data provided is required by the model and is provided in a format consistent with the master dictionary. In general, a six-character identifier field is reserved for each identifier listed in an identifier-tuple or as an index function value. The LEXICON prototype expects identifiers to be right-justified within their fields. However this presentation can be easily changed to left-justification in a production version. A real number value is entered by an attribute field. Attribute fields may be no more than twelve characters long. A typical data entry always begins in column 1 and is an identifier field (see Figure 3.20).



Prototype Restriction: No more than ten identifier fields

Figure 3.20 Data Entry Format

The number of descriptor data blocks needed to completely or A-partially specify a model, can be reduced by using an intrinsic system option. Rather than specify all the

identifier-tuples of an attribute genus' index set twice (once in the &SET data block of the referenced index set and a second time in the attribute genus &ATR data block), LEXICON automatically values and orders the tuples of an undefined index set from the &ATR data block alone.

Data entry requirements are also diminished for unindexed attribute genera and attribute genera that refer to certain index sets. Because an unindexed genus has no identifier-tuple, an unindexed attribute genus is valued by providing only one real number for its element. Other attribute genera may be specified by listing only the real number values of their elements in two instances. The first instance occurs when the genus index set is inherited 1:1 from a previously defined direct index set. The second occasion when the requirement for an identifier-tuple can be relaxed is when the genus index set is a Cartesian-indirect index set. In the sense of this paper, a Cartesian-indirect index set is a set formed by the Cartesian product of two or more direct index sets or by the use of this set operation between index sets that were originally defined by the Cartesian product of direct index sets. When attribute values are not prefaced by identifier-tuples, their attribute fields begin in column 1. The sequence of presentation within a block is assumed to correspond to the index-induced order of the genus index set.

F. ILLUSTRATIVE EXAMPLE--THE TANGLEWOOD CHAIR MANUFACTURING COMPANY

This section introduces an A-partially specified structured model as a prelude to the modeling scenario presented in Chapter V. The scenario is an enrichment of a formulation exercise, posed by Jensen and Barnes [Ref. 9: pg. 12] that appears below.

THE TANGLEWOOD CHAIR MANUFACTURING COMPANY

> The Tanglewood Manufacturing Co. has four plants located around the country. The fabrication and assembly cost per chair and the minimum and maximum monthly production for each plant are shown in Table 3.2. The company obtains the twenty pounds of wood required to make each chair from two suppliers who have agreed to supply any amount ordered. In return, the company guarantees the purchase of at least 8 tons of wood per month from each supplier. The cost of wood is $0.10/lb. from Supplier 1 and $0.075/lb. from Supplier 2. The shipping cost in $/lb. from each supplier to each plant is shown in Table 3.3. The chairs are sold in New York, Houston, San Francisco, and Chicago. Transportation costs in $/chair between the cities and plants are listed in Table 3.4. Table 3.5 shows the minimum demand that must be satisfied, the maximum demand and the selling price for chairs in each city.

The model required is to be used for optimization. Subject to a criterion of minimizing total cost, a policy is needed which specifies:

1. where and in what quantities the raw materials for each plant should be purchased;
2. the number of chairs to be produced by each plant; and
3. a distribution plan for each plant's output.

Table 3.2

FABRICATION COST AND PRODUCTION RESTRICTIONS BY PLANT

Plant	Cost Per Chair	Max. Production	Min. Production
1	$5.00	500	0
2	7.00	750	400
3	3.00	1000	500
4	4.00	250	250

Table 3.3

SHIPPING COST FROM SOURCE TO PLANT

$/lb. of Wood	Plant 1	Plant 2	Plant 3	Plant 4
Wood 1	0.01	0.02	0.04	0.04
Source 2	0.04	0.03	0.02	0.02

Table 3.4

TRANSPORTATION COST BETWEEN PLANTS AND CITIES

$/Chair	New York	Houston	San Francisco	Chicago
1	1.00	1.00	2.00	0.00
2	3.00	6.00	7.00	3.00
Plant 3	3.00	1.00	5.00	3.00
4	8.00	2.00	1.00	4.00

Table 3.5

SELLING PRICE AND DEMAND RESTRICTIONS BY CITY

City	Selling Price Per Chair	Max. Demand	Min. Demand
New York	$20.00	2000	500
Houston	15.00	400	100
San Franciso	20.00	1500	500
Chicago	18.00	1500	500

A master dictionary and element section for the Tanglewood problem are shown in Appendices A and B, respectively. The model is written using the notational conventions of this chapter and is executable by the LEXICON modeling software. It is based on a formulation prepared by Geoffrion for the same exercise [Ref. 4: c. 2].

The modular structure of the model is shown in Figure 3.21. The leaf nodes of the arborescence are the model genera. Each sub-tree forms a distinct conceptual unit. Part of the model describes the wood sources (&SDATA); part describes plants (&PDATA); part describes transportation (&TDATA); part describes the decisions to be made (&DECISIONS); part describes the volume and cost consequences of decisions (&CONSEQ); and part describes the systems material production and sales restrictions (&TESTS).

A restricted preorder traversal of the arborescence yields the indentation and presentation sequence of the master dictionary. Preorder is restricted such that the successors of a module/node are visited in an acyclic sequence with respect to calling dependencies. Indentation level of a module/node is analogous to the path length from the module/node to the model module or root node.

The generic structure of the model is based on three primitive entity genera (SOURCE, PLANT, CUST) and two compound entity genera (IBLINK, OBLINK). These five genera represent all the physical elements of the problem. The remaining

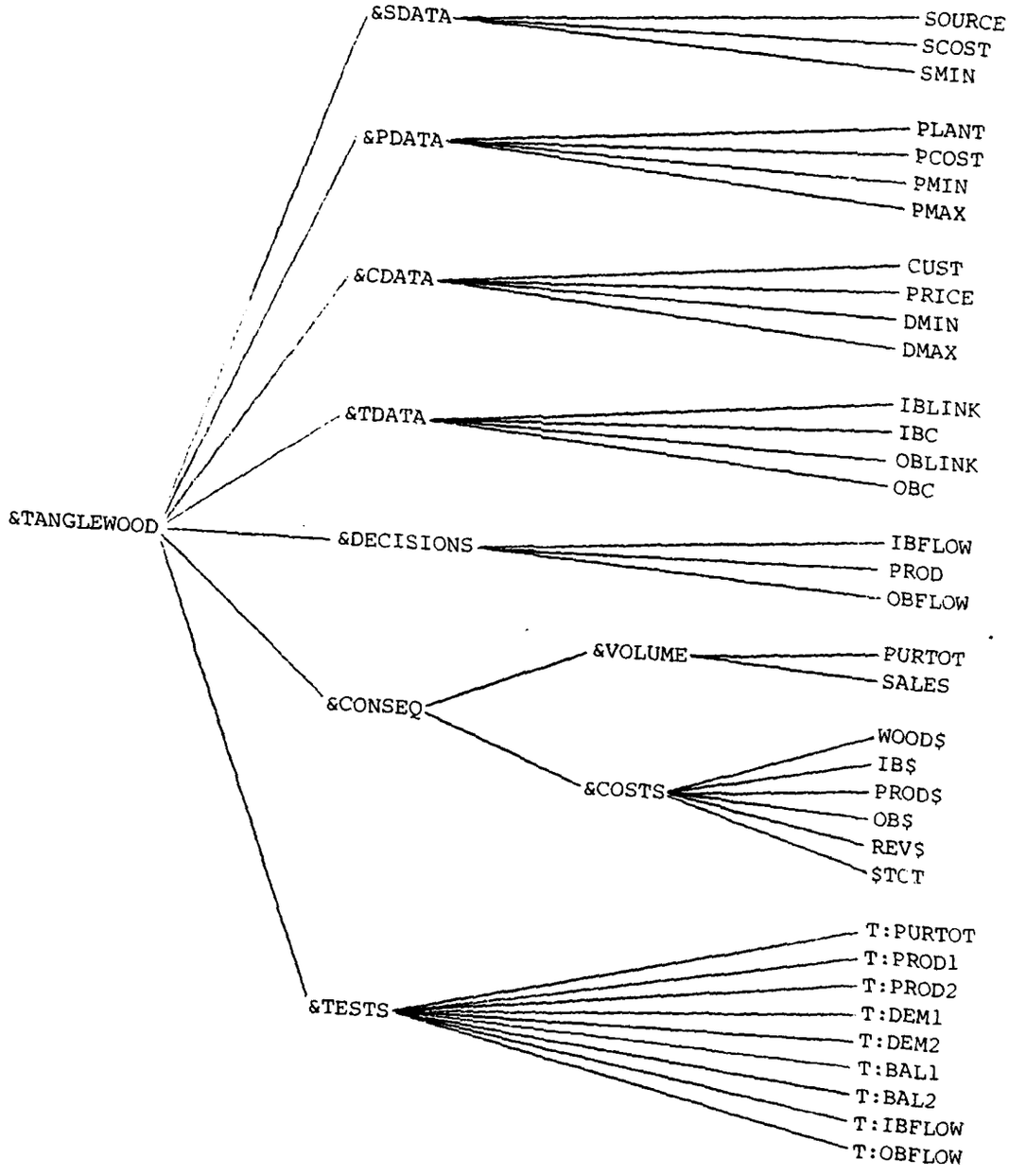


Figure 3.21 TANGLEWOOD Modular Structure

genera either assign values to these elements (attribute and variable attribute genera) or specify rules which relate the elements of one genus to another genus to form real-valued or relational expressions (function and test genera). The intent of each genus is documented in natural language in its master dictionary paragraph.

The element section for this model contains eleven descriptor data blocks. Three blocks are used to value the three domain indices (i,j,k) introduced by the directly-indexed genera (SOURCE,PLANT,CUST). The remaining eight blocks specify values for attribute genera. Values for OBC have been listed without their identifier tuples to demonstrate the format option available to attributes indexed by Cartesian-indirect sets.

IV. SOFTWARE DESIGN

A. DESIGN PRINCIPLES

Structured modeling is an evolving concept. Although a description of a computer environment to support it has been written [Ref. 4: c. 3], the desired functional capabilities will be revised as the technique is practiced. A software design for a prototype, therefore, must be robust. It must be both extensible or contractible as additional or unneeded capabilities are identified. This flexibility has been achieved in this design through adherence to four criteria: subroutinization (we hesitate to use the term modularization in the current context), hierarchical structure, information hiding, and the creation of a virtual machine.

Subroutinization is a mechanism for decomposing a system into partial system descriptors. It differs from the conventional notion of a subroutine as a single sub-program performing one function by its assignment of a task responsibility to groups of interdependent sub-programs. Subroutinizations include design decisions which must be made before work on individual subroutines may begin. This approach enhances flexibility by enabling major revisions to be made to one subroutine without a need to change others. An added benefit of subroutinization is comprehensibility. As system-level functions are clearly defined, the whole system can be better understood.

A hierarchial structure exists in a system, in the sense illustrated by Dijkstra [Ref. 10], if a relation may be defined between the components of the system and that relation is a partial ordering. The relation used in this design is 'uses or depends upon' as defined by Parnas [Ref. 11]. Partial ordering facilitates implementation of a software prototype by providing a usable, testable subset of the system at each level. Coded subroutinizations may be used before the system is complete, avoiding the problem of 'nothing works until everything works' encountered in unstructured designs. The 'uses' relation is characterized in the design by the existence of a large number of single-purpose programs on the lowest level of the structure. These utility routines simplify the implementation of the upper level programs of the system hierarchy.

A system may be difficult to revise if too many programs are written assuming that a particular feature is present or absent. The concept of information hiding is to identify those design decisions which are most likely to change as the prototype matures, and to cloak them in subroutinizations. The changeable features of each subroutine are not disclosed by its interface and remain hidden from other system components which use it. These access restrictions increase the robust nature of the design by hiding details which should not affect other parts of the system. Binding design decisions are, thereby, postponed.

In the sense that a virtual machine is a suite of programs which, when combined with a base machine, provide a machine which is more convenient to use than the underlying machine, a virtual machine has been created to operate on the data types defined by the design. This approach avoids the problems posed by 'a chain of data transforming components' [Ref. 11]. In such a chain, each component receives its input from a previous component, performs its function, and changes the format of the data before passing it to the next stage in the process. If one program is no longer needed and is removed, the output provided by its predecessor is incompatible with the input required by its successor.

B. SYSTEM DESCRIPTION

Two system abstractions are introduced in the following paragraphs in order to make what the system does clearer and more understandable. A procedural abstraction is presented first to express what is done to the master dictionary/element section representation of the model provided by the user. A data abstraction is then presented last to explain the internal representation of the model created by the system software.

1. Procedural Abstraction

The software consists of the seven sub-systems depicted in Figure 4.1. A synopsis of the purpose, input and output of each sub-system is provided below. Since this prototype is designed for linear optimization models, the OPTIMIZE sub-program is treated in greater detail.

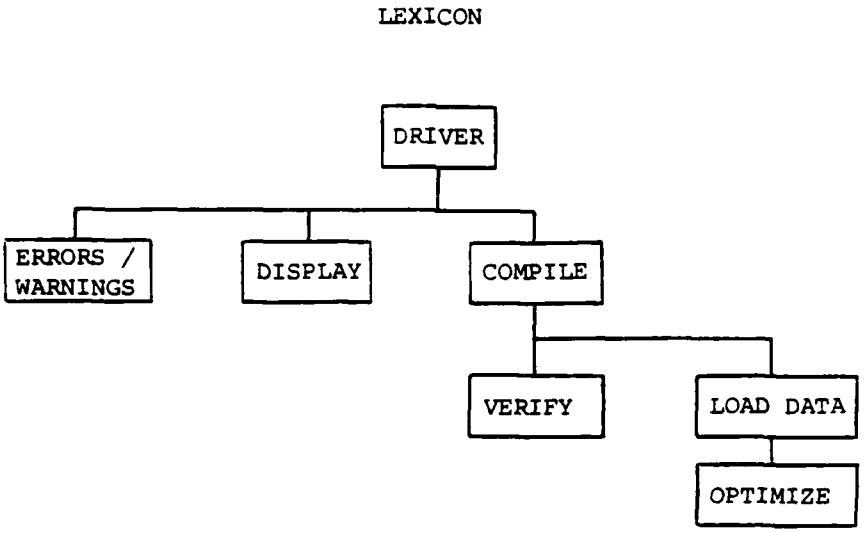


Figure 4.1 LEXICON Procedural Abstraction

a. DRIVER Sub-program System Control

This sub-system displays menu options to the user. It accepts as input the device numbers of designated input and output files, and coded options selected from its menus.

b. ERROR/WARNINGS Sub-program

This sub-system accepts as input unique error codes produced by the five sub-programs listed below. It prints error/warning messages containing these codes and displays the file line(s) or arithmetic expression producing each error up to the point of error recognition.

c. DISPLAY Sub-program

The DISPLAY sub-system allows the user to create different views of the master dictionary representation.

Sub-system options offer the following perspectives of the entire master dictionary file or of the model attribute paragraphs alone:

1) view natural language interpretations of each paragraph only;
2) view technical documentation (paragraphs less interpretation statements) only, or
3) view paragraph names only.

DISPLAY performs its responsibilities without creating an internal representation of the model.

d. COMPILE Sub-program

This sub-system accepts the user-designated master dictionary file as input and creates an internal representation of the model. It ensures that each genus paragraph is in correct format and that its executable statements are consistent with those introduced by preceding paragraphs. When the model is compiled without error, one of two alternatives must be chosen. The user may invoke the VERIFY module to verify the integrity of the internal representation before loading the model data or the model data may be loaded directly.

e. VERIFY Sub-program

The VERIFY sub-system reduces each test genus rule statement in the internal representation to an arithmetic expression composed of variable attributes (variables), attributes (data) and real-valued constants. This is done by

iteratively replacing each occurrence of a function generic name in a test rule statement by the corresponding functional rule statement. After reduction is complete, the sub-program checks each term and sub-expression of the aggregated rule for possible grammatical errors and for agreement with the internally represented model. If an error is discovered during this task, the module sends a unique error code to the ERROR/WARNINGS module and begins verification of the next test rule statement.

f. LOAD DATA Sub-program

This sub-system accepts as input a user-designated element section file. The data extracted from this file are stored in data structures prepared by the COMPILE sub-program from the master dictionary. Errors occurring during data input are identified by unique error codes and their respective file line numbers. After the last data entry has been accepted, the sub-system checks that the data requirements of each genus have been met. The user is notified of violations found by the ERROR/WARNINGS sub-program.

g. OPTIMIZE Sub-program

The OPTIMIZE sub-system performs three primary tasks. First, it prompts the user to formulate a linear program from rule statements of the test genera (constraints) and function genera (candidate objective functions) contained in the model. Its second task is to generate a representation of the optimization model for the solver. This is done

by using the sub-program depicted in Figure 4.2. After the last constraint has been generated, the OPTIMIZE sub-system invokes the solver and displays the results of the optimization.

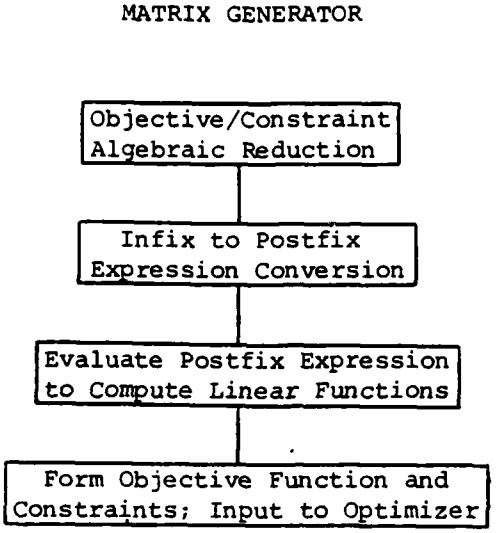


Figure 4.2 Matrix Generator Sub-programs

The solver representation of the LP is a matrix generated in row form by a four-stage process (Figure 4.2). The first stage in the sequence performs the task responsibilities of VERIFY, and screens flaws from the formulation in the following way. Flawed test rule statements are reported to the user and voided from the solver matrix. If an error is detected in the specified objective function, the user is required to replace it with an existing function rule statement or quit the LEXICON system. The output

of the first stage expresses each rule-type in an infix code. The second stage of the transformation converts the infix code to postfix notation. The third stage retrieves real-valued data from the internal structure and evaluates the rule in postfix coding to produce the objective function or a constraint. The fourth stage has three responsibilities. It aggregates multiple instances into a single, real-valued coefficient for each variable in the rule; it aggregates real-valued constants and data to form a row resource value for constraints; and it passes the objective function vector or a canonical row vector to the optimizer. If a test genus is indexed, stages three and four are repeated until a constraint has been generated for each test element in that genus.

2. Data Abstraction

LEXICON requires two input files to create a model which may be manipulated. The master dictionary file documents the conceptual model and is transformed into a machine representation that can be checked for internal consistency. The element section file completes the model instance by specifying the index sets, index functions and attribute values necessary to formulate the case desired.

The internal representation of the conceptual model is composed of genus records, index set records, index function records, attribute table records and lists of character data. Each genus paragraph in the master

dictionary file produces one genus record. The paragraph-to-record mapping may be one-to-many, depending on the genus type and whether a new index set or index function has been introduced by the paragraph. Each paragraph may produce multiple index function records; it may produce at most one of each of the remaining record types.

The components of each record type are shown in Tables 4.1 and 4.2. Lists, accessed by special purpose programs, are used to store genus names, domain indexes, index function names, and rule statements. These are the only positions of the genus paragraph which are stored intact. Access to the internal representation is restricted by the system's virtual machine.

The techniques used to store the index sets, index functions and real-valued data contained in the element section file are principal features of the design. Since identifier-tuples are components of the two latter data types, index set representation and storage will be discussed first.

The fundamental index set is the direct index set introduced by a directly-indexed genus. A direct index set has a defined domain index whose values are specified by an ordered list of identifiers. The system represents this concept by creating an identifier proxy, called an element number, that is equal to the list position of each identifier. Hence, element numbers for a domain index range from 1 to

Table 4.1

GENUS AND INDEX SET RECORD DETAIL

Data Record Type	Fields
1. Genus	a. Character string length of the genus name.
	b. Pointer to the name string in the Genus Name List.
	c. Pointer to the Index Set Record referenced by the genus.
	d. Code designating genus type.
	e. Character string length of the generic type.
	f. Pointer to the rule string in the Genus Rule List.
	g. Pointer to the Attribute Table Record.
	Note: (i) e,f are null for *pe*, *ce*, *va*, and *a* genera. (ii) g is null for *pe*, *ce*, *t* and *f* genera.
2. Index Set	a. Pointer to the Genus Record that introduced the index set.
	b. Code designating the set type.
	c. Logical field, valued 'true' if the index set is a Cartesian product of the underlying direct sets or a direct set itself.
	d. Set cardinality.
	e. Set dimension.
	f. Pointer to the list of tuple numbers computed for each non-Cartesian product index set; or Pointer to the list of identifier string lengths for sets that introduce an index.
	g. Pointer to the list of identifier tokens.
	Note: (i) f is null for index sets derived by Cartesian products of the underlying direct sets. (ii) g is null for index sets that do not introduce an index.

Table 4.2

ATTRIBUTE TABLE AND FUNCTIONAL INDEX RECORD DETAIL

Data Record Type	Fields
1. Attribute Table	a. Pointer to the Genus Record that introduced the attribute table.
	b. Pointer to the Index Set Record referenced by the attribute table.
	c. Pointer to the Data List.
2. Functional Index	a. Pointer to the Index Set Record that provides the index function range.
	b. Index function domain dimension = n.
	c. Pointers to each of the n Index Set Record domain arguments.
	d. Index function cardinality.
	e. Pointer to the list of tuple numbers, computed from the domain argument, and to the range element number mapped from that argument.

the number of identifiers specified for its direct index set. The identifiers of each direct set are explicitly stored in a list. Respective element numbers can be derived, as needed, from their list sequence.

More complex index sets are introduced by genus paragraphs that are not directly-indexed. Each identifier-tuple of an indirectly indexed set may be equivalently expressed as a tuple of the element numbers of its components. For simplicity, and to save storage, each element number tuple is represented as a single integer rather than a tuple. The tuple number preserves the index-induced ordering of the tuples of the indirect index set. An additional benefit of this scheme is that the tuple numbers of directly indexed sets, and indirectly indexed sets that are Cartesian products of the underlying directly indexed sets, need not be stored. Instead, they are computed as required. This distinction is recorded in the index set record logical field (see Table 4.1). The tuple numbers of other indirectly indexed sets are sorted and stored in a list. The list is accessed via a pointer in the index set record.

Each tuple number is a function of the cardinalities of the domain indices over which the set is defined. The largest tuple number that can be computed for a given set is the product of its index cardinalities. For a multi-dimensional set, this number could exceed the largest integer representable on a computer ($2^{31} - 1$ on an IBM 3033, a number over two billion).

Although this bound does limit the dimension and range of index values of indirect sets, we believe that few practical problems will be hindered by this restriction. For example, a set with nine indices, where each index has ten elements, would contain only one billion members.

Each real-valued data element in the model instance is stored in a list. A block of memory locations, equivalent to the cardinality of the index set over which the attribute is defined, is reserved in the list for each attribute genus in the model. Data value storage assignments within a specific attribute genus block are made by converting each value's identifier-tuple to an ordinal position within the referenced index set. Locations within the block which are not assigned values retain a unique initialized value. This coding alerts the matrix generator module to data elements required by the model that remain unspecified.

Functional index data, like attribute data, is stored in a list, partitioned into separate blocks of storage for each index function defined by the model, Unlike attributes, the length of a specific index function block is determined by enumeration of the element section file entries provided for that record. Two values are stored for each index function instance: the tuple number calculated from its identifier-tuple component and the element number of its identifier value. After the last block entry is stored, the block is sorted into an ascending sequence over the tuple number entries.

C. CAPABILITIES DESIGNED BUT NOT IMPLEMENTED

Sub-program interfaces have been designed for two additional capabilities. These represent advanced features of structured modeling that require considerable coding and add relatively little to the evaluation goal of the prototype.

1. Default Index Set

Two limitations of the implemented prototype are the requirements that each genus index set statement be written explicitly, and that index sets which are not Cartesian products of the underlying direct index sets be enumerated in the element section file and explicitly stored. In practice, the user may desire to construct an index set for an indirectly indexed genus that consists of all possible tuples for which the generic calling sequence is well-defined. Geoffrion [Ref. 4] has written an algorithm that uses relational algebra to calculate the default set. When implemented, this feature will free the user from both limitations when the size of an index set is unrestricted.

2. Filter Index Sets

The index sets derivable from generic calling sequences and index set statements are meant to represent the physical relationships which exist in the model. These static relationships are adequately represented by the index set types (direct, indirect-Cartesian, indirect-default, and indirect-select) provided by the design. These sets, however, are not sufficient to represent all sets needed

for computations defined over sets in a richer grammar. Consider the generic rule

sum k {Z} EQ {Q} k LT i (A(i,k))

where

- {Q} is an indirect set defined on domain indices 'i', and 'k';
- A(i,k) is an attribute or function genus indexed by {Q}

The set {Z} is a function of domain index 'i' and may have no physical interpretation outside the context of the summation. (Geoffrion has suggested that if the set {Z} did have physical meaning in a structured model, it should be formally introduced as a compound entity genus.) In general, these dynamically contrived sets, or filter sets, are subsets of the underlying set, created by a filter or rule defined on the domain indices of the underlying set. The software design includes the capability to implement filter sets at a later date.

V. PRACTICAL TAXONOMY

In this chapter we highlight some of the practical aspects of structured models and of the LEXICON system. Section A addresses the issue of model revision. Section B demonstrates the facility of expressing a general linear programming model as an executable, structured equivalent. Section C illustrates the LEXICON user interface through an optimization session vignette. Section D describes the diagnostic features of the modeling software.

A. REVISING A STRUCTURED MODEL

A model does not always progress in an orderly fashion. The practitioner's concept of his model will often change as he gains insight about the inter-relationships and specifications of the system he is attempting to abstract. For this reason, any computer environment designed for modeling must allow the modeler to detail, revise, and, if necessary, undo this work.

A fundamental issue in the design of a structured modeling prototype is whether the user should be allowed to modify the internal representation of his model without changing its external form. The decision not to allow internal manipulation simplifies the design and maintains the principle that the master dictionary/element section files and the internal representation portray the same model.

Thus, we have decreed that the model can be modified only by changing the master dictionary/element section files.

The organizing framework provided by structured modeling enables model flexibility by helping the practitioner codify his concept into a hierarchy of conceptual modules. This modular structure provides two benefits. First, the model is truly separate from the data. Recall the Tanglewood Chair Manufacturing Company problem intruduced in Section III.E. For example, the number of plants, the number of sources and the real-valued specifications of the Tanglewood model can all be changed without modifying the conceptual model expressed by the master dictionary. Second, stepwise-refinements of non-entity genera, and changes which do not violate the order of the modular structure are easily accommodated. This latter feature is particularly valuable when the real-valued data provided must be scaled or aggregated to a more convenient form.

Two examples are presented to illustrate how revisions might be made in practice to an existing structured model. An original assumption made when the Tanglewood master dictionary (Appendix A) was prepared was that all plants required 20 lbs. of wood to produce one chair. This figure was explicitly used in the T:BAL1 paragraph's rule statement,

```
SUM i {SOURCE} (IBFLOW(i,j)) EQ 20.0 * PROD(j)
```

to specify that total wood purchases must match production at each plant. Suppose that new machinery has been installed

at plant 2 that reduces the amount of scrap produced in the manufacturing process. Plant 2 now requires only 18 lbs./ chair. How would the model be revised to reflect this change?

This factor is clearly a real-valued specification, associated with each plant, that belongs conceptually in the &PDATA module. Two dictionary edits, followed by one element section edit are required to affect the change. The first edit introduces the genus attribute paragraph

```
PMAT (PLANT(j)) /a/ {PLANT} :: Each PLANT has a unit
..MATERIAL REQUIREMENT in lbs/chair.
```

in the &DATA module in any position after PLANT. This placement guarantees that the calling sequence statement and index set statement of PMAT are well-defined. The second edit revises the T:BAL1 paragraph's rule statement and calling sequence statement:

```
T:BAL1 (IBFLOW(*,j),PMAT(j),PROD(j)) /t/ {PLANT}
.. ; SUM i {SOURCE} (IBFLOW(i,j)) EQ PMAT(j)*
.. PROD(j) :: Do total WOOD PURCHASES match
..PRODUCTION at each PLANT?
```

The calling sequence statement now reflects that each element of the genus refers to one element of PMAT. The nature of this reference is the PMAT(j) term in the rule statement. Note that the derived domain index of the genus is still 'j'. The last edit inserts the following data block into the element section file

```
&ATR^^PMAT
^^^PL1^^^20.0
^^^PL2    18.0
^^^PL3    20.0
```

The next change made to the Tanglewood problem demonstrates a transition from the low-level data available from the physical system to the high-level data required by the model. Suppose that the freight rate from each source to each plant, IBC, is not available explicitly for each transportation link. Instead, each individual rate must be computed as the product of a fixed factor and the highway mileage from each source to each plant.

The first step in this transformation is to write a genus paragraph for each kind of data. Since mileage is a real-valued specification defined for each inbound transportation link IBLINK(I,J), it should be expressed as an IBLINK-related attribute

```
IBMILES (IBLINK(i,j)) /a/ {IBLINK}:: There is an
.. INBOUND TRANSPORTATION MILEAGE for each INBOUND
.. TRANSPORTATION LINK expressed in miles.
```

The fixed factor can be expressed as an unindexed attribute genus or, like IBMILES, can be indexed by IBLINK. The later alternative requires that a separate data entry be made for each transportation link. Indexing the paragraph would be a preferred approach if the cost factor is likely to vary by origin or destination in the future. The unindexed

paragraph for the cost factor genus is

```
IB$RATE (IBLINK) /a/ 1 :: All INBOUND TRANSPORT-
..ATION LINKS have the same fixed BASIC INBOUND
..FREIGHT RATE in $ per pound of wood.
```

IB$RATE's status as an unindexed genus is apparent by its lack of domain indices in its calling sequence statement and its index statement's binary value. Both paragraphs would be introduced in the &TDATA module after IBLINK but before IBC.

The second step in this transformation is to rewrite the IBC genus in the format prescribed for a function genus.

```
IBC (IBMILES(i,j),IBSRATE) /f/ {IBLINK}; IBMILES(i,j)
..*IBSRATE :: Computed INBOUND FREIGHT RATE for
..each INBOUND TRANSPORTATION LINK.
```

The new function genus inherits its index set {IBLINK} from the IBMILES component of its calling sequence statement. These three new paragraphs complete the data transition. No other paragraphs which call the IBC genus need know that its elements are computed rather than supplied as element section data. An instance of the new model would be specified by removing the data descriptor block provided for IBC and inserting two new blocks for the two new attribute genera. Block sequence would parallel the sequence used in the master dictionary.

The conceptual hierarchy thus allows changes to be made in a module without having any impact on any other module.

Modular structure, however, is not a panacea. If the practitioner makes so many changes that his conceptual model is no longer valid, he may be able to recover very little of his previous work.

B. FORMULATING A LINEAR PROGRAM AS A STRUCTURED MODEL

All abstractions used in the traditional linear programming approach [Ref. 12: pg. 34] to model building have structured model counterparts. Figure 5.1 exhibits a general linear programming model expressed in structured modeling notation. The same model in algebraic notation appears below.

Symbol	Definition
i	Resource i in the set of resources I
j	Activity j in the set of activities J
b_i	Availability of resource i
x_j	Activity level of activity j
a_{ij}	The quantity of resource i consumed by operating activity j at its unit level
c_j	The cost of operating activity j at its unit level

Objective

$$\sum_j c_j x_j$$

subject to

$$\sum_j a_{ij} x_j \leq b_i \qquad i \in I$$

$$x_j \geq 0 \qquad j \in J$$

```
&RDATA :: Certain RESOURCE Data are supplied.

   RESOURCE i /pe/ :: There is a list of RESOURCES.

   B (RESOURCE(i)) /a/ {RESOURCE} :: Every RESOURCE has its
   .. AVAILABILITY in resource units.

&ADATA :: Certain ACTIVITY DATA are supplied.

   ACTIVITY j /pe/ :: There is a list of ACTIVITIES.

   X (ACTIVITY(j)) /va/ {ACTIVITY} :: Every ACTIVITY has an
   .. operating LEVEL measured in activity units.

   C (ACTIVITY(j)) /a/ {ACTIVITY} :: Every ACTIVITY has its
   .. UNIT COST in $ per activity unit.

A (RESOURCE(i),ACTIVITY(j)) /a/ {RESOURCE}x{ACTIVITY} ::
.. There are COEFFICIENTS that indicate the quantity of each
.. RESOURCE consumed by operating each ACTIVITY at its unit
.. Level.

&RESULTS :: Certain RESULTS follow from the given DATA and
.. the choice of ACTIVITY LEVELS.

   OBJ$ (C,X) /f/ 1 ; SUM j {ACTIVITY} (C(j) * X(j)) ::
   .. Operating all ACTIVITIES at their LEVELS incurs a
   .. TOTAL OPERATING COST.

   T:RES (A(i,.),X,B(i)) /t/ {RESOURCE} SUM j {A} (A(i,j) *
   .. X(j)) LE B(i) :: Does the TOTAL CONSUMPTION of each
   .. RESOURCE fall within its AVAILABILITY?

   T:NEG (X(j)) /t/ {ACTIVITY} ; X(j) GE 0.0 :: Does each
   .. ACTIVITY LEVEL obey the non-negativity assumption?
```

Figure 5.1 Structured LP Model Master Dictionary

For the purposes of this illustration, assume that the solution algorithm accepts only minimization problems. The first common property of these forms is that both assert the existence of fundamental elements called resources and activities. Resource and activity elements are represented in the algebraic form as the index sets I and J. These same elements appear in the structured LP model (Figure 5.1) as members of the primitive entity genera RESOURCE and ACTIVITY. Each primitive entity genus introduces an index set: {RESOURCE} indexed by 'i' and {ACTIVITY} indexed by 'j'.

The resource levels (b_i), cost factors (c_j), activity levels or decision variables (x_j) and technological coefficients (a_{ij}) defined in Figure 5.1 are real-valued data. Each algebraic symbol represents a quantity or property associated with resource elements, activity elements, or resource-activity interactions. These real-valued traits are manifested in the master dictionary as the attribute genera B, C, X and A, respectively. Note that X is specified as a variable attribute because the values it imparts to the models' activity elements are unknown.

The mathematical equivalence of the two LP representations is confirmed by observing that a rule statement exists in the master dictionary for each linear functional in the algebraic form. These rule statements are contained in the function and test genera paragraphs. The objective function is the unindexed function genus OBJ$. Each constraint is

an element of the test genus, T:RES. T:RES, like the algebraic constraint, is indexed by the resource set. The last test genus, T:X, specifies in its rule statement that each decision variable must be non-negative. Its index set is {ACTIVITY}.

Once an appropriate element section has been specified, the structured linear programming problem can be executed to solve for the X(j) decision variables. The complete structured LP problem would be stated as

Minimize OBJ$ subject to T:RES, T:NEG, by choice of X.

Exactly how a modeler would interact with the LEXICON modeling system to execute a structured LP model is explained in the next section.

C. OPTIMIZATION USING LEXICON

The aim of this section is to sketch how someone would interact with LEXICON to solve a specific linear program. Assume that the problem is in the same canonical class or 'schema' as the Tanglewood Chair Company model in Chapter III and that an element section file has been prepared to specify the desired instance (Appendix A). Since LEXICON requires an interactive computing environment, the session is presented as a series of 'screens' interspersed with commentary. System responses are prefaced with a '*' in each mock video display line to distinguish them from the responses of the user. For reasons of brevity and clarity,

both external files in the scenario are error-free. (Discussion of LEXICON error procedures is deferred to Section D.)

1. Screen 1

The session begins with the prompts and responses shown below.

```
*LEXICON-VERSION 1.0
*SPECIFY OUTPUT FILEDEF, ELSE ENTER '6' FOR SCREEN
6
*ENTER MASTER DICTIONARY FILEDEF
11
* SELECT PROCEDURE DESIRED: ENTER
*      "DISPLAY" TO EXAMINE DIFFERENT MASTER
                 DICTIONARY VIEWS
*      "COMPILE" TO EXECUTE THE MODEL, OR
*      "QUIT"    TO EXIT
DISPLAY
```

The user designates the TANGLEWOOD model, one of several validated models he maintains, as the master dictionary. For convenience, he routes all session output directly to the video terminal rather than create a listing in his disk space.

2. Screen 2

The "DISPLAY" command entered in Screen 1 allows the user to select one of the sixteen master dictionary reformatting alternatives supported by the system.

```
*SELECT DESIRED VIEW: ENTER
*      CODE          PRESENTATION
*       1            ALL PARAGRAPHS
*       2            ATTRIBUTE PARAGRAPHS ONLY
2
*SELECT RESOLUTION: ENTER
*      CODE          PARAGRAPH FORMAT
*       1            GENUS NAMES ONLY
*       2            INTERPRETATIONS ONLY.
*       3            OMIT INTERPRETATIONS
3
*ENTER "YES" IF LINE NUMBERS ARE DESIRED
YES
```

The perspective selected is shown in Figure 5.2. After scanning this dictionary excerpt, the modeler concludes that the element section file prepared satisfies the real-data needs of the model. As no indentation or paragraph length errors were detected, the system issues an error-free completion message

```
* VIEW COMPLETE
```

and prompts the user to specify the next procedure.

```
.COMPILE
```

3. Screen 3

LEXICON responds to the last command by compiling the TANGLEWOOD master dictionary into an internal structure ready to receive data. No departures from notational convention or structural consistency are found.

```
* DICTIONARY COMPILED: MODEL IS READY TO ACCEPT DATA
* SELECT NEXT PROCEDURE: ENTER
*     "VERIFY" TO CHECK SYNTAX AND BASIC
*              PROPERTIES OF FUNCTION AND TEST
*              GENERA BEFORE LOADING DATA, OR
*     "LOAD"   TO LOAD THE ELEMENT SECTION FILE
*              NOW
```

The user decides to skip the intermediate procedure and load the model data directly. (If the problem added genus paragraphs to the TANGLEWOOD master dictionary or revised its executable fields, a prudent user would verify the model before loading data.)

```
LOAD
* SPECIFY ELEMENT SECTION FILEDEF
12
```

```
 5  SCOST (SOURCE(i)) /a/ {SOURCE} :: Each SOURCE has a UNIT
 6  .. WOOD COST in $/lb.
 7
 8  SMIN (SOURCE(i)) /a/ {SOURCE} :: Each SOURCE has a MINI-
 9  ..MUM WOOD PURCHASE in pounds/month.
10
16  PCOST (PLANT(j)) /a/ {PLANT} : R+ :: Each PLANT has a
17  .. UNIT PRODUCTION COST in $/chair.
18
19  PMIN (PLANT(j)) /a/ {PLANT} : R+ :: Each PLANT has a
20  .. MINIMUM PRODUCTION limit in chairs/month.
21
22  PMAX (PLANT(j)) /a/ {PLANT} : R+ :: Each PLANT has a
23  .. MAXIMUM PRODUCTION limit in chairs/month.
24
30  PRICE (CUST(k)) /a/ {CUST} : R+ :: There is a SELLING
31  .. PRICE in $/chair for each customer.
32
33  DMIN (CUST(k)) /a/ {CUST} : R+ :: Each CUSTOMER has a
34  .. MINIMUM DEMAND in chairs/month.
35
36  DMAX (CUST(k)) /a/ {CUST} : R+ :: Each CUSTOMER has a
37  .. MAXIMUM DEMAND in chairs/month.
38
45  IBC (IBLINK(i,j)) /a/ {IBLINK} :: There is an INBOUND
46  .. FREIGHT RATE for each INBOUND TRANSPORTATION LINK in
47  .. $ per pound of wood.
48
53  OBC (OBLINK(j,k)) /a/ {OBLINK} :: There is an OUTBOUND
54  .. FREIGHT RATE for each OUTBOUND TRANSPORTATION LINK in
55  .. $ per chair.
56
```

Figure 5.2 TANGLEWOOD Attribute Perspective

Element detail is read from the element section file to fill the requirements of the model. After the data has been stored, LEXICON completes the internal structure and signals

```
* READY TO GENERATE A LINEAR PROGRAM.
```

4. Screen 4

The next sequence of prompts requires the user to specify a linear programming model from the function and test genera contained in the master dictionary.

```
* SELECT CONSTRAINTS: ENTER
*       "SELECT" TO GENERATE A SUBSET OF THE MODEL
*                TEST GENERA, OR
*       "ALL"
```

"SELECT" offers the user the opportunity to insert or exclude families of constraints (test genera) in/from his linear programming formulation. Since every test genera is of interest, the

```
ALL
```

response is entered.

```
* ENTER OBJECTIVE FUNCTION GENUS NAME
$TOT
* SPECIFY OPTIMIZATION: ENTER
*    "MAX"      TO MAXIMIZE, OR
*    "MIN"      TO MINIMIZE
MIN
```

Provision of the linear objective function and the optimization mode are the last interactive inputs required by the system.

5. The Solution

LEXICON uses the linear programming model and the internally stored data to generate the LP in a form compatible with the optimizer. After the solution is obtained, the system stores the calculated values of the model's variable attributes in the internal structure and issues the report shown in Figure 5.3. Additional optimization output is provided by the attached optimizer for each test element (constraint).

D. LEXICON ERROR PROCEDURES

LEXICON has extensive features for detecting compilation and data input errors. The system is capable of detecting over 100 different departures from notational convention and structured model properties in a master dictionary file alone. When an element section file is submitted to A- partially specify a compiled model, over 30 additional data- model and format inconsistencies can be identified. Although these abilities do not help the user create the 'right' model, they can speed the development of a grammatically correct and internally consistent representation of the problem.

The model debugging cycle, diagrammed in Figure 5.4, begins when a master dictionary file and element section file are initially submitted for processing. It ends when an optimized solution for the model specified by these files is obtained. To reach the terminal state, the model must pass

OBJECTIVE: $TOT OPTIMAL VALUE = -2.3110E+04

LISTING OF ATTRIBUTE: IBFLOW
INDEX SET: IBLINK SET TYPE: CARTESIAN NO. OF ELEMENTS: 8

NUMBER	VALUE	DOMAIN INDICES: I	J
1	1.0000E+04	1	1
2	0.6000E+04	1	2
3	0.0	1	3
4	0.0	1	4
5	0.0	2	1
6	0.9000E+04	2	2
7	2.0000E+04	2	3
8	0.5000E+04	2	4

LISTING OF ATTRIBUTE: PROD
INDEX SET: PLANT SET TYPE: DIRECT NO. OF ELEMENTS: 4

NUMBER	VALUE	DOMAIN INDICES: J
1	0.5000E+03	1
2	0.7500E+03	2
3	0.1000E+04	3
4	0.2500E+03	4

LISTING OF ATTRIBUTE: OBFLOW
INDEX SET: OBLINK SET TYPE: CARTESIAN NO. OF ELEMENTS: 16

NUMBER	VALUE	DOMAIN INDICES: J	K
1	0.0	1	1
2	0.0	1	2
3	0.0	1	3
4	0.5000E+03	1	4
5	0.7500E+03	2	1
6	0.0	2	2
7	0.0	2	3
8	0.0	2	4
9	0.0	3	1
10	0.0	3	2
11	0.1000E+04	3	3
12	0.0	3	4
13	0.0	4	1
14	0.2500E+03	4	2
15	0.0	4	3
16	0.0	4	4

Figure 5.3 Tanglewood Solution in LEXICON Format

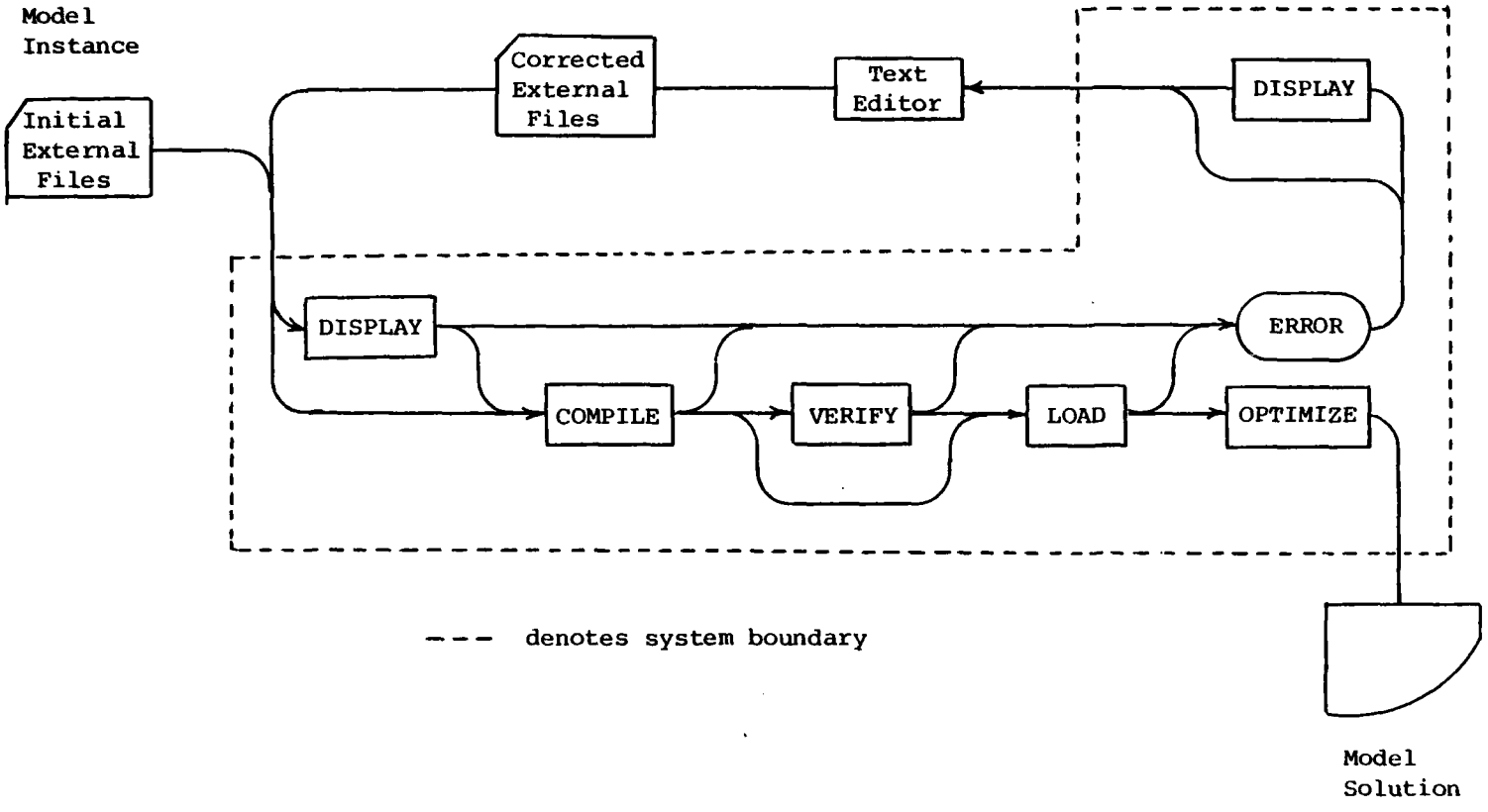


Figure 5.4 Model Debugging Cycle

each LEXICON sub-program entered without error. If an error occurs, the model is denied access to all subsequent LEXICON procedures until the fault is corrected. Due to the decision not to allow the user to modify the model's internal representation, no run-time corrections can be made. Error occurrences require the user to quit the LEXICON environment and correct the responsible model file before the cycle can be continued.

Debugging either file is made easier by two system diagnostic aids: unambiguous error messages and line-numbered master dictionary perspectives. Each error detected during compilation or data input is identified by a unique code and located, at a minimum, by the file line number of the fault or the name of its genus paragraph. The exact format of the error message is both fault and module dependent. After all error messages have been issued, the system offers the user the opportunity to create and browse a complementing, line-numbered display prior to leaving the system. (Text editor and LEXICON line numbering conventions are the same.) When both features are used in concert, faults can be diagnosed and located quickly during the correction step of the cycle.

Three eclectic error messages, based on the Tanglewood model (Appendices A,B) are presented below. Each has been contrived to illustrate a different software error diagnostic ability. The first example:

```
41  IBLINK (SOURCE(i),PLANT(j))) /ce/ {SOURCE} {
FATAL COMPILATION ERROR 2404 RECOGNIZED AT LAST
PRINTED CHARACTER
```

is representative of the messages produced for errors detected during manipulation or compilation of the master dictionary file. It contains a file line number label, a partially displayed paragraph, and an error code. The paragraph display is truncated at the point in the line text where the error is recognized. This particular message indicates that the IBLINK genus paragraph, located at file line 41, contains a fault in its index set statement. Errorcode 2404 means that the index set operator, required between SOURCE and the beginning of the next index set name, is missing. (The correct statement is {SOURCE}x{PLANT}.)

Compilation errors in rule statements are signalled using a different format. The example message

```
TBAL1 - SUM i {PLANT} (IBFLOW(i)
FATAL RULE COMPILATION ERROR 521 RECOGNIZED AT LAST
PRINTED CHARACTER
```

includes a genus name label, the partially displayed rule statement of that genus, and an error code. Like the first format, the point of truncation is the position where the rule was no longer decipherable. A file line number is not provided because the rule statement text is internally stored. The message instance states that a fault has been found in the displayed term of the T:BAL1 genus rule statement. Error code 521 pinpoints the problem:

too few domain index subscripts have been provided for a generic name. (The IBFLOW paragraph in Appendix A is indirectly indexed by i and j.)

The third example is characteristic of the messages issued for model-data inconsistencies found in the element section file.

```
ELEMENT SECTION FILE LINE 41
CONTAINS ERROR 5324
```

Although the format is terse, the information content of the message is sufficient to locate the offending line using a text editor and to correct the fault. This particular message refers to the first data record in the DMAX attribute block (see Appendix B). Error code 5324 means that the identifier tuple listed on this line is not a member of the index set specified for the attribute genus by the master dictionary. An instance of this fault would be obtained by replacing the correct line

```
NY^^500.0
```

by

```
PLT1^^500.0
```

LEXICON would detect that PLT1 is not a member of the set (NY,H,SF,C) and issue the cited diagnostic. (NY,H,SF,C) is the set of identifiers inherited by DMAX in its index set statement from CUST. (See Appendix B.)

VI. CONCLUSION

This paper describes a modeling language and a suite of computer programs for obtaining an executable structured linear programming model. The prototype software contains in excess of 4000 lines of FORTRAN source code, over 50 percent of which are comments. A very high standard of self-documentation has been followed to enable additional capabilities to be added later, such as a sophisticated data editor and/or report writer. FORTRAN was chosen as the host language because of its availability.

Although the LEXICON system is experimental, we do not consider it to be limited to textbook-size problems. The language it accepts has symbolic indexing sufficient to express large-scale LP models. The data storage required to run the system is linear in the size of the parameter data. Moreover, the algorithm form produced is submitted to a commercial-quality optimizer capable of solving mixed-integer linear programs (MIPs) as well as traditional LPs. LEXICON can be extended to accept MIP formulations by making the genus paragraph range statement an executable field. This enhancement and the incorporation of extensive data management, and report writing capabilities, e.g., ATHENA [Ref. 13], would be essential in a production version of the system.

A survey of the literature performed by Fourer [Ref. 3: pp. 164-166] and our own review of papers published since 1981 [Refs. 14,15] identify fifteen computation-capable LP modeling language implementations since 1970. We believe that LEXICON embodies more of the characteristics of an ideal modeling language [Ref. 3: pp. 172-174] than any of its predecessors. Its notation is powerful and understandable, and its structured modeling framework enforces a model organization which can be comprehended by a computer in one pass. Another advantage of this form is its acyclic nature: the model is guaranteed to be finite and closed.

There are drawbacks to using structured modeling as a basis for a modeling language. Implicit in the argument for this approach is that modelers will find it worth the trouble. Top-down design, an intrinsic discipline of structured modeling, can be useful in dealing with complexity, but it is not always a natural way to model. Also, the modular structure of a model is more conceptual than operational. Structured model modules are context dependent: their interfaces to other modules are not simple and preclude their immediate use in building other models.

In summary, the implementation we have achieved is ambitious and is ready for a full-scale computational evaluation. We hope our work stimulates further developments in modeling software which ultimately lead to real-time decision-making with linear programming.

APPENDIX A

TANGLEWOOD CHAIR MANUFACTURING COMPANY
MASTER DICTIONARY

```
&SDATA :: WOOD SOURCE DATA

    SOURCE i /pe/ :: There are WOOD SOURCEs available.

    SCOST (SOURCE(i)) /a/ {SOURCE} :: Each SOURCE has a UNIT
    .. WOOD COST in $/lb.

    SMIN (SOURCE(i)) /a/ {SOURCE} :: Each SOURCE has a MINI-
    ..MUM WOOD PURCHASE in pounds/month.

&PDATA :: PLANT DATA

    PLANT j /pe/ :: There are PLANTS that produce wooden
    .. chairs.

    PCOST (PLANT(j)) /a/ {PLANT} : R+ :: Each PLANT has a
    .. UNIT PRODUCTION COST in $/chair.

    PMIN (PLANT(j)) /a/ {PLANT} : R+ :: Each PLANT has a
    .. MINIMUM PRODUCTION limit in chairs/month.

    PMAX (PLANT(j)) /a/ {PLANT} : R+ :: Each PLANT has a
    .. MAXIMUM PRODUCTION limit in chairs/month.

&CDATA :: CUSTOMER DATA

    CUST k /pe/ There are CUSTOMER CITIES where chairs are
    .. sold.

    PRICE (CUST(k)) /a/ {CUST} : R+ :: There is a SELLING
    .. PRICE in $/chair for each CUSTOMER.

    DMIN (CUST(k)) /a/ {CUST} : R+ :: Each CUSTOMER has a
    .. MINIMUM DEMAND in chairs/month.

    DMAX (CUST(k)) /a/ {CUST} : R+ :: Each CUSTOMER has a
    .. MAXIMUM DEMAND in chairs/month.

&TDATA :: TRANSPORTATION DATA

    IBLINK (SOURCE(i),PLANT(j)) /ce/ {SOURCE}x{PLANT} ::
    .. There is an INBOUND TRANSPORTATION LINK from every
    .. SOURCE to every PLANT.
```

```
    IBC (IBLINK(i,j)) /a/ {IBLINK} :: There is an INBOUND
    .. FREIGHT RATE for each INBOUND TRANSPORTATION LINK in
    .. $ per pound of wood.

    OBLINK (PLANT(j),CUST(k)) /ce/ {PLANT}x{CUST} :: There
    .. is an OUTBOUND TRANSPORTATION LINK from every PLANT
    .. to every CUSTOMER.

    OBC (OBLINK(j,k)) /a/ {OBLINK} :: There is an OUTBOUND
    .. FREIGHT RATE for each OUTBOUND TRANSPORTATION LINK in
    .. $ per chair.

&DECISIONS :: Certain DECISIONS must be made.

    IBFLOW (IBLINK(i,j)) /va/ {IBLINK} : R+ :: WOOD PUCHASES
    .. (inbound flows) must be decided: how many pounds of
    .. wood per month are shipped over each INBOUND TRANS-
    .. PORTATION LINK.

    PROD (PLANT(j)) /va/ {PLANT} : R+ :: PRODUCTION must be
    .. decided: how many chairs per month each PLANT
    .. produces.

    OBFLOW (OBLINK(j,k)) /va/ {OBLINK} : R+ :: CUSTOMER
    .. SHIPMENTS (outbound flows) must be decided: how many
    .. chairs per month are shipped over each OUTBOUND
    .. TRANSPORTATION LINK.

&CONSEQ :: Operating CONSEQUENCES of DECISIONS

    &VOLUME :: VOLUME CONSEQUENCES

        PURTOT (IBFLOW(i,.) /f/ {SOURCE} ; SUM j {IBLINK}
        .. (IBFLOW(i,j)) :: TOTAL PURCHASES from each
        .. SOURCE in pounds of wood per month.

        SALES (OBFLOW(.,k) /f/ {CUST} ; SUM j {OBLINK}
        .. (OBFLOW(j,k)) :: SALES to each CUSTOMER in
        .. chairs per month.

    &COSTS :: COST CONSEQUENCES

        WOOD$ (SCOST,PURTOT) /f/ 1 ; SUM i {SOURCE} (SCOST(
        .. i)*PURTOT(i)) :: WOOD COST in $/month.

        IB$ (IBFLOW,IBC) /f/ 1 ; SUM i,j {IBLINK} (IBFLOW(i
        .. ,j)*(IBC(i,j)) :: INBOUND TRANSPORTATION COST in
        .. $/month.

        PROD$ (PCOST,PROD) /f/ 1 ; SUM j {PLANT} (PCOST(j)*
        .. PROD(j)) :: PRODUCTION COST in $/month.
```

```
        OB$ (OBFLOW,OBC) /f/ 1 ; SUM j,k {OBLINK} (OBFLOW(j
        .. ,k)*(OBC(j,k)) :: OUTBOUND TRANSPORTATION COST
        .. in $/month.

        REV$ (PRICE,SALES) /f/ 1 ; SUM k {CUST} (PRICE(k)*
        .. SALES(k)) :: REVENUES in $/month.

        $TOT (WOOD$,PROD$,IB$,OB$,REV$) /f/ 1 ; (WOOD$+PROD$
        .. +IB$+OB$) - REV$ :: TOTAL COST in $/month.

&TESTS :: The DECISIONS are subjected to certain TESTS.

    T:PURTOT (PURTOT(i),SMIN(i)) /t/ {SOURCE} ; PURTOT(i) GE
    .. SMIN(i) :: Do TOTAL PURCHASES satisfy the MINIMUM
    .. WOOD PURCHASE requirement for each SOURCE?

    T:PROD1 (PROD(j),PMIN(j)) /t/ {PLANT} ; PROD(j) GE PMIN(
    .. j) :: Does PRODUCTION satisfy the MINIMUM PRODUCTION
    .. limit for each PLANT?

    T:PROD2 (PROD(j),PMAX(j)) /t/ {PLANT} ; PROD(j) LE PMAX(
    .. j) :: Does PRODUCTION satisfy the MAXIMUM PRODUCTION
    .. limit for each PLANT?

    T:DEM1 (SALES(k),DMIN(k)) /t/ {CUST} ; SALES(k) GE DMIN(
    .. k) :: Do SALES satisfy the MINIMUM DEMAND requirements
    .. for each customer?

    T:DEM2 (SALES(k),DMAX(k)) /t/ {CUST} ; SALES(k) LE DMAX(
    .. k) :: Do SALES satisfy the MAXIMUM DEMAND requirements
    .. for each customer?

    T:BAL1 (IBFLOW(.,j),PROD(j)) /t/ {PLANT} ; SUM i {IBLINK}
    .. (IBFLOW(i,j)) EQ 20.0*PROD(j) :: Does total WOOD PUR-
    .. CHASES match PRODUCTION at each PLANT?

    T:BAL2 (PROD(j),OBFLOW(j,.)) /t/ {PLANT} ; PROD(j) EQ SUM
    .. k {OBLINK} (OBFLOW(j,k)) :: Does PRODUCTION at each
    .. PLANT match its total CUSTOMER SHIPMENTS?

    T:IBFLOW (IBLINK(i,j)) /t/ {IBLINK} ; IBFLOW(i,j) GE 0.0
    .. :: Do INBOUND FLOWS satisfy non-negativity?

    T:OBFLOW (OBLINK(j,k)) /t/ {OBLINK} ; OBFLOW(j,k) GE 0.0
    .. :: Do OUTBOUND FLOWS satisfy non-negativity?
```

APPENDIX B

TANGLEWOOD CHAIR MANUFACTURING COMPANY
ELEMENT SECTION

```
&SET  SOURCE
  SUP1
  SUP2
&ATR  SCOST
  SUP1  0.1
  SUP2  0.075
&ATR  SMIN
  SUP1  16000.0
  SUP2  16000.0
&SET  PLANT
  PLT1
  PLT2
  PLT3
  PLT4
&ATR  PCOST
  PLT1  5.0
  PLT2  7.0
  PLT3  3.0
  PLT4  4.0
&ATR  PMIN
  PLT1  0.0
  PLT2  400.0
  PLT3  500.0
  PLT4  250.0
&ATR  PMAX
  PLT1  500.0
  PLT2  750.0
  PLT3  1000.0
  PLT4  250.0
&SET  CUST
    NY
     H
    SF
     C
&ATR  PRICE
    NY  20.0
     H  15.0
    SF  20.0
     C  18.0
```

```
&ATR  DMIN
    NY  500.0
     H  100.0
    SF  500.0
     C  500.0
&ATR  DMAX
    NY  2000.0
     H  400.0
    SF  1500.0
     C  1500.0
&ATR  IBC
  SUP1  PLT1  0.01
  SUP1  PLT2  0.02
  SUP1  PLT3  0.04
  SUP1  PLT4  0.04
  SUP2  PLT1  0.04
  SUP2  PLT2  0.03
  SUP2  PLT3  0.02
  SUP2  PLT4  0.02
&ATR  OBC
  1.0
  1.0
  2.0
  0.0
  3.0
  6.0
  7.0
  3.0
  3.0
  1.0
  5.0
  3.0
  8.0
  2.0
  1.0
  4.0
```

LIST OF REFERENCES

1. Little, J.D.C., "Models and Managers: The Concept of a Decision Calculus," Management Science, V. 16, No. 8, pp. 466-485, April 1970.

2. Thomas, G. and Mitchell, M., "Operations Research in the Marine Corps: A Characterization," Interfaces, V. 13, No. 3, pp. 82-90, June 1983.

3. Fourer, R., "Modeling Languages Versus Matrix Generators for Linear Programming," ACM Transactions on Mathematical Software, V. 9, No. 2, pp. 143-183, June 1983.

4. Geoffrion, A.M., Structured Modeling and Aggregation, unpublished manuscript, March 1984.

5. Brown, G., and Graves, G., "Design and Implementation of a Large Scale (Mixed Integer, Nonlinear) Optimization System," paper presented at ORSA/TIMS Meeting, Las Vegas, Nevada, November 1975.

6. IBM Data Processing Division Publication GX20-1703-9, IBM System/360 Reference Data, pp. 11-14.

7. Day, A.C., Compatible Fortran, Cambridge University Press, 1978.

8. James, G., James, R.C., and others, Mathematics Dictionary, 3d ed., Van Nostrand Reinhold, 1968.

9. Jensen, P.A., and Barnes, J.W., Network Flow Programming, Wiley, 1980.

10. Dijkstra, E.W., "Structured Programming," In Software Engineering Techniques, J.N. Buxton and B. Randall (Eds.), NATO Science Committee, 1970, pp. 84-88.

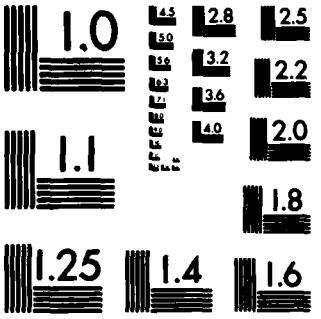
11. Parnas, D.L., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, V. SE-5, No. 2, pp. 128-138, March 1979.

12. Dantzig, G.B., Linear Programming and Extensions, 6th ed., Princeton University Press, 1974.

13. Naval Postgraduate School Technical Report NPS-52-80-006, "ATHENA: User's Manual for Interactive Analysis of Large-Scale Optimation Models," by G.H. Bradley, G.G. Brown and P.I. Galatas, April 1980.

AD-A150 536 LEXICON: A STRUCTURED MODELING SYSTEM FOR OPTIMIZATION 2/2
(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R D CLEMENCE
JUN 84

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

14. Dolk, D.R., "The Use of Abstractions in Model Management," Ph.D. Thesis, University of Arizona, 1982.

15. Bürger, R.C., "MLD: A Language and Data Base for Modeling," IBM Research Report RC 9639 (#42311), 9 September 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Professor Gordon H. Bradley, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
4.	Professor Gerald G. Brown, Code 55Bw Department of Operations Research Naval Postgraduate School Monterey, California 93943	15
5.	Captain Robert D. Clemence, Jr., Code 55 Department of Operations Research Naval Postgraduate School Monterey, California 93943	5
6.	Professor Arthur M. Geoffrion Graduate School of Management University of California, Los Angeles Los Angeles, California 90024	1

END

FILMED

3-85

DTIC