WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

XML MODELING LANGUAGE FOR LINEAR PROGRAMMING: SPECIFICATION AND EXAMPLES

1006-78

May 1978

Robert Fourer

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

Working Paper

Alfred P. Sloan School of Management

Center for Computational Research
in Economics and Management Science

XML MODELING LANGUAGE FOR LINEAR PROGRAMMING: SPECIFICATION AND EXAMPLES

1006-78

May 1978

Robert Fourer

Massachusetts Institute of Technology
50 Memorial Drive
Cambridge, Massachusetts 02139

Research for this report was supported by grant MCS76-01311 to the Center from the National Science Foundation.

Address inquiries to the author as follows:

Robert Fourer
Department of Operations Research
Stanford University
Stanford, California 94305

ABSTRACT

XML is a language for describing linear-programming models to computer systems. Parts I and II of this report together comprise a full syntactic and semantic specification of XML. Several extended examples of XML models are given in Part III.

XML's purpose and structure are also set forth in general terms in "A Modern Approach to Computer Systems for Linear Programming" by Fourer and Harrison (MIT Sloan School Working Paper 988-78).

CONTENTS

INTRODUCTION

PART I: GENERAL XML LANGUAGE FORMS

PART II: XML LANGUAGE FORMS FOR PARTICULAR COMPONENTS AND ELEMENTS

PART III: EXAMPLES OF XML MODELS

INTRODUCTION

This report is a specification of XML, a language for describing linear-programming models to computer systems. Parts I and II together comprise a full syntactic and semantic specification of an initial version of XML. Several extended examples of XML models are given in Part III.

This arrangement is intended to serve two purposes. First, it should make clear in detail what an LP modeling language may be like, thereby making a case that such a language is practical and desirable. Second, it should be sufficiently precise to serve as a basis for a first implementation of XML. These goals are sometimes in conflict -- one cannot always be both clear and precise -- and so examples have been added to Part I where the specifications are especially complicated.

This report is not intended to give a formal grammar for parsing XML. There are many ways in which such a grammar might be devised, but the choice is properly a matter of implementation rather than specification.

On the other hand, this report also is not organized to serve as a user's manual or primer for XML. Readers unfamiliar with the idea of a modeling language are urged to look first at "A Modern Approach to Computer Systems for Linear Programming" [2] which offers a general justification and summary of XML.

Syntactic conventions

XML syntactic forms are written in *italics* throughout this report.

XML employs the full ASCII character set. No distinction is made, however, between the lower-case and upper-case forms of a letter; they may be used interchangeably in any XML expression.

The lexical tokens of XML are *reals* (defined in section §2.1), *strings* (§3.1), *names* (§5.1), and the following special characters:

+ - * / < = > ~ ( ) [ ] { } ' " , :

Appearance of a space or special character indicates the beginning of a new token.

The following notation is used in defining syntactic forms:

→ The syntactic form to the left of the arrow is defined to represent any of the syntactic expressions listed after the arrow.

[] In syntactic expressions, anything within brackets is optional (except where, in §5.2, the brackets are part of the XML language). Section numbers in brackets (for example, [§3.4]) refer to syntactic definitions in other sections. Bracketed numbers to the right of syntactic expressions are line numbers referred to in the ensuing discussion.

... The preceding syntactic form may be repeated any number of times.

Different appearances of the same syntactic form are sometimes distinguished by numbers following the form's name (for example, *argument*1 and *argument*2). Numbered forms are referred to collectively by writing i for the number (*argumenti*).

PART I

GENERAL XML LANGUAGE FORMS

1 STRUCTURE OF A MODEL

§1.1 Model components

An XML *model* is a representation of a class of linear programming problems.

Every model is composed of units called *components*. There are five types of components, each describing a different aspect of the model:

Set components describe collections of objects, over which parts of the model are indexed.

Parameter components describe numerical data required by the model.

Variable components describe the model's structural variables.

Constraint components describe equations and inequalities that restrict the activities of the variables.

Objective components describe functions of the variables to be computed or optimized.

§1.2 Declarations of components

A model is represented by a collection of *declarations* that describe its components.

A declaration may describe just one component separately from all others declared in the model. Such a component is said to be *single*.

Alternatively, a declaration may describe a *group* having any number of related components, all of the same type. Every group is indexed by a specified set value (§3.2): there is exactly one component of the group corresponding to each set member.

A declaration's *name* is a unique identifier used throughout the model to refer to the component or components that the declaration represents. Its *alias* is an alternative to the name provided for use in printed output that refers to the model.

The type of a declaration is the type of component that it declares.

§1.3 Elements of a declaration

Each declaration comprises one or more parts called elements, which are written by use of the XML syntactic forms described in succeeding sections.

There are six types of elements, each pertaining to a different aspect of the component or components being declared:

> A name element gives the declaration's name (§1.2).
>
> An attribute element specifies simple and fundamental properties of a component or group of components.
>
> An indexing element specifies a set value by which a group of components is indexed.
>
> A specification element gives an explicit or symbolic expression for a component or group of components.
>
> An alias element gives the declaration's alias (§1.2).
>
> A comment element is a string of explanatory text that accompanies the declaration.

A declaration contains at most one element of each type.

Each of the five declaration types uses these elements in a somewhat different way. Thus, the precise syntactic form and meaning of an element depend to some extent on the type of declaration in which it is employed. Further, not all types of elements need appear in a declaration. A name element is required, and a specification element is required in constraint and objective declarations; but otherwise all elements are optional. Omission of an optional element is interpreted according to a default convention for that element.

The syntax, meaning, and default for each element type are given separately for each component type in Part II of this specification.

2 REPRESENTATION OF NUMERICAL VALUES

§2.1 Numerical constants

Numerical constants of the forms *integer* and *real* are employed to represent literal numerical values.

An *integer* is any sequence of digits, optionally preceded by a sign (+ or -). *Integer*s represent integral numerical values in *arithmetic-expressions* (§§2.2-2.4), and stand for numerical values contained in sets (§3.2).

A *real* is any sequence of digits, optionally: (a) preceded by a sign (+ or -); (b) including a decimal point before, among, or after the digits; or (c) followed by the letter E and an *integer* exponent. *Real*s represent rational approximations to real numerical values. Every *integer* is also a *real*.

Two rational approximations r_1 and r_2 to real numerical values are equal when their difference is within a sufficiently small tolerance of zero. If r_1 and r_2 are not equal, r_1 is greater than or less than r_2 in the usual sense. (XML incorporates no definition of "sufficiently small". Choice of a tolerance is left to the solution algorithms.)

A (rational approximation to a) real numerical value is, in certain contexts, rounded to yield the nearest integral numerical value. Formally, a positive real value r is rounded to the greatest integer value i such that $i \leq r + 1/2$; $-r$ is rounded to $-i$.

§2.2 *Numerics*

A *numeric* represents a single (rational approximation to a) real numerical value. *Numerics* are the basic building blocks of *arithmetic-expressions* (§2.4). Their general form is:

---

numeric →

real [§2.1]
parameter-reference [§5.2]
variable-reference [§5.2]
objective-reference [§5.2]
index-name [§6.2]

A *real* may serve as a *numeric* anywhere in a model.

A *parameter-reference* may serve as a *numeric* anywhere in a model, subject to certain exceptions to prevent circular definitions (§5.3). The numerical value that a *parameter-reference* represents may be determined from the parameter's declaration (§B.4).

A *variable-reference* may serve as a *numeric* only in a specification element of a constraint or objective declaration. It represents the activity of the referenced variable.

An *objective-reference* may serve as a *numeric* only in a specification element of an objective declaration. It represents the value computed for the referenced objective.

An *index-name* may serve as a *numeric* only within its scope of definition (§6.7). It represents integral numerical values chosen from a specified index set, according to the rules given in §§6.4-6.6. An *index-name* serving as a *numeric* is invalid if these rules assign it a character-string value (§3.1) rather than an integral value.

§2.3 *Arithmetic-functions*

An *arithmetic-function* represents a numerical value computed from one or more other values. Its general form is:

---

arithmetic-function →

function-name(*argument* [, *argument*] ...)

function-name → ABS
CEIL
FLOOR
MAX
MIN
ROUND
TRUNC

argument → *arithmetic-expression* [§2.4]

Each *function-name* imposes certain additional requirements upon the number of *arguments*, and indicates a particular method of computing a value. Particulars are given in §§2.3.1-2.3.3 below.

§2.3.1 Absolute value: ABS

ABS(*arithmetic-expression*)

arithmetic-expression → [§2.4]

The computed value is the magnitude (absolute value) of the value represented by the *arithmetic-expression*.

§2.3.2 Integer functions: CEIL, FLOOR, ROUND, and TRUNC

CEIL(*arithmetic-expression*)
FLOOR(*arithmetic-expression*)
ROUND(*arithmetic-expression*)
TRUNC(*arithmetic-expression*)

arithmetic-expression → [§2.4]

The *arithmetic-expression* is evaluated to yield a numerical value *r*, from which the *arithmetic-functions* compute the following:

CEIL: the smallest integer not less than *r*.

FLOOR: the largest integer not greater than *r*.

ROUND: the integer that results from rounding *r* (§2.1).

TRUNC: the integer part of *r*.

§2.3.3 Greatest or least: MAX and MIN

MAX(*argument*, *argument* [, *argument*] ...)
MIN(*argument*, *argument* [, *argument*] ...)

argument → *arithmetic-expression* [§2.4]

MAX computes the greatest amont the values represented by the *arguments*.

MIN computes the least among the values represented by the *arguments*.

§2.4 *Arithmetic-expressions*

An *arithmetic-expression* is the most general form for representation of (rational approximations to) real numerical values. It is written:

---

arithmetic-expression →
term [1]
arithmetic-expression + *term* [2]
arithmetic-expression - *term* [3]

term → *factor* [A1]
term * *factor* [A2]
term / *factor* [A3]
term DIV *factor* [A4]
term MOD *factor* [A5]

factor → *atom* [B1]
+ *factor* [B2]
- *factor* [B3]
atom ** *factor* [B4]

atom → *numeric* [§2.2] [C1]
arithmetic-function [§2.3] [C2]
(*arithmetic-expression*) [C3]
sigma [C4]

sigma → SIGMA *indexing-expression* (*arithmetic-expression*)

indexing-expression → [§6.3]

An *arithmetic-expression* represents the numerical value determined by the following recursive algorithm:

The value of an *arithmetic-expression* is the value of a *term* [1], the sum of the values of an *arithmetic-expression* and of a *term* [2], or the value of an *arithmetic-expression* minus the value of a *term* [3].

The value of a *term* is the value of a *factor* [A1], the product of the values of a *term* and a *factor* [A2], the value of a *term* divided by the value of a *factor* [A3], the rounded value of a *term* integer-divided by the rounded value of a *factor* [A4], or the rounded value of a *term* modulo the rounded value of a *factor* [A5].

The value of a *factor* is the value of an *atom* [B1] or of another *factor* [B2], the negative of the value of a *factor* [B3], or the value of an *atom* raised to the power of the value of a *factor* [B4].

The value of an *atom* is the value represented by a *numeric* [C1] or by an *arithmetic-function* [C2]; or is the value of the parenthesized *arithmetic-expression* [C3]; or is the value of a *sigma* [C4].

The value of a *sigma* is found as follows: the parenthesized *arithmetic-expression* is evaluated once with respect to each index determined by the *indexing-expression* (§§6.4-6.6); and all resulting values are summed. If the *indexing-expression* specifies an empty index set, the value of the *sigma* is zero.

(Examples of *arithmetic-expressions* of many kinds appear in the sample models in Part III of this specification.)

§2.5 *Constant-arithmetic-expressions*

A *constant-arithmetic-expression* is any *arithmetic-expression* (§2.4) containing no *variable-references* or *objective-references* (§5.2).

The value of a *constant-arithmetic-expression* does not depend on the variables' activities, and so is unchanged from solution to solution. In this sense, it is a constant of the model.

§2.6 *Linear-arithmetic-expressions*

A *linear-arithmetic-expression* is any *arithmetic-expression* (§2.4) that satisfies the following restrictions:

- It contains no *objective-references* (§5.2).
- In every *term*, at most one *factor* contains *variable-references* (§5.2).
- In every *term* of the form *term* / *factor*, the *factor* contains no *variable-references*.
- No *term* of the form *term* DIV *factor* or *term* MOD *factor* contains *variable-references*.
- No *factor* of the form *atom* ** *factor* contains *variable-references*.
- No *arithmetic-function* contains *variable-references*.

Every *linear-arithmetic-expression* is either a *constant-arithmetic-expression* (§2.5), or represents a linear combination of the activities of one or more variables.

3 REPRESENTATION OF SET VALUES

§3.1 *Strings*

A *string* is any sequence of characters beginning and ending with an apostrophe (') and containing no apostrophes elsewhere, or beginning and ending with a double-quote (") and containing no double-quotes elsewhere.

Strings represent character-string values: arbitrary sequences of characters. The character-string value represented by a particular *string* is exactly the sequence of characters between the apostrophes or double-quotes.

An empty sequence of characters (that is, a sequence of zero characters) is called the null character-string value. It is represented by a *string* comprising two consecutive apostrophes or double-quotes ('' or "").

§3.2 Items

An item is either an integral numerical value or a character-string value. Items represent the "things" that a model is concerned with (factories, products, cities, periods, and so forth); they are the fundamental constituents of sets.

Two items are equal if they are identical integer values or identical character-string values. (An integer item is never equal to a string item.)

Items are represented by *item-expressions* of the form:

---

item-expression →

string [§3.1]
integer [§2.1]
constant-arithmetic-expression [§2.5]
index-name [§6.2]

A *string* (representing a character-string value) or an *integer* (representing an integral numerical value) may serve as an *item-expression* anywhere in a model.

A *constant-arithmetic-expression* may serve as an *item-expression* anywhere in a model, subject to restrictions on the *numerics* within it (§2.2). It represents the integer item produced by rounding its arithmetic value.

An *index-name* may serve as an *item-expression* only within its scope of definition (§6.7). It represents items chosen from a specified index set, according to the rules given in §§6.4-6.6.

§3.3 Objects

An object comprises a single item or an ordered sequence of two or more items. The number of items in an object is its length.

Two objects are equal if and only if they have the same length and comprise the same items in the same order.

Thus, an object of length 1 is essentially just an item. An object of length 2 represents an "ordered pair" of items, an object of length 3 represents an "ordered triple" of items, and so forth. In general, an object of length *n* represents an "ordered list" of *n* items.

Objects are represented by use of the form *object-expression*:

---

object-expression →

item-expression [1]
(*item-expression*, *item-expression* [, *item-expression*] ...) [2]

item-expression → [§3.2]

Form [1] represents an object comprising a single item. Form [2] represents the ordered sequence of items in an object of length 2 or more.

§3.4 Set values

A set value is an unordered collection of any number of distinct objects. These objects are said to be contained in the set value, and are referred to as its members. A set value having no members is empty.

All members of a set value must have the same length, referred to as its member-length. Thus one may have sets of single items (member-length of 1), sets of ordered pairs of items (member-length of 2), and so forth. The member-length of an empty set is undefined.

Two set values are equal if every member of the first is equal to some member of the second, and every member of the second is equal to some member of the first.

Set values may be represented by use of the form *set-constant*:

---

set-constant →

{}

{*member* [, *member*] ...}

member → *object-expression* [: *alias*]

object-expression → [§ 3.3]

alias → *string* [§ 3.1]

Each *object-expression* between the braces represents one member of the set value denoted by the *set-constant*. Braces with nothing between them denote an empty set.

The optional *alias* specifies a member's alternative name for use in reports. For example, a set of cities could be written

{'BO':'BOSTON', 'NY':'NEW YORK', 'PH':'PHILADELPHIA'}

Aliases are not part of the set value; they are ignored in interpreting *set-functions* (§3.5) and *set-expressions* (§3.6).

§3.5 *Set-functions*

A *set-function* represents a set value computed from one or more other set values, items, or numerical values. Its general form is:

---

set-function →

function-name(*argument* [, *argument*] ...)

function-name → PROJ
SECT
SEQ

argument → *item-expression* [§3.2]
set-expression [§3.6]
constant-arithmetic-expression [§2.5]

Each *function-name* imposes additional requirements upon the number and form of *arguments*, and indicates a particular method of computing a set value. Particulars are given in §§3.5.1-3.5.3 below.

§3.5.1 Projection of a set: PROJ

PROJ(*set-expression* [, *constant-arithmetic-expression*] ...)

set-expression → [§3.6]

constant-arithmetic-expression → [§2.5]

PROJ projects a set value (represented by the *set-expression*) onto specified coordinates (either as indicated by the *constant-arithmetic-expressions*, or else the first coordinate by default). The projection is determined as follows:

If there are *constant-arithmetic-expressions*, they must represent distinct positive integral numerical values; denote these values by $i_1, \ldots, i_n$. Otherwise, let $n = 1$ and $i_1 = i_n = 1$.

Denote by S the set value represented by the *set-expression*; denote by ℓ the member-length of S. Then it is required that $\ell \geq n$, and $\ell \geq i_1, \ldots, \ell \geq i_n$.

The computed (projection) set value has member-length n: for each object $(a_1, \ldots, a_\ell)$ in S, the object $(a_{i_1}, \ldots, a_{i_n})$ is in the computed projection.

Example: Suppose ROUTES represents the set

{('BO','NY',0), ('BO','PH',0), ('BO','PH',1),
('BO','WA',0), ('BO','WA',1), ('NY','WA',0)}

The following are some possible uses of PROJ:

PROJ(ROUTES) is {'BO','NY'}
PROJ(ROUTES,2) is {'NY','PH','WA'}
PROJ(ROUTES,1,2) is {('BO','NY'),('BO','PH'),('BO','WA'),('NY','WA')}
PROJ(ROUTES,3,1) is {(0,'BO'),(1,'BO'),(0,'NY')}

§3.5.2 Section of a set: SECT

SECT(*set-expression*,*item-expression*)
SECT(*set-expression* [,*item-expression*,*constant-arithmetic-expression*] ...)

set-expression → [§3.6]

item-expression → [§3.2]

constant-arithmetic-expression → [§2.5]

SECT sections, or "slices", a set value (represented by the *set-expression*) on specified items (represented by the *item-expressions*) at specified coordinates (as indicated by the *constant-arithmetic-expressions*, or else the first coordinate if none is indicated). The section is determined as follows:

Denote by $e_1, \ldots, e_n$ the items represented by the *item-expressions*. If there are *constant-arithmetic-expressions*, they must represent distinct positive integral numerical values; denote these values by $i_1, \ldots, i_n$. Otherwise, $n = 1$; set $i_1 = 1$.

Denote by S the set value represented by the *set-expression*; denote by ℓ the member-length of S. It is required that $\ell > n$, and $\ell \geq i_1, \ldots, \ell \geq i_n$.

The computed (section) set value has member-length $\ell - n$. For each object $(a_1, \ldots, a_\ell)$ in S for which

$$a_{i_1} = e_1, \ldots, a_{i_n} = e_n$$

the computed section contains the same object with items $a_{i_1}, \ldots, a_{i_n}$ deleted.

Example: Consider again the set ROUTES of §3.5.1. Some possible uses of SECT are:

SECT(ROUTES,'BO')	is	{('NY',0),('PH',0),('PH',1), ('WA',0),('WA',1)}
SECT(ROUTES,'NY')	is	{('WA',0)}
SECT(ROUTES,'PH')	is	{ }
SECT(ROUTES,'WA',2)	is	{('BO',0),('BO',1),('NY',1)}
SECT(ROUTES,1,3)	is	{('BO','PH'),('BO','WA')}
SECT(ROUTES,1,3,'BO',1)	is	{'PH','WA'}

§3.5.3 Set of a sequence of integral values: SEQ

SEQ(*argument1*, *argument2* [, *argument3*])

argumenti → *constant-arithmetic-expression* [§2.5]

Argument1 and *argument2* are evaluated and rounded; denote the resulting integral values by m and n, respectively. If *argument3* is present, it is evaluated and rounded; denote the resulting integral value by k. If *argument*3 is not present, let k be 1.

The computed set value contains all integers of the form $m + ik$ for which i is a nonnegative integer and $m + ik \leq n$.

Example: Some instances of SEQ are:

SEQ(1,10)	is	{1,2,3,4,5,6,7,8,9,10}
SEQ(1975,2000,5)	is	{1975,1980,1985,1990,1995,2000}

§3.6 *Set-expressions*

A *set-expression* is the most general form for representation of set values. It is written:

---

set-expression →

set-difference [1]
set-expression * *set-difference* [2]

set-difference → *set-union* [A1]
set-difference - *set-union* [A2]

set-union → *set-intersection* [B1]
set-union OR *set-intersection* [B2]

set-intersection → *set-atom* [C1]
set-intersection AND *set-atom* [C2]

set-atom → *set-constant* [§3.4] [D1]
set-function [§3.5] [D2]
set-reference [§5.2] [D3]
(*set-expression*) [D4]

A *set-expression* represents the set value determined by the following recursive algorithm:

The value of a *set-expression* is the value of a *set-difference* [1], or is the cartesian product of the values of a *set-expression* and a *set-difference* [2] determined as follows: Let S_1 and S_2 denote the values of the *set-expression* and *set-difference*; let ℓ_1 and ℓ_2 denote their respective member-lengths. Then the cartesian product has member-length $\ell_1 + \ell_2$; for every pair of members $(a_1, \ldots, a_{\ell_1})$ of S_1 and $(b_1, \ldots, b_{\ell_2})$ of S_2, the cartesian product contains $(a_1, \ldots, a_{\ell_1}, b_1, \ldots, b_{\ell_2})$.

The value of a *set-difference* is the value of a *set-union* [A1], or is the complement of a *set-union* in a *set-difference* [A2] determined as follows: Let S_1 and S_2 denote the values of the *set-difference* and *set-union*, respectively; they must have the same member-length. Then the complement contains all members of S_1 that are not members of S_2.

The value of a *set-union* is the value of a *set-intersection* [B1], or is the union of a *set-union* and a *set-intersection* [B2] determined as follows: Let S_1 and S_2 denote the values of the *set-union* and *set-intersection*; they must have the same member-length. Then their union contains an object if and only if it is a member of S_1 <u>or</u> a member of S_2.

The value of a *set-intersection* is the value of a *set-atom* [C1], or is the intersection of the values of a *set-intersection* and a *set-atom* [C2] determined as follows: Let S_1 and S_2 denote the values of the *set-intersection* and *set-atom*; they must have the same member-length. Then their intersection contains an object if and only if it is a member of S_1 <u>and</u> a member of S_2.

The value of a *set-atom* is the value represented by a *set-constant* [D1], *set-function* [D2], or *set-reference* [D3], or is the value of the parenthesized *set-expression* [D4].

<u>Examples</u>. Represent some sets as follows:

S1	{1,2,3}
S2	{2,3,4,5}
S3	{'A','B'}
S4	{'C','D'}

Some typical set expressions and their values are:

S3 * S4	{('A','C'),('A','D'),('B','C'),('B','D')}
S1 * S1	{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)}
S2 - S1	{4,5}
S1 - S2	{1}
S4 - S3	{}
S1 OR S2	{1,2,3,4,5}
S1 AND S2	{2,3}
S3 AND S4	{}

4 REPRESENTATION OF LOGICAL VALUES

§4.1 Logical values

There are two *logical values*: *true* and *false*. Forms for representing logical values are given in the following sections.

A syntactic form is said to *be true* when it represents the logical value true, and to *be false* when it represents the value false.

§4.2 *Equalities*

An *equality* is true or false according to whether two values are equal or unequal. Its form is:

---

equality →

*value*1 = *value*2 [1]
*value*1 ~= *value*2 [2]

*value*1, *value*2 → *constant-arithmetic-expression* [§2.5]
item-expression [§3.2]
object-expression [§3.3]
set-expression [§3.6]

Form [1] is true and form [2] is false if and only if the two indicated values are equal. The sort of equality to be tested is determined by *value*1 and *value*2 as follows:

If *value*1 and *value*2 are *constant-arithmetic-expressions*, the *equality* tests whether they represent equal numerical values [§2.1].

Otherwise, if *value*1 and *value*2 are *item-expressions*, the *equality* tests whether they represent equal items [§3.2]; if *value*1 and *value*2 are *object-expressions*, the *equality* tests whether they represent equal objects [§3.3].

If *value*1 and *value*2 are *set-expressions*, the *equality* tests whether they represent equal set values [§3.4].

§4.3 *Inequalities*

An *inequality* is true or false according to whether one numerical value is greater than or less than another. Its form is:

inequality →

value1 > *value2* [1]
value1 <= *value2* [2]
value1 < *value2* [3]
value1 >= *value2* [4]

value1, *value2* → *constant-arithmetic-expression* [§2.5]

Form [1] is true and form [2] is false if and only if the numerical value represented by *value1* is greater than that represented by *value2*.

Form [3] is true and form [4] is false if and only if the numerical value represented by *value1* is less than that represented by *value2*.

§4.4 *Memberships*.

A *membership* is true or false according to whether an object or objects are members of a given set. Its form is:

membership →

object-expression IN *set-expression* [1]
set-expression IN *set-expression* [2]

object-expression → [§3.3]

set-expression → [§3.6]

Form [1] is true if and only if the object represented by *object-expression* is a member of the set value represented by *set-expression*.

Form [2] is true if and only if every object contained in the set value represented by the left-hand *set-expression* is also contained in the set value represented by the right-hand *set-expression*.

§4.5 *Logical-expressions*

A *logical-expression* is the most general form for representation of logical values. It is written as follows:

logical-expression →
logical-term [1]
logical-expression OR *logical-term* [2]
logical-term → *logical-factor* [A1]
logical-term AND *logical-factor* [A2]
logical-factor → *logical-atom* [B1]
NOT *logical-factor* [B2]
logical-atom → *equality* [§4.2] [C1]
inequality [§4.3] [C2]
membership [§4.4] [C3]
(*logical-expression*) [C4]

The truth or falsity of a *logical-expression* is determined by the following recursive algorithm:

The value of a *logical-expression* is the value of a *logical-term* [1]; or is false if and only if both a *logical-expression* and a *logical-term* are false [2].

The value of a *logical-term* is the value of a *logical-factor* [A1]; or is true if and only if both a *logical-term* and a *logical-factor* are true [A2].

The value of a *logical-factor* is the value of a *logical-atom* [B1]; or is true if and only if a *logical-factor* is false [B2].

The value of a *logical-atom* is the value represented by an *equality* [C1], *inequality* [C2], or *membership* [C3], or is the value of the parenthesized *logical-expression* [C4].

5 REFERENCES TO MODEL COMPONENTS

§5.1 *Names*

A *name* is any sequence of letters, digits (0, 1, 2, ..., 9), and underscores (_) of which the first character is a letter.

Names serve as identifiers for model components, as described below. *Names* also identify indices employed in indexing operations (§§6.1-6.7).

The following reserved words are *names* that XML uses for special purposes, and so are not allowed as *component-names* (§5.2) or *index-names* (§6.2):

AND
BY
DIV
FROM
IN
MOD
NOT
OR
OVER
SIGMA
TO
WHERE

§5.2 *Component-references*

A *component-reference* refers to a particular component (§1.1) of the model, and represents a value associated with that component (§§2.2, 3.6). Its form is:

---

component-reference →

component-name [1]

component-name[*item-expression* [, *item-expression*] ...] [2]

component-name → *name* [§5.1]

item-expression → [§3.2]

Note: underscored brackets [and] are part of the *component-reference*.

The component that is referred to is determined as follows:

Form [1]: The *component-name* must be the name (§1.2) of a declaration of a single (§1.2) component in the model. The *component-reference* refers to this component.

Form [2]: The *component-name* must be the name of a declaration of a group (§1.2) of components in the model. This group must be indexed by a set value that contains either (a) a member represented by

item-expression

if the *component-reference* contains just one *item-expression*; or (b) a member represented by

(*item-expression*1, ..., *item-expression*i)

if the *component-reference* contains $i \geq 2$ *item-expressions*. The *component-reference* refers to the group's component corresponding to this member.

Component-references are designated *set-references*, *parameter-references*, *variable-references*, *constraint-references*, or *objective-references* according to the type of component to which they refer.

Examples: (1) A variable declaration has name X and declares a group indexed over {'WA','PH','NY','BO'}. The following *variable-references* represent the variables:

```
X['WA']
X['PH']
X['NY']
X['BO']
```

(2) A parameter declaration has name DEMAND and declares a group indexed over {(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}. The following *parameter-references* represent the parameters:

```
DEMAND[1,2]
DEMAND[1,3]
DEMAND[1,4]
DEMAND[2,3]
DEMAND[2,4]
DEMAND[3,4]
```

§5.3 Circular declarations

Some restriction must be placed upon *component-references* to avoid declarations that define components in terms of themselves.

For this purpose, declaration A is said to depend upon declaration B if either

(a) A contains a *component-reference* for a component declared by B; or

(b) A depends on some declaration C, and C depands on B.

A declaration is circular if it depends on itself.

A *component-reference* is invalid if its presence results in a circular declaration.

6 INDEXING

§6.1 Definition

Indexing is the means by which model entities are associated with members of sets. XML uses indexing in two ways: to define groups of components (§1.2) and to specify indexed (Σ) sums (§2.4).

There are two aspects to indicating indexing in XML:

(a) Specifying a set value called the index set. Members of this set are called indices, and the items in each member are a list of index items.

(b) Evaluating *set-expressions*, *arithmetic-expressions*, and *logical-expressions* with respect to each of the indices. For this purpose, *index-names* may be defined to represent index items.

The syntactic forms for indicating indexing are given in §6.2 and §6.3. Rules for determining the index set are put forth in §6.4, and evaluation of expressions with respect to indices is described in §§6.5-6.6. Limits on appearance of an *index-name* within a model are explained in §6.7.

§6.2 *Indexing-units*

Indexing-units express the simplest concepts of indexing. They are written in either of two forms:

indexing-unit →

[*index-list*] OVER *over-exp* [WHERE *where-exp*] [1]
[*index-name*] FROM *from-exp* TO *to-exp* [BY *by-exp*] [WHERE *where-exp*] [2]

index-list → *index-name*
(*index-name*, *index-name* [, *index-name*] ...)

index-name → *name* [§5.1]

over-exp → *set-expression* [§3.6]

$$\left.\begin{array}{l}\textit{from-exp}\\ \textit{to-exp}\\ \textit{by-exp}\end{array}\right\} \rightarrow \textit{constant-arithmetic-expression}\quad [\S 2.5]$$

where-exp → *logical-expression* [§4.5]

Each *index-name* in the *index-list* (in form [1]), or the initial *index-name* (in form [2]), is said to be defined in the *indexing-unit*.

An *index-name* defined in an *indexing-unit* must be different from:

(a) every other *index-name* defined in the *indexing-unit*;

(b) every other *index-name* in whose scope (§6.7) the *indexing-unit* lies; and

(c) every *name* specified in a name element of any declaration.

The number of *index-names* in the *index-list* (for form [1]) must equal the member-length (§3.4) of the set value represented by the *over-exp*.

§6.3 *Indexing-expressions*

Indexing-expressions are the general forms for indicating indexing. They are written as lists of *indexing-units*:

---

indexing-expression →

indexing-unit [A1]
initial-indexing-expression, *terminal-indexing-unit* [A2]

initial-indexing-expression → *indexing-expression*
terminal-indexing-unit → *indexing-unit*
indexing-unit → [§6.2]

Form [A1] consists of a single *indexing-unit*, while form [A2] comprises a sequence of *indexing-units*.

§6.4 Determining the index set

Every *indexing-expression* specifies an index set (§6.1). The manner in which this set is determined depends on the form of the *indexing-expression*, as follows:

Single *indexing-unit*, form [1], no *where-exp*: The index set is the set value represented by the *over-exp*.

Single *indexing-unit*, form [2], no *where-exp*: If the *by-exp* is not present, the index set is the set value represented by

SEQ(*from-exp*, *to-exp*)

If the *by-exp* is present, the index set is the set value represented by

SEQ(*from-exp*, *to-exp*, *by-exp*)

(On SEQ, see §3.5.3.)

Single *indexing-unit*, with *where-exp*: The index set is the set value determined by the following algorithm:

(a) Determine a tentative index set by ignoring the *where-exp* and following the rules for form [1] or [2] given above.

(b) Place an index in the index set if and only if (i) it is a member of the tentative index set, and (ii) the *where-exp* is true with respect to it.

(On evaluation with respect to an index, see §6.6.)

Sequence of *indexing-units*: The index set is the set value determined by the following algorithm:

(a) Determine an initial index set from the *initial-indexing-expression*, by applying the rules of this section (§6.4) recursively. Denote the members of the initial index set by $m_1, \ldots, m_n$.

(b) For each m_i determine a terminal index set T_i from the *terminal-indexing-unit*, by (i) interpreting any *over-exp*, *from-exp*, *to-exp*, *by-exp*, or *where-exp* with respect to m_i (§6.6), and (ii) following the rules for single-unit *indexing-expressions* given above.

(c) Compute the index set as:
$(\{m_1\} * T_1)$ OR $(\{m_2\} * T_2)$ OR ... OR $(\{m_n\} * T_n)$

(On operators * and OR, see §3.6.)

Examples: (1) Suppose that CITY represents the set {'PH','NY','BO'}. Then the index set specified by

OVER CITY, FROM 1972 TO 1984 BY 4

is just CITY * SEQ(1972,1984,4), whose members (indices) are

('PH',1972)	('NY',1972)	('BO',1972)
('PH',1976)	('NY',1976)	('BO',1976)
('PH',1980)	('NY',1980)	('BO',1980)
('PH',1984)	('NY',1984)	('BO',1984)

(2) Consider the following *indexing-expression*:

```
I FROM 1 TO 5, J FROM 1 TO I
```

The initial index set is {1,2,3,4,5}; the terminal index sets are $T_i = \{1, \ldots, i\}$. Thus the entire index set is

```
({1}*{1}) OR ({2}*{1,2}) OR ({3}*{1,2,3})
                    OR ({4}*{1,2,3,4}) OR ({5}*{1,2,3,4,5})
```

which is just the "triangular" set of pairs

```
{(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),
            (4,4),(5,1),(5,2),(5,3),(5,4),(5,5)}
```

(3) Either of these *indexing-expressions*:

```
C1 OVER CITY, C2 OVER CITY WHERE C1 ~= C2
(C1,C2) OVER CITY * CITY WHERE C1 ~= C2
```

specifies an index set that contains all members of CITY * CITY whose two items are not the same. Hence the index set is:

```
{('PH','NY'),('PH','BO'),('NY','PH'),
                    ('NY','BO'),('BO','PH'),('BO','NY')}
```

§6.5 Representing index items by *index-names*

An *index-name* defined in an *indexing-expression* represents the *k*th item of every index, for some *k*. The value of *k* is determined by where the *index-name* appears in the *indexing-expression*, in the following way:

Single *indexing-unit*, form [1]: The *k*th *index-name* in the *index-list* represents the *k*th item in each index.

Single *indexing-unit*, form [2]: The one *index-name* represents the first (and only) item in each index.

Sequence of *indexing-units*: Let ℓ be the length of the indices determined by the *initial-indexing-expression*. Then an *index-name* is determined to represent the *k*th item in each index as follows:

(a) For an *index-name* defined in the *initial-indexing-expression*, determine k by applying the rules of this section (§6.5) recursively to the *initial-indexing-expression*.

(b) For an *index-name* defined in the *terminal-indexing-unit*, determine a k' by applying the above rules for form [1] or [2] to the *terminal-indexing-unit*. Then $k = \ell + k'$.

Examples: Let ROUTE represent a set value whose member-length is 2, and let PRODUCT and USE represent sets whose member-lengths are 1.

(1) The following *indexing-expression* specifies indices whose length is 4:

```
I OVER PRODUCT, (J1,J2) OVER ROUTE, K OVER USE
```

Index-name I represents the first item in each index. J1 represents the second item, J2 the third, and K the fourth.

(2) Indices specified by the following *indexing-expression* also have length 4:

```
I OVER PRODUCT, FROM 1968 TO 1992 BY 4, (J,K) OVER ROUTE
```

Index-name I represents the first item in each index, J represents the third, and K represents the fourth. No *index-name* represents the second item in each index.

§6.6 Evaluation with respect to an index

An *arithmetic-expression*, *set-expression*, or *logical-expression* is evaluated with respect to an index as follows:

(a) Each *index-name* in the expression is taken to represent some item of the index, as explained in §6.5.

(b) The value of the expression is computed according to the usual rules.

Arithmetic-expressions are evaluated with respect to indices in *sigmas* (§2.4); in *from-exps*, *to-exps*, and *by-exps* of *indexing-expressions* (§§6.2-6.4); and in specification elements for parameter, variable, constraint, and objective components (§§B.4,C.4,D.4,E.4).

Set-expressions are evaluated with respect to indices in *over-exps* of *indexing-expressions* (§§6.2-6.4) and in specification elements for set components (§A.4).

Logical-expressions are evaluated with respect to indices in *where-exps* of *indexing-expressions* (§§6.2-6.4) and in specification elements for constraint components (§D.4).

Example: Consider the following *sigma*, in which X names a variable:

```
SIGMA I OVER {'PH','NY','BO'},
         J FROM 1972 TO 1984 BY 4 ( (J-1970) * X[I,J+4] )
```

The *indexing-expression* in this *sigma* specifies a set of 12 indices of length 2 (see §6.4, example 1); *index-name* I represents the first item of each index, and J represents the second item of each index.

Thus evaluating the expression (J-1970) * X[I,J+4] with respect to, say, the index ('NY',1976) yeilds the value of 6 * X['NY',1980]. Evaluating the same expression with respect to ('BO',1984) yields 14 * X['BO',1988].

§6.7 Scopes of *index-names*

A scope of an *index-name* is that portion of the model in which the *index-name* may be used to represent values of index items (as described in §§6.5-6.6).

An *index-name* has one scope for each *indexing-unit* that defines it. Scopes of the same *index-name* for different *indexing-units* may not overlap (§6.2).

A scope of an *index-name* for a particular *indexing-unit* includes:

(a) the *where-exp* of the *indexing-unit* (if any, §6.2);

(b) all succeeding *indexing-units* in the same *indexing-expression* (if any);

(c) if the *indexing-unit* is part of a declaration's indexing element (§§A.3,B.3,C.3,D.3,E.3): any specification element (§§A.4,B.4,C.4,D.4,E.4) in the same declaration;

(d) if the *indexing-unit* is part of a *sigma* (§2.4): the parenthesized *arithmetic-expression* that follows that *sigma*'s *indexing-expression*.

The remainder of the model is not included in the scope.

PART II

XML LANGUAGE FORMS FOR PARTICULAR COMPONENTS AND ELEMENTS

A SET DECLARATIONS

§A.1 Name element (required)

---

set-name-element →

name [§5.1]

This element specifies the name (§1.2) for the declaration.

No *set-name-element* may be the same as any other *set-name-element*, *parameter-name-element* (§B.1), *variable-name-element* (§C.1), *constraint-name-element* (§D.1), *objective-name-element* (§E.1), or *index-name* (§6.2) defined in the model.

§A.2 Attribute element (optional)

---

set-attribute-element →

[LENGTH] *n*

n → *integer* [§2.1]

This element states that the declared set or sets must represent a set value whose member-length (§3.4) is *n*.

The constant *n* must be a positive *integer*.

Default: LENGTH 1 is assumed.

§A.3 Indexing element (optional)

set-indexing-element →

    indexing-expression   [§6.3]

A group (§1.2) of sets is declared if and only if this element is present. The group contains one set corresponding to each index (§6.1) specified by the *indexing-expression*.

Default: A single (§1.2) set component is declared.

§A.4 Specification element (optional)

set-specification-element →

    set-expression   [§3.6]

This element indicates what set value each declared set represents, as follows:

If the declaration is for a single set: any reference to that set represents the set value yielded by evaluating the *set-expression*.

If the declaration is for a group of sets: any reference to a set in the group represents the value yielded by evaluating the *set-expression* with respect to that set's corresponding index (§§6.6, A.3).

Default. The model does not indicate what set value each declared set is to represent.

§A.5 Alias element (optional)

---

set-alias-element →

string [§3.1]

The sequence of characters represented by the *string* is the alias (§1.2) for the declaration.

Default: No alias is defined.

§A.6 Comment element (optional)

---

set-comment-element →

string [§3.1]

The sequence of characters represented by the *string* is a comment to accompany the declaration.

Default: No comment is defined.

B PARAMETER DECLARATIONS

§B.1 Name element (required)

---

parameter-name-element →

name [§5.1]

This element specifies the name (§1.2) for the declaration.

No *parameter-name-element* may be the same as any other *parameter-name-element*, *set-name-element* (§A.1), *variable-name-element* (§C.1), *constraint-name-element* (§D.1), *objective-name-element* (§E.1), or *index-name* (§6.2) defined in the model.

§B.2 Attribute element (optional)

---

parameter-attribute-element →

restriction [*restriction*]

restriction → POSITIVE
NEGATIVE
NONPOSITIVE
NONNEGATIVE
NONZERO
INTEGER

This element states that numerical values represented by the declared parameter(s) must satisfy the listed *restrictions*.

Default: The declared parameter(s) may represent any numerical values.

§B.3 Indexing element (optional)

---

parameter-indexing-element →

indexing-expression [§6.3]

A group (§1.2) of parameters is declared if and only if this element is present. The group contains one parameter corresponding to each index (§6.1) specified by the *indexing-expression*.

Default: A single (§1.2) parameter component is declared.

§B.4 Specification element (optional)

---

parameter-specification-element →

constant-arithmetic-expression [§2.5]

This element indicates what numerical value each declared parameter represents, as follows:

If the declaration is for a single parameter: any reference to that parameter represents the numerical value yielded by evaluating the *constant-arithmetic-expression*.

If the declaration is for a group of parameters: any reference to a parameter in the group represents the value yielded by evaluating the *constant-arithmetic-expression* with respect to that parameter's corresponding index (§§6.6, B.3).

Default: The model does not indicate what numerical value each declared parameter is to represent.

§B.5 Alias element (optional)

---

parameter-alias-element →

string [§3.1]

The sequence of characters represented by the *string* is the alias (§1.2) for the declaration.

Default: No alias is defined.

§B.6 Comment element (optional)

---

parameter-comment-element →

string [§3.1]

The sequence of characters represented by the *string* is a comment to accompany the declaration.

Default: No comment is defined.

C VARIABLE DECLARATIONS

§C.1 Name element (required)

---

variable-name-element →

name [§5.1]

This element specifies the name (§1.2) for the declaration.

No *variable-name-element* may be the same as any other *variable-name-element*, *set-name-element* (§A.1), *parameter-name-element* (§B.1), *constraint-name-element* (§D.1), *objective-name-element* (§E.1), or *index-name* (§6.2) defined in the model.

§C.2 Attribute element (optional)

---

variable-attribute-element →

attributei ...

*attribute*1 → NONNEGATIVE
NONPOSITIVE
UNRESTRICTED

*attribute*2 → CONTINUOUS
INTEGER

There must be at most one *attribute*1. The choice of *attribute*1 may impose a restriction on the class of feasible solutions, as follows:

NONNEGATIVE -- A solution is feasible only if all variables defined by the declaration have zero or positive activities.

NONPOSITIVE -- A solution is feasible only if all variables defined by the declaration have zero or negative activities.

UNRESTRICTED -- A solution may be feasible regardless of the signs of the variables defined by the declaration.

There must be at most one *attribute2*. If *attribute2* is INTEGER, a solution is feasible only if all variables defined by the declaration have integral activities. If *attribute2* is CONTINUOUS, a solution may be feasible regardless of the activities' integrality.

Default: If no *attribute1* is specified, UNRESTRICTED is assumed. If no *attribute2* is specified, CONTINUOUS is assumed.

§C.3 Indexing element (optional)

---

variable-indexing-element →

indexing-expression [§6.3]

A group (§1.2) of variables is declared if and only if this element is present. The group contains one variable corresponding to each index (§6.1) specified by the *indexing-expression*.

Default: A single (§1.2) variable component is declared.

§C.4 Specification element (optional)

---

variable-specification-element →

linear-arithmetic-expression [§2.6]

This element indicates that each declared variable represents the activity of a particular linear form (which may be a constant), as follows:

If the declaration is for a single variable: any reference to that variable represents the activity yielded by evaluating the *linear-arithmetic-expression*.

If the declaration is for a group of variables: any reference to a variable in the group represents the activity yielded by evaluating the *linear-arithmetic-expression* with respect to that variable's corresponding index (§§6.6, C.3).

Default: The model does not indicate what activity each declared variable is to represent.

§C.5 Alias element (optional)

---

variable-alias-element →

string [§3.1]

The sequence of characters represented by the *string* is the alias (§1.2) for the declaration.

Default: No alias is defined.

§C.6 Comment element (optional)

---

variable-comment-element →

string [§3.1]

The sequence of characters represented by the *string* is a comment to accompany the declaration.

Default: No comment is defined.

D CONSTRAINT DECLARATIONS

§D.1 Name element (required)

---

constraint-name-element →

name [§5.1]

This element specifies the name (§1.2) for the declaration.

No *constraint-name-element* may be the same as any other *constraint-name-element*, *set-name-element* (§A.1), *parameter-name-element* (§B.1), *variable-name-element* (§C.1), *objective-name-element* (§D.1), or *index-name* (§6.2) defined in the model.

§D.2 Attribute element (optional)

---

constraint-attribute-element →

BOUND
GUB

This element indicates that each declared constraint has one of the following special forms:

BOUND -- an upper bound, lower bound, or both on one particular variable.

GUB -- a "generalized upper bound": upper and lower bounds on an unweighted sum of variables, or on the difference of two such sums.

Default: No special form is indicated.

§D.3 Indexing element (optional)

---

constraint-indexing-element →

indexing-expression [§6.3]

A group (§1.2) of constraints is declared if and only if this element is present. The group contains one constraint corresponding to each index (§6.1) specified by the *indexing-expression*.

Default: A single (§1.2) constraint component is declared.

§D.4 Specification element (required)

---

constraint-specification-element →

lhs = *rhs* [1]
lhs <= *rhs* [2]
lhs >= *rhs* [3]
*rhs*1 <= *lhs* <= *rhs*2 [4]
*rhs*1 >= *lhs* >= *rhs*2 [5]
logical-expression [§4.5] [6]

lhs → *linear-arithmetic-expression* [§2.6]

rhs → *linear-arithmetic-expression* [§2.6]

*rhs*1, *rhs*2 → *constant-arithmetic-expression* [§2.5]

If the declaration is for a single constraint, this element specifies a condition that must be satisfied by a feasible solution.

If the declaration is for a group of constraints, each constraint in the group imposes a condition that must be satisfied by a feasible solution. The condition imposed by any particular constraint is determined by interpreting *lhs*, *rhs* or *rhs*1 and *rhs*2, or the *logical-expression* with respect to the particular constraint's corresponding index (§§6.6, D.3).

The nature of the condition represented by a *constraint-specification-element* depends on its form. Specifically, a solution is feasible only if:

Form [1]: The numerical value represented by *lhs* equals that represented by *rhs*.

Form [2]: The numerical value represented by *lhs* is less than or equal to that represented by *rhs*.

Form [3]: The numerical value represented by *lhs* is greater than or equal to that represented by *rhs*.

Form [4]: The numerical value represented by *rhs*1 is less than or equal to that represented by *lhs*, and the numerical value represented by *lhs* is less than or equal to that represented by *rhs*2.

Form [5]: The numerical value represented by *rhs*1 is greater than or equal to that represented by *lhs*, and the numerical value represented by *lhs* is greater than or equal to that represented by *rhs*2.

Form [6]: The *logical-expression* is true. (Since a *logical-expression* may not contain *variable-references*, this form does not constrain the activities of any variable; rather, it specifies a condition on the model's set and parameter data. Hence this form can be used to insure that the model's data are valid, as demonstrated in the examples of Part III.)

§D.5 Alias element (optional)

---

constraint-alias-element →

string [§3.1]

The sequence of characters represented by the *string* is the alias (§1.2) for the declaration.

Default: No alias is defined.

§D.6 Comment element (optional)

---

constraint-comment-element →

string [§3.1]

The sequence of characters represented by the *string* is a comment to accompany the declaration.

Default: No comment is defined.

E OBJECTIVE DECLARATIONS

§E.1 Name element (required)

---

objective-name-element →

name [§5.1]

This element specifies the name (§1.2) for the declaration.

No *objective-name-element* may be the same as any other *objective-name-element*, *set-name-element* (§A.1), *parameter-name-element* (§B.1), *variable-name-element* (§C.1), *constraint-name-element* (§D.1), or *index-name* (§6.2) defined in the model.

§E.2 Attribute element (optional)

---

objective-attribute-element →

COMPUTE
MAXIMIZE
MINIMIZE

MAXIMIZE indicates that: (a) the declared objective, or objectives in the declared group, specify linear combinations of the model's variables; and (b) the value represented by any such objective may meaningfully be maximized subject to conditions imposed by the model on the variables' feasibility (§§C.2, D.4).

MINIMIZE is analogous to MAXIMIZE, except that it indicates objectives whose values may meaningfully be minimized.

COMPUTE indicates that no assumptions of the sort specified by MAXIMIZE or MINIMIZE are to be made.

Default: COMPUTE is assumed.

§E.3 Indexing element (optional)

---

objective-indexing-element →

indexing-expression [§6.3]

A group (§1.2) of objectives is declared if and only if this element is present. The group contains one objective corresponding to each index (§6.1) specified by the *indexing-expression*.

Default: A single (§1.2) objective component is declared.

§E.4 Specification element (required)

---

objective-specification-element →

arithmetic-expression [§2.4]

This element indicates what each declared objective represents, as follows:

If the declaration is for a single objective, it represents the function of the variables' activities determined by evaluating the *arithmetic-expression*.

If the declaration is for a group of objectives, each objective in the group represents the function of the variables determined by interpreting the *arithmetic-expression* with respect to that objective's corresponding index (§§6.6, E.3).

If the declaration's *objective-attribute-element* (§E.2) is MAXIMIZE or MINIMIZE, the *arithmetic-expression* must be a *linear-arithmetic-expression* (§2.6).

§E.5 Alias element (optional)

---

objective-alias-element →

string [§3.1]

The sequence of characters represented by the *string* is the alias (§1.2) for the declaration.

Default: No alias is defined.

§E.6 Comment element (optional)

---

objective-comment-element →

string [§3.1]

The sequence of characters represented by the *string* is a comment to accompany the declaration.

Default: No comment is defined.

PART III

EXAMPLES OF XML MODELS

EXAMPLE 1:
A MULTIPLE-PERIOD INPUT-OUTPUT MODEL

This problem is a variation on common input-output economic models. A fixed set of activity matrices is used to model production over many periods. The objective is maximum cumulative production in one industry (rather than minimum cost, which is more common).

The model given here is a generalization of an example presented in G. Hadley's Linear Programming [3], problem 13-23, page 513. An account of how a typical matrix-generator system handles the same model may be found in the DATAMAT Reference Manual [5], Part 1, Example 2.

The transcription to XML is fairly straightforward. Original parameter and variable names have been retained in the XML model for clarity; alias elements could be added if more mnemonic names were desired for reporting. Note that the model declares a group of objectives, each representing total production of a different industry in set OBJ: the modeler may choose to maximize any of these, or even several successively.

ORIGINAL FORMULATION, EXAMPLE 1

An economy comprises a variety of industries, each manufacturing a particular product. Production is to be modeled over a number of time periods, subject to the following constraints:

- There is an initial stock of each product. Stocks may be built up or run down in subsequent periods.
- Each industry requires certain fixed amounts of various inputs for each unit of its product manufactured. The inputs are of two sorts: endogenous inputs which are products of industries in the economy, and exogenous inputs whose supplies are postulated (labor, for instance).
- Each industry has an initial capacity. Capacities may be increased (but not decreased) in any period, but the added capacity may not be used until the following period. Analogously to production, each industry requires certain fixed amounts of various inputs -- endogenous and exogenous -- for each unit of increase in capacity.
- There is an initial supply of each exogenous input; the supply increases by a fixed percentage in each subsequent period.
- Each industry must satisfy an exogenous demand for its product in each period. There is an initial exogenous demand for each product, and this demand increases by a fixed percentage in each subsequent period.

The objective is to maximize the total production of a particular industry over all periods.

To express the problem as a linear program, let T be the number of periods, n the number of industries, and $\hat{n}$ the number of exogenous inputs. The variables may then be specified as:

$s_i(t)$	stock of product i at beginning of period t; $i = 1, \dots, n$; $t = 1, \dots, T+1$
$x_i(t)$	quantity of product i manufactured in period t; $i = 1, \dots, n$; $t = 1, \dots, T$
$r_i(t)$	increase in capacity of industry i in period t; $i = 1, \dots, n$; $t = 1, \dots, T$

The parameters of the problem can be specified as four matrices and six vectors, whose elements are:

A_{ij}	number of units of product i required to produce 1 unit of product j; $i = 1, \dots, n$; $j = 1, \dots, n$
$\hat{A}_{ij}$	number of units of exogenous input i required to produce 1 unit of product j; $i = 1, \dots, \hat{n}$; $j = 1, \dots, n$
D_{ij}	number of units of product i required to increase capacity of industry j by 1 unit; $i = 1, \dots, n$; $j = 1, \dots, n$
$\hat{D}_{ij}$	number of units of exogenous input i required to increase capacity of industry j by 1 unit; $i = 1, \dots, \hat{n}$; $j = 1, \dots, n$
e_i	initial stock of product i; $i = 1, \dots, n$
c_i	initial capacity of industry i; $i = 1, \dots, n$
$\hat{c}_i$	initial supply of exogenous input i; $i = 1, \dots, \hat{n}$
γ_i	fractional increase in supply of exogenous input i per period; $i = 1, \dots, \hat{n}$
b_i	initial exogenous demand for product i; $i = 1, \dots, n$
β_i	fractional increase in exogenous demand for product i per period; $i = 1, \dots, n$

The objective is to maximize the total production, $\Sigma_t \, x_z(t)$, of some industry z. The constraints may be expressed in five classes. First are the <u>initial stock constraints</u>

$$s_i(1) = e_i \qquad\qquad i = 1, \dots, n$$

Second are the production constraints, which specify that the quantity of a product manufactured in a period equals (i) the quantity required by all industries for production in the period, plus (ii) the quantity required by all industries for expansion of capacity in the period, plus (iii) the exogenous demand in the period, plus (iv) the net change in stocks:

$$x_i(t) = \sum_j A_{ij} x_j(t) + \sum_j D_{ij} r_j(t) + (1+\beta_i)^{t-1} b_i + s_i(t+1) - s_i(t)$$
$$i = 1, \ldots, n; \; t = 1, \ldots, T$$

Third, capacity constraints dictate that production must not exceed an industry's capacity, which is its initial capacity plus the sum of all increases in prior periods:

$$x_i(t) \leq c_i + \sum_{\tau=1}^{t-1} r_i(\tau) \qquad i = 1, \ldots, n; \; t = 1, \ldots, T$$

Fourth, supply constraints ensure that the quantity of exogenous inputs consumed does not exceed the available supplies:

$$\sum_j \hat{A}_{ij} x_j(t) + \sum_j \hat{D}_{ij} r_j(t) \leq (1+\gamma_i)^{t-1} \hat{c}_i$$
$$i = 1, \ldots, \hat{n}; \; t = 1, \ldots, T$$

Fifth, all variables must be nonnegative.

XML REPRESENTATION, EXAMPLE 1

SETS

```
ind     COMM:  'Industries'

ex      COMM.  'Exogenous inputs'

obj     COMM:  'Subset of industries whose production may be maximized'
```

PARAMETERS

```
p       ATTR:  POSITIVE INTEGER

        COMM:  'Number of periods'

a       ATTR:  NONNEGATIVE

        INDX:  OVER ind * ind

        COMM:  'Endogenous production matrix:  a[i,j] is units of
               product i required to make 1 unit of product j'

ahat    ATTR:  NONNEGATIVE

        INDX:  OVER ex * ind

        COMM:  'Exogenous-input production matrix:  ahat[i,j] is units
               of exogenous input i required to make 1 unit of
               product j'

d       ATTR:  NONNEGATIVE

        INDX:  OVER ind * ind

        COMM:  'Endogenous capacity-increase matrix:  d[i,j] is units
               of product i required to increase capacity of industry
               j by 1 unit'

dhat    ATTR:  NONNEGATIVE

        INDX:  OVER ex * ind

        COMM:  'Exogenous-input capacity-increase matrix:  dhat[i,j]
               is units of exogenous input i required to increase
               capacity of industry j by 1 unit'

e       ATTR:  NONNEGATIVE

        INDX:  OVER ind

        COMM:  'Initial stocks of products'

c       ATTR:  NONNEGATIVE

        INDX:  OVER ind

        COMM:  'Initial capacities of industries'
```

PARAMETERS (continued)

```
     chat     ATTR:  NONNEGATIVE

              INDX:  OVER ex

              COMM:  'Initial supplies of exogenous inputs'

     gamma    ATTR:  NONNEGATIVE

              INDX:  OVER ex

              COMM:  'Fractional increases (per period) in supplies of
                     exogenous inputs'

     b        ATTR:  NONNEGATIVE

              INDX:  OVER ind

              COMM:  'Initial exogenous demands for products'

     beta     ATTR:  NONNEGATIVE

              INDX:  OVER ind

              COMM:  'Fractional increases (per period) in exogenous demands'
```

VARIABLES

```
     s        ATTR:  NONNEGATIVE

              INDX:  OVER ind, FROM 1 TO p+1

              COMM:  'Stocks:  s[i,t] is stock of product i at beginning
                     of period t'

     x        ATTR:  NONNEGATIVE

              INDX:  OVER ind, FROM 1 TO p

              COMM:  'Production:  x[i,t] is quantity of product i
                     manufactured in period t'

     r        ATTR:  NONNEGATIVE

              INDX:  OVER ind, FROM 1 TO p

              COMM:  'Capacity increase:  r[i,t] is increase in capacity
                     of industry i in period t'
```

OBJECTIVES

```
     prod     ATTR:  MAXIMIZE

              INDX:  i OVER obj

              SPEC:  SIGMA t FROM 1 TO p (x[i,t])

              COMM:  'Maximize total production (over all periods) in
                     objective industry'
```

CONSTRAINTS

```
    subset  SPEC:  obj IN ind

            COMM:  'Objective industries are a subset of all industries.'

    init    ATTR:  BOUND

            INDX:  i OVER ind

            SPEC:  s[i,1] = e[i]

            COMM:  'For each industry:  stock variable for period 1 must
                   equal initial stock.'

    prod    INDX:  i OVER ind, t FROM 1 TO p

            SPEC:  x[i,t] = SIGMA j OVER ind (a[i,j] * x[j,t]) +
                            SIGMA j OVER ind (d[i,j] * r[j,t]) +
                            (1+beta[i])**(t-1) * b[i] + s[i,t+1] - s[i,t]

            COMM:  'For each industry in each period:  production must
                   equal the sum of:
                        (1) Output used in endogenous production,
                        (2) Output used in increasing capacity,
                        (3) Output absorbed by exogenous demands, and
                        (4) Net change in stocks.'

    cap     INDX:  i OVER ind, t FROM 1 TO p

            SPEC:  x[i,t] <= c[i] + SIGMA tt FROM 1 TO t-1 (r[i,tt])

            COMM:  'For each industry in each period:  production must
                   not exceed initial capacity plus total of capacity
                   added in previous periods.'

    sup     INDX:  i OVER ex, t FROM 1 TO p

            SPEC:  SIGMA j OVER ind (ahat[i,j] * x[j,t]) +
                   SIGMA j OVER ind (dhat[i,j] * r[j,t])
                                   <= (1+gamma[i])**(t-1) * chat[i]

            COMM:  'For each exogenous input in each period:  total input
                   used in production plus total input used to increase
                   capacity must not exceed current supply.'
```

EXAMPLE 2:

A MODEL FOR ALLOCATING TRAIN CARS

This model describes the allocation of cars to trains in a route network, given a schedule and demands for each scheduled train. The objective may be to minimize total cars, total car-miles, or some tradeoff between the two.

The original formulation given below is adapted from a study of service requirements in the Northeast Corridor [1]. The equivalent XML model differs mainly in employing more mnemonic terminology. Note the convenience of using a set of ordered quadruples to represent the schedule.

ORIGINAL FORMULATION, EXAMPLE 2

A uniform fleet of passenger cars provides railroad service to a set of cities. Service is offered by means of a set of scheduled "trains", each comprising one or more cars and running between a given pair of cities. At any given time, each car in the fleet is either part of some currently running train, or is sitting in storage at one of the cities.

Three things constrain the size and deployment of the fleet: a fixed schedule, known demands for scheduled trains, and a standard station length at all cities.

Fixed schedule. The schedule lists all trains that depart in a chosen schedule-period (a day, for example). During the schedule-period, every scheduled train must be run, carrying one or more cars.

It is assumed that each schedule-period is followed immediately by another, identical schedule-period. Moreover, the same service is to be provided in every schedule-period: that is, the same schedule must be run, with the same allocation of cars to cities and trains.

Each entry in the schedule specifies a city of departure and a city of arrival, and corresponding departure and arrival times. In general, a train may arrive during the schedule-period (e.g., day) of departure, or during any subsequent period. For simplicity, however, it is assumed here that every train arrives either in the same period, or at an earlier time in the next period. (If the schedule-period is a day, this just says that a train arrives either the same day that it leaves, or the next day; and that every trip lasts less than 24 hours.)

A car that arrives at city c at time t is free to leave c in any scheduled train that departs at t or later. (Stopover delays at the arrival city -- to discharge and board passengers, for example -- are considered part of the preceding trip, and are reflected by adjusting the arrival time in the schedule accordingly.)

Demands. For each scheduled train there is a known demand which must be met; hence there is a minimum number of cars required in each train. A train may be larger than its minimum size, however, if circumstances require that extra (deadhead) cars be shifted from one city to another.

Station length. Stations' loading platforms can accomodate only a certain number of cars (assumed to be fixed throughout the system). If the demand for a scheduled train exceeds this number, two or more sections are run.

For each train, there is a minimum number of sections that can meet demand. Since sections are expensive, it is required that no train be run with more than the minimum number of sections. This requirement places an upper limit on the number of cars (including deadheads) assigned to a train.

All of these requirements can be expressed formally as linear constraints on variables. To begin, define the following sets:

C	The set of cities
$T = \{0, \ldots, \tau-1\}$	A set of τ times into which the schedule-period is divided.
$S \subset \{(c_1,t_1,c_2,t_2)$:	$c_1 \varepsilon C,\ c_2 \varepsilon C,\ t_1 \varepsilon T,\ t_2 \varepsilon T;\ c_1 \neq c_2\}$ The schedule: each member represents a train that leaves city c_1 at time t_1 and arrives at city c_2 at t_2

Let ℓ be the maximum length of a section. Represent the demands by

$d_{c_1c_2}[t_1,t_2]$	The smallest (integral) number of cars required to meet demand for train $(c_1,t_1,c_2,t_2) \in S$

Define two collections of variables:

$u_c[t]$	Unused cars stored at city c in the interval beginning at time t, for all $c \in C$, $t \in T$
$x_{c_1c_2}[t_1,t_2]$	Number of cars assigned to the scheduled train that leaves c_1 at t_1 and arrives at c_2 at t_2

The model can now be expressed by three sets of constraints. First, all cars must be accounted for at each time in each city:

$$u_c[t] = u_c[(t-1) \bmod \tau] + \sum_{(c_1,t_1,c,t)} x_{c_1c}[t_1,t] - \sum_{(c,t,c_2,t_2)} x_{cc_2}[t,t_2] \qquad \text{for all } c \in C,\ t \in T$$

(In words: cars in storage at c at time t must equal cars in storage in the preceding interval, plus cars that arrived in trains at t, less cars that left in trains at t.) Second, the number of cars in each train must both meet demand and require no superfluous sections:

$$d_{c_1c_2}[t_1,t_2] \le x_{c_1c_2}[t_1,t_2] \le \ell \cdot \lceil d_{c_1c_2}[t_1,t_2]/\ell \rceil \qquad \text{for all } (c_1,t_1,c_2,t_2) \in S$$

Third, the number of cars stored must be nonnegative:

$$u_c[t] \ge 0 \qquad \text{for all } c \in C,\ t \in T$$

In addition, a useful solution must deal only in integral numbers of cars; but, fortunately, these constraints have a special form that guarantees integrality of every basic solution produced by the simplex algorithm.

It remains to formulate some linear objectives for the model. Two simple ones are as follows:

Cars. Minimize the number of cars in the system, expressed as the following linear form:

$$\sum_{c \in C} u_c[\tau-1] + \sum_{\substack{(c_1,t_1,c_2,t_2) \in S \\ t_2 < t_1}} x_{c_1 c_2}[t_1,t_2]$$

This counts all cars at the last interval, $\tau-1$, of the schedule-period. The first sum is all cars in storage during this interval, while the second counts all cars in transit during the interval.

Miles. Minimize total car-miles run in a schedule-period. Letting $m_{c_1 c_2}$ be the distance from c_1 to c_2, this objective is also a linear form:

$$\sum_{(c_1,t_1,c_2,t_2) \in S} m_{c_1 c_2} \, x_{c_1 c_2}[t_1,t_2]$$

XML REPRESENTATION, EXAMPLE 2

SETS

```
cities          COMM:   'Set of cities'

times           SPEC:   SEQ(0,intervals-1)

                COMM:   'Set of intervals into which a schedule-period
                        is divided'

schedule        ATTR:   LENGTH 4

                COMM:   'Member (c1,t1,c2,t2) of this set represents a train
                        that leaves city c1 at time t1 and arrives at city c2
                        at t2'
```

PARAMETERS

```
intervals       ATTR:   POSITIVE INTEGER

                COMM:   'Number of intervals into which a schedule-period
                        is divided'

section         ATTR:   POSITIVE INTEGER

                COMM:   'Maximum number of cars in one section of a train'

demand          ATTR:   POSITIVE INTEGER

                INDX:   OVER schedule

                COMM:   'For each scheduled train:  smallest (integral) number
                        of cars that meets demand for the train'

distance        ATTR:   POSITIVE

                INDX:   OVER PROJ(schedule,1,3)

                COMM:   'Inter-city distances:  distance[c1,c2] is miles
                        between city c1 and city c2'
```

VARIABLES

```
u               ATTR:   NONNEGATIVE

                INDX:   OVER cities * times

                ALIAS:  'unused'

                COMM:   'Storage variables.  u[c,t] is number of unused cars
                        stored at city c in the interval beginning at time t'

x               INDX:   OVER schedule

                ALIAS:  'train'

                COMM:   'Train variables:  x[c1,t1,c2,t2] is number of cars
                        assigned to scheduled train that leaves c1 at t1 and
                        arrives c2 at t2'
```

OBJECTIVES

```
cars          ATTR:  MINIMIZE

              SPEC:  SIGMA c OVER cities (u[c,intervals-1]) +
                     SIGMA (c1,t1,c2,t2) OVER schedule
                                         WHERE t2 < t1 (x[c1,t1,c2,t2])

              COMM:  'Number of cars in the system:  Sum of unused cars
                     and cars in trains during the last interval of the
                     schedule-period.'

miles         ATTR:  MINIMIZE

              SPEC:  SIGMA (c1,t1,c2,t2) OVER schedule
                                         (distance[c1,c2] * x[c1,t1,c2,t2])

              COMM:  'Total car-miles run by all scheduled trains in
                     one schedule-period.'
```

CONSTRAINTS

```
city_sched    INDX:  (c1,c2) OVER PROJ(schedule,1,3)

              SPEC:  c1 IN cities AND c2 IN cities AND c1 ~= c2

              COMM:  'Every train in the schedule must go between two
                     different recognized cities.'

time_sched    INDX:  (t1,t2) OVER PROJ(schedule,2,4)

              SPEC:  t1 IN times AND t2 IN times

              COMM:  'Every train in the schedule must leave and arrive
                     at recognized times.'

account       INDX:  c OVER cities, t OVER times

              SPEC:  u[c,t] = u[c,(t-1) MOD intervals] +
                      SIGMA (c1,t1) OVER SECT(schedule,c,3,t,4) (x[c1,t1,c,t]) -
                      SIGMA (c2,t2) OVER SECT(schedule,c,1,t,2) (x[c,t,c2,t2])

              COMM:  'For every city and time:  Unused cars in present
                     interval must equal unused cars in previous interval,
                     plus cars just arriving in trains, minus cars just
                     leaving in trains.'

satisfy       ATTR:  BOUND

              INDX:  (c1,t1,c2,t2) OVER schedule

              SPEC:  demand[c1,t1,c2,t2]
                          x[c1,t1,c2,t2] <=
                          section * CEIL(demand[c1,t1,c2,t2]/section)

              COMM:  'For each scheduled train:  Number of cars must meet
                     demand, but must not be so great that unnecessary
                     sections are run.'
```

EXAMPLE 3:
MODELING ALTERNATIVE ENERGY SOURCES

This model was developed by Alan S. Manne and Oliver S. Yu [4] to project the interaction of alternative future electricity-generating systems, with their corresponding fuel needs and limits, under certain assumptions of demand growth.

The original description of the model is reproduced below, followed by a fairly straightforward XML transcription. Periods numbered through 75 in the original have been changed in XML to actual year numbers (1970 through 2045). For clarity, all parameters of the XML model are symbolic: the explicit numerical values for some parameters in the original would be included with the rest of the XML model's data.

ORIGINAL FORMULATION, EXAMPLE 3

Unknowns:

PC_i^t = base-load equivalent capacity, source i, period t

DP_i^t = annual rate of capacity, source i, period t

(units of measurement: TW = terawatts = 10^3 GW

= $6.57 \cdot 10^{12}$ KWH/year.)

$NURQ_\ell^t$ = annual requirements for natural uranium, cost level ℓ, period t

(Units of measurement: 10^6 tons)

Note: A bar above a symbol indicates that the particular variable is exogenously specified.

Constraints:

CP_i^t - capacities, energy sector (i = COAL, LWR, FBR, ADV)

$$\begin{bmatrix}\text{capacity}\\ \text{and produc-}\\ \text{tion, current}\\ \text{period}\end{bmatrix} = \begin{bmatrix}\text{capacity and}\\ \text{production,}\\ \text{3 years}\\ \text{previously}\end{bmatrix} + 3 \left[\begin{array}{lcl}\text{annual capacity} & & \text{annual capacity}\\ \text{increase,} & - & \text{retirement,}\\ \text{current period} & & \text{after 30 years}\\ & & \text{of service}\end{array}\right]$$

$$PC_i^t = PC_i^{t-3} + 3\,[\,DP_i^t - DP_i^{t-30}\,]$$

where $PC_i^0 = 0$

NC_i^t - new capacity shares (i = COAL, LWR, FBR, ADV)

These represent limitations on the availability and rate of adoption of new technologies.

$$DP_{COAL}^t \geq \theta^t \; [DP_{COAL}^t + DP_{LWR}^t + DP_{FBR}^t]$$

where $\theta^t = .30$ for $t = 3, 6, \ldots, 30$

and $\theta^t = .24, .18, .12, .06$

for $t = 33, 36, 39, 42,$ respectively.

$$DP_{LWR}^t \leq .00148t \quad \text{for } t = 3, 6, \ldots, 30.$$

$$DP_{FBR}^t \leq \theta^t \; [DP_{COAL}^t + DP_{LWR}^t + DP_{FBR}^t]$$

where $\theta^t = .10, .20, .40, .60$

for $t = 24, 27, 30, 33,$ respectively.

Note: $DP_{FBR}^t = 0$ for $t = 3, 6, \ldots, 21.$

$$DP_{ADV}^t \leq \theta^t \; [DP_{COAL}^t + DP_{LWR}^t + DP_{FBR}^t + DP_{ADV}^t]$$

where $\theta^t = .10, .20, .30, .40, .50, .60$

for $t = 45, 48, 51, 54, 57, 60,$ respectively.

Note: $DP_{ADV}^t = 0$ for $t = 3, 6, \ldots, 42.$

DM^{t}_{ELEC} - final demands

$$\begin{bmatrix}\text{remaining}\\ \text{initial}\\ \text{electric}\\ \text{capacity,}\\ \text{fossil-}\\ \text{fired,}\\ \text{exogenous}\end{bmatrix} + \begin{bmatrix}\text{hydro-}\\ \text{electric}\\ \text{capacity,}\\ \text{exogenous}\\ \\ \\ \end{bmatrix} + \begin{bmatrix}\text{new electric capacity,}\\ \text{endogenous}\\ \\ \\ \\ \\ \end{bmatrix} \geq \begin{bmatrix}\text{final}\\ \text{demands}\\ \text{for}\\ \text{electricity,}\\ \text{exogenous}\\ \\ \end{bmatrix}$$

$$\overline{RI}^{\,t}_{FOSS} + \overline{PC}^{\,t}_{HYDR} + [PC^{t}_{COAL} + PC^{t}_{LWR} + PC^{t}_{FBR} + PC^{t}_{ADV}] \geq \overline{DM}^{\,0}_{ELEC}\ (1.05)^{t}$$

where $\overline{DM}^{\,0}_{ELEC} = \dfrac{1.53 \quad 10^{12} \text{ KWH/year}}{8.76(.75)\ 10^{3} \quad \text{H/year}} = .233 \quad \text{TW}$

$$\overline{PC}^{\,t}_{HYDR} = \frac{.247}{8.76(.75)}\ (1.01)^{t} = .038\ (1.01)^{t} \text{ TW}$$

$$\overline{RI}^{\,0}_{FOSS} = \frac{1.53-.247}{8.76(.75)} = .195 \quad \text{TW}$$

The retirement schedule for the remaining initial electric capacity from fossil fuel ($\overline{RI}^{\,t}_{FOSS}$) is calculated on the basis of a 30-year service life, assuming that the capacity increments grew at the annual rate of 7% during the 30 years preceding time 0.

SM^{t}_{COAL} - cumulative sum of coal cunsumption $(10^{18}$ BTU)

$$\begin{bmatrix}\text{cumulative sum,}\\ \text{end of current}\\ \text{period}\end{bmatrix} = \begin{bmatrix}\text{cumulative sum,}\\ \text{3 years}\\ \text{previously}\end{bmatrix} + 3 \begin{bmatrix}10^{3}\text{ hours}\\ \text{per year}\\ \ \end{bmatrix} \begin{bmatrix}\text{heat}\\ \text{rate}\\ \ \end{bmatrix} \begin{bmatrix}\text{base-load}\\ \text{equivalent}\\ \text{capacity}\end{bmatrix}$$

$$CS^{t}_{COAL} = CS^{t-3}_{COAL} + 3\,[8.76(.75)]\,[.01\ ]\,[.54\ \overline{RI}^{\,t}_{FOSS} + PC^{t}_{COAL}]$$

SM^{t}_{NATU} - natural uranium requirements $(10^{6}$ tons)

$$\begin{bmatrix}\text{current annual}\\ \text{consumption}\\ \text{of uranium}\\ \ \end{bmatrix} \geq \begin{bmatrix}\text{annual}\\ \text{refueling}\\ \text{require-}\\ \text{ments}\end{bmatrix} + \begin{bmatrix}\text{annual requirements}\\ \text{for initial}\\ \text{inventories, next}\\ \text{period}\end{bmatrix}$$

$$\textstyle\sum_{\ell=1}^{10} NURQ^{t}_{\ell} \geq .18\ PC^{t}_{LWR} + .50\ DP^{t+3}_{LWR}$$

$CRQU_{\ell}$ - upper bound on uranium consumption at cost level ℓ $(10^{6}$ tons)

$$\begin{bmatrix}\text{cumulative uranium}\\ \text{consumption at cost}\\ \text{level } \ell\end{bmatrix} \leq \begin{bmatrix}\text{cumulative availability of uranium at}\\ \text{cost level } \ell\text{, exogenous}\\ \ \end{bmatrix}$$

$$3 \textstyle\sum_{t=0}^{75} NURQ^{t}_{\ell} \leq \overline{CAVU}_{\ell} \qquad \ell = 1, 2, \ldots, 10$$

COST - minimand

Present value of costs incurred annually during each 3-year period over 75-year horizon:

$$\left[\begin{array}{l}\text{present}\\\text{value of}\\\text{3-year}\\\text{costs}\\ \\ \\ \end{array}\right] \left[\left(\begin{array}{l}\text{current}\\\text{costs, annual}\end{array}\right) + \left(\begin{array}{l}\text{investment}\\\text{costs, annual}\end{array}\right)\left(\begin{array}{l}\text{terminal}\\\text{valuation}\\\text{factor,}\\\text{30-year}\\\text{service}\\\text{life}\end{array}\right)\left(\begin{array}{l}\text{present value}\\\text{factor for}\\\text{incurring}\\\text{capital costs}\\\text{2 years prior}\\\text{to period t}\end{array}\right)\right]$$

$$\left[\sum_{t=0}^{75} 3\beta^{t} \quad \left[\sum_{i} cur_{i}\, PC_{i}^{t} \quad + \quad \left(\sum_{i} cap_{i}\, DP_{i}^{t}\right) \quad (1 - TV_{t}) \quad (\beta^{-2}) \quad \right]\right]$$

(unit: $\$10^{9}$)

Where $\beta = \frac{1}{1+r}$ = one year present-value factor at r% discount rate

$TV_{t} = \beta^{78-t}$ for $t > 45$; 0 otherwise

It is supposed that interest during construction is included in the capital cost coefficients, cap_{i}. These costs are incurred at the commissioning date -- two years prior to full power operations -- hence the term β^{-2}.

XML REPRESENTATION, EXAMPLE 3

SETS

```
source          SPEC:  {coal: 'coal-fueled fossil plants',
                        lwr:  'light-water reactors',
                        fbr:  'fast breeder reactors',
                        adv:  'advanced technology'}

                COMM:  'Energy sources'

cost            COMM:  'Arbitrary set of uranium cost levels'

horizon         SPEC:  SEQ(1970,2045,3)

                COMM:  'Planning horizon'
```

PARAMETERS

```
pv              ATTR:  POSITIVE

                COMM:  'One-year present-value factor'

tv              INDX:  t OVER horizon

                SPEC:  (pv**(2048-t)) * MAX(0,t-2000)/(t-2000)

                COMM:  'Terminal valuation factors:  tv[t] = pv ** (2048-t)
                       if t > 2000, = 0 otherwise'

cur             ATTR:  POSITIVE

                INDX:  OVER source

                COMM:  'Annual current costs'

cap             ATTR:  POSITIVE

                INDX:  OVER source

                COMM:  'Annual capital costs'

sh_coal         ATTR:  NONNEGATIVE

                INDX:  FROM 1973 TO 2012 BY 3

                COMM:  'Minimum share in new capacity, coal'

sh_lwr          ATTR:  NONNEGATIVE

                COMM:  Factor for maximum new capacity, lwr'

sh_fbr          ATTR:  NONNEGATIVE

                INDX:  FROM 1994 TO 2003 BY 3

                COMM:  'Maximum share in new capacity, fbr'
```

PARAMETERS (continued)

sh_adv	ATTR:	NONNEGATIVE
	INDX:	FROM 2015 TO 2030 BY 3
	COMM:	'Maximum share in new capacity, adv'
dm_elec	ATTR:	POSITIVE
	COMM:	'Initial final demand for electricity'
pc_hydr	ATTR:	POSITIVE
	COMM:	'Initial hydro-electric capacity'
ri_foss	ATTR:	NONNEGATIVE
	INDX:	OVER horizon
	COMM:	'Remaining initial electric capacity, fossil fired'
cavu	ATTR:	POSITIVE
	INDX:	OVER cost
	COMM:	'Cumulative availabilities of uranium'

VARIABLES

pc	ATTR:	NONNEGATIVE
	INDX:	OVER source * horizon
	COMM:	'Base-load equivalent capacities, each source and period'
dp	ATTR:	NONNEGATIVE
	INDX:	OVER source * horizon
	COMM:	'Annual rate of addition of new capacity, each source and period'
nurq	ATTR:	NONNEGATIVE
	INDX:	OVER cost * horizon
	COMM:	'Annual requirement for natural uranium, each cost level and period'
cs_coal	ATTR:	NONNEGATIVE
	INDX:	OVER horizon
	COMM:	'Cumulative sum of coal consumption, end of each period'

OBJECTIVES

```
cost            ATTR:  MINIMIZE

                SPEC:  SIGMA t OVER horizon (3 * pv**(t-1970) *
                        (SIGMA i OVER source (cur[i] * pc[i,t]) +
                         SIGMA i OVER source (cap[i] * dp[i,t]) *
                                                   (1-tv[t]) / pv**2) )

                COMM:  'Sum of present values of current costs and
                       investment costs incurred in all periods over
                       horizon'
```

CONSTRAINTS

```
pc_init         ATTR:  BOUND

                INDX:  i OVER source

                SPEC:  pc[i,1970] = 0

                COMM:  'Initial base-load equivalent capacities are zero.'

cp              INDX:  i OVER source, t OVER horizon WHERE t ~= 1970

                SPEC:  pc[i,t] = pc[i,t-3] + 3*(dp[i,t]-dp[i,MAX(1970,t-30)])

                COMM:  'For each source and period:  Capacity now must
                       equal capacity 3 years ago, plus change in capacity
                       over previous 3 years (assuming retirement after
                       30 years' service).'

init_dp_coal    ATTR:  BOUND

                SPEC:  dp['coal',1970] = 0

                COMM:  'Initialize dp for coal.'

nc_coal         INDX:  t FROM 1973 TO 2012 BY 3

                SPEC:  dp['coal',t] >= sh_coal[t] *
                                  (dp['coal',t]+dp['lwr',t]+dp['fbr',t])

                COMM:  'From 1973 to 2012:  Coal must have at least a
                       certain share of new capacity.'

init_dp_lwr     ATTR:  BOUND

                SPEC:  dp['lwr',1970] = 0

                COMM:  'Initialize dp for lwr.'

nc_lwr          ATTR:  BOUND

                INDX:  t FROM 1973 TO 2000 BY 3

                SPEC:  dp['lwr',t] <= sh_lwr * (t-1970)

                COMM:  'From 1973 to 2000:  New capacity in lwr cannot
                       exceed sh_lwr per year.'
```

CONSTRAINTS (continued)

```
init_dp_fbr  ATTR: BOUND

             INDX: t FROM 1970 TO 1991 BY 3

             SPEC: dp['fbr',t] = 0

             COMM: 'No fbrs before 1994.'

nc_fbr       INDX: t FROM 1994 TO 2003 BY 3

             SPEC: dp['fbr',t] <= sh_fbr[t] *
                                  (dp['coal',t]+dp['lwr',t]+dp['fbr',t])

             COMM: 'From 1994 to 2003:  fbr must have at most a
                   certain share of new capacity.'

init_dp_adv  ATTR: BOUND

             INDX: t FROM 1970 TO 2012 BY 3

             SPEC: dp['adv',t] = 0

             COMM: 'No adv before 2015.'

nc_adv       INDX: t FROM 2015 TO 2030 BY 3

             SPEC: dp['adv',t] <= sh_adv[t] * SIGMA i OVER source (dp[i,t])

             COMM: 'From 2015 to 2030:  adv must have at most a
                   certain share of new capacity.'

fdm_elec     INDX: t OVER horizon

             SPEC: ri_foss[t] + 1.01**(t-1970) * pc_hydr
                             + SIGMA i OVER source (pc[i,t])
                                         >= 1.05**(t-1970) * dm_elec

             COMM: 'For each period:  Final demand for electricity must
                   be met by sum of:  remaining initial fossil-fired
                   electric capacity (exogenous); hydro-electirc
                   capacity (exogenous); and new endogenous capacities.'

init_cs_coal ATTR: BOUND

             SPEC: cs_coal[1970] = 0

             COMM: 'Initial cumulative sum of coal consumption is zero.'

sm_coal      INDX: t OVER horizon WHERE t ~= 1970

             SPEC: cs_coal[t] = cs_coal[t-3] +
                               1.971 * (0.54*ri_foss[t] + pc['coal',t])

             COMM: 'For each period:  Cumulative sum of coal consumption
                   equals cumulative sum to previous period plus
                   additional consumption during period.'
```

CONSTRAINTS (continued)

```
sm_natu         INDX:  t OVER horizon WHERE t ~= 2045

                SPEC:  SIGMA c OVER cost (nurq[c,t])
                                 >= 0.18*pc['lwr',t] + 0.50*dp['lwr',t+3]

                COMM:  'For each period:  Annual consumption of uranium
                       must equal or exceed annual refueling requirements
                       plus annual requirement for initial inventories
                       in next period.'

crqu            ATTR:  GUB

                INDX:  c OVER cost

                SPEC:  3 * SIGMA t OVER horizon (nurq[c,t]) <= cavu[c]

                COMM:  'For uranium at each cost level:  Cumulative
                       consumption must not exceed cumulative availability.'
```

REFERENCES

[1] Fourer, Robert, Judith B. Gertler, and Howard J. Simkowitz, "Models of Railroad Passenger-Car Requirements in the Northeast Corridor". Annals of Economic and Social Measurement, vol. 6, no. 4 (1977), pp. 367-398.

[2] Fourer, Robert and Michael J. Harrison, "A Modern Approach to Computer Systems for Linear Programming". Working Paper 988-78, Center for Computational Research in Economics and Management Science, Alfred P. Sloan School of Management, MIT. March 1978.

[3] Hadley, G., Linear Programming. Addison-Wesley Publishing Company, Reading, Massachusetts, 1962.

[4] Manne, Alan S. and Oliver S. Yu, "Breeder Benefits and Uranium Ore Availability" (preliminary draft). Electric Power Research Institute, Palo Alto, California, October 1974.

[5] National Bureau of Economic Research, DATAMAT Reference Manual (Third Edition). NBER Computer Research Center for Economics and Management Science, order no. D0078, July 1975.