NBSIR 85-3125

Problem and Data Specification for Linear Programs

Christoph Witzgall
Marjorie McClain

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

November 1984

Issued April 1985

Sponsored by:
U.S. Department of Transportation
Urban Mass Transportation Administation (UMTA)

NBSIR 85-3125

PROBLEM AND DATA SPECIFICATION FOR LINEAR PROGRAMS

Christoph Witzgall
Marjorie McClain

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

November 1984

Issued April 1985

Sponsored by:
U.S. Department of Transportation
Urban Mass Transportation Administration (UMTA)

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

Table of Contents

Abstract

A language for specifying linear programs is proposed. The specification language is designed so as to enable the user to define the input for a particular linear program in terms of a given database of multi-dimensional tables. The specification language is formulated within the general framework of the UTPS system developed by the U.S. Department of Transportation. The structure of the underlying database system is described, and instructions for the writing of reports, again within the framework of the UTPS system, are discussed. Generation of the matrix for the specified linear program can be achieved during a single sequential pass through the database.

Keywords: database, input generation, lexical analysis, lexical synthesis, linear programming, modeling language, matrix generator, modeling language, multidimensional table, optimization, problem specification, report generator, software engineering.

Acknowledgements

We are grateful for the support received from the Division of Planning, Methods and Support of the Urban Mass Transportation Administration (UMTA) in this developmental effort, and in particular, for the interest and encouragement by its former chief, Dr. R. Dial, many of whose ideas are incorporated in this work. At NBS, we acknowledge the contributions of Dr. H. Hung during an early phase of the language formulation and of Mr. M. Knapp-Cordes, who programmed a first version of the database as well as making a key contribution to the theoretical development of the matrix generator. While at NBS, Mr. J. Wolfe single-handedly implemented the problem formulation portion of the ULP prototype package. We deeply appreciate his commitment and his tireless efforts.

Introduction

In this report a method, tentatively called ULP, is proposed for specifying linear programming problems in terms of a table-oriented database system.

The position is taken that the database system is a self-contained entity, and that a linear programming code is just one of several analysis programs supported by the database system. Consequently, the task of generating and updating the database system is viewed as conceptually and mechanically different from the task of specifying a particular linear programming problem. The latter task will therefore not address the input of data but, with minor exceptions, concentrate on the selection of data from a pre-existing database.

The underlying database system is not assumed to be geared towards quick on-line retrieval of small portions of information as, for instance, in management information systems (MIS), nor is it envisioned as continually modified and appended as, for instance, in reservation systems. Rather it is seen as a fairly static repository of large blocks of information geared towards the selective off-line generation of data streams for computation-intensive analysis programs and report generation. Its purpose may also be viewed as providing a superstructure for organizing information which resides in separate files. A database consisting of multidimensional tables appears to be ideally suited for such tasks.

Both the problem specification method and the database system will be discussed in this report, the specification method in Chapter 1, the database system in Chapter 2. In Chapter 3, report generation needs are examined pertaining both to the results of linear programming runs and to the contents of the database.

Chapter 4 will deal with questions of implementation. "Tokenization", that is, lexical analysis or lexical synthesis, of the specification language as well as the generation and representation of the matrix of the linear program are discussed. A major result of theoretical as well as practical interest is the observation that matrix generation can be accomplished during a single sequential pass through the database (M. Knapp-Cordes [9]). That does not imply that the entire database need be read in a consecutive fashion but that access can be organized in an efficient manner. This is particularly important if the database information is distributed over several files.

In a seminal paper, "Modeling Languages versus Matrix Generators for Linear Programming", R. Fourer [4] conducts a comprehensive examination of the many formulation tools for linear programming that are currently available. In this context, he distinguishes a "modeler's form" from a "matrix generator form" of the linear program specification. The former concerns itself first and foremost with stating the problem at hand in a concise and self-documentary fashion. When using the matrix generator form, the user has moved already one step away from simply stating the problem: he has organized the information into a pre-matrix format expecting a matrix generator package to complete the job. Matrix generator forms, typically, refer to "rows" and "columns" of the linear program. Fourer finds that most specification methods currently in use for linear programs are of this latter kind. He goes on to propose an instance of a "modeling language" based on a modeler's form rather than a matrix generator form of a linear program. Our specification method ULP is a "modeling language" in the sense of Fourer. In fact, it is similar to the LPMODEL system

of S. Katz, L.J. Risman and M. Rodeh [7], the main differences being twofold: (i) a different implicit summation convention, fashioned after a similar convention in tensor algebra where "common indexes" are automatically summed over; (ii) the availability of an index map construct by which indexes in different index ranges can be related to each other. We reiterate our emphasis on the conceptual separation between the database on the one hand and the problem specification on the other, expecting the database to serve purposes also other than providing data support for a linear programming model.

An effort has been made to use a few flexible constructs in a logically consistent fashion rather than introducing many different constructs of narrow applicability. The reader is to judge where the logical coherence of the specification language could be improved. We note that many current specification languages use simlar notation. This may be considered as an indication of an emerging consensus, and notations proposed in this report are intentionally chosen close to established notations in such instances.

OMNI (e.g., C. Boudrye and R. Greenberg [3]) and DATAFORM (Ketron, Inc. [8]) are probably the most commonly used advanced specification methods for linear programming. DATAFORM has been extended to PLATOFORM, which is described in a separate Exxon monograph by K. H. Palmer et al. [12]. These methods have proved themselves capable of handling the large linear programs arising in practice. However, they are, as R. Fourer points out, still oriented mainly towards matrix generation rather than problem specification. In Appendix A, we will formulate for comparison two sample problems in XML, OMNI, DATAFORM, LPMODEL, and ULP (our proposed method). Other, mostly theoretical work, has been directed towards structured matrix generation, that is, generation of more complex matrices from smaller ones. See, for instance, R. Bayer and C. Witzgall [1],[2], H. J. Greenberg and James E. Kalan [5], J. R. Phillips and H. C. Adams [13]. The paper of R. Fourer [4] includes an extensive list of references on various problem specification methods.

LINDO (L. Schrage [14],[15]) is a popular linear programming system with on-line numerical specification of small linear programs in a standard arithmetic format. While any modeling language should emphasize the use of an off-line database it should — and ULP does — also encompass the specification capabilities of LINDO. S. Shen and G. Krulee [16] have proposed a system for extracting from normal English sentences a LINDO-type formulation of a linear program. The use of computer facilities to aid in the formulation of linear programs, for instance, by providing suitable menu features and screen displays, is promoted by G. Mitra and coworkers [18],[19],[20].

Report-writing involves three major aspects. One of these is "output control". This requires the capability to specify, at the outset, those quantities which will actually be produced as output for a specified linear programming model. The second aspect is "output editing", namely the selective extraction, aggregation and display of output information after it has been produced. Finally, there is the task of "output analysis". In the words of R.P. O'Neill [11], "the listings of the solution file ... are often well over 10,000 lines forcing a very tedious and awkward process for examining information on an ad hoc basis". Output analysis therefore calls for sophisticated interactive scanning tools such as the PERUSE system developed by R.P. O'Neill [11] and his collaborators. In this report, we address mainly the problem of output control. Since output will be defined in tabular form in the process of problem specification, some of the table mainpulation operations described in Chapter 2 may be potentially applicable to the task of output editing. We do not,

however, suggest methods for interactive examination of solution, although the importance of such a tool cannot be overestimated. Finally, we remark that all tools for output control, editing and examination must be based on a coherent system of problem specification.

A prototype linear programming system, ULP, has been the object of a developmental effort during the years 1979-1982 by NBS on behalf of the Urban Mass Transportation Administration (UMTA). This prototype system includes an implementation of the major problem and data specification features described in this report. ULP has been designed to work within the framework of the Urban Transportation Planning System (UTPS), which is a collection of intercommunicating software modules for transportation planning and analysis developed by the UMTA. For more detailed information about UTPS, the reader is referred to UMTA publication [17]. Knowledge of UTPS rules and conventions, however, is not required for reading this report. Furthermore, adherence to UTPS conventions does not diminish the generality of the material presented.

UTPS modules adhere to common formats of programming and documentation. A common feature are "run statements", each of which define the intent of a computer run to be made. In UTPS, these run assignments are in the form of so-called "&SELECT statements". They consist of a series of instructions each headed by a "keyword". The format is free in that the keywords and what follows them are not tied to particular positions on a line. "Ends-of-lines" are generally ignored, as are blank characters in most circumstances. An important general principle is that the sequence of the instructions in the run statement should not matter. The problem specification method discussed in this paper uses such instructions in a run statement and adheres to UTPS principles.

The instructions in run statements are divided into two classes: "specification instructions" and "disposition instructions". Disposition instructions cause the execution of computer code. In other words, they trigger a "run". Specification instructions, in contrast, only serve to provide information for a run, without causing the computer to execute. Among the instructions introduced in Chapter 1, the LPMIN and LPMAX are the sole disposition instructions, causing the execution of linear programming code. In Chapter 2, additional disposition instructions are introduced for modifying the database, as well as specification instructions for the conceptual definition of tables and ranges. Such specification instructions may also be used in conjunction with an LPMIN or LPMAX disposition.

"Data statements", which specify direct data support to a run statement, have been similarly standardized in UTPS. They consist of digital and alphanumeric information following an "&DATA card". The database input format to be proposed in Chapter 2 follows the UTPS conventions for data statements.

We conclude with the introduction some general remarks. The ULP prototype system implemented by NBS includes many, but not all, of the specification features proposed in this report. It demonstrates, however, the soundness of the implementation framework and the general capability for handling such features. The purpose of this report then is threefold: (i) to document a past extensive effort to develop and implement a useful user-interface with a linear programming package; (ii) to set forth a modeling philosophy by providing a list of potential features -- by no means exhaustive -- which are compatible with this philosophy; (iii) to stimulate discussions about the form and the scope of modeling languages for linear programming tasks.

CHAPTER 1: Formulating Linear Programs

In this chapter, the general form of linear programs, and how such programs can be described using a specification language, is discussed. Specific examples of linear programming problems have been selected to illustrate the use of various features of the specification language. The text of each particular problem specification represents a run statement as described in the Introduction. Such run statements for linear programs may also contain some types of specification instructions whose discussion will be postponed until Chapter 2 because they are mainly intended for database manipulation.

1.1 Mathematical Formulation of Linear Programs

The "Linear Programming Problem" or "LP-Problem" consists of finding the minimum or, alternatively, the maximum of a linear function,

$$c_1x_1 + \ldots + c_nx_n ,$$

of n unknown "variables",

$$x_1 , \ldots , x_n .$$

The coefficients $c_1, \ldots, c_n$ are given numbers. The linear function is called the "objective function".

For the objective function to have a minimum (or maximum), the values of the variables $x_1, \ldots, x_n$ must be subject to "constraints", namely linear equations of the form

$$a_1x_1 + \ldots + a_nx_n = b$$

or linear inequalities of the form

$$a_1x_1 + \ldots + a_nx_n \quad b$$

$$a_1x_1 + \ldots + a_nx_n \geq b$$

where $a_1, \ldots, a_n$ and b are given real numbers. Furthermore, the variables are assumed to be nonnegative, or more generally, to have specified "lower bounds"

$$x_1 \geq L_1 , \ldots , x_n \geq L_n ,$$

and "upper bounds"

$$x_1 \leq H_1 , \ldots , x_n \leq H_n ,$$

where $L_1, \ldots, L_n$ and $H_1, \ldots, H_n$ are again given real numbers. Any instance of such an objective function and constraints is commonly called a "linear program".

A particular set of values for the variables $x_1, \ldots, x_n$ forms a "feasible solution" if it satisfies the constraints as well as the bounds. If it also minimizes (or maximizes) the objective function, then it is called an "optimal solution". In certain exceptional cases, it may happen that no feasible solutions exist at all, or that feasible but no optimal solutions exist because every feasible solution can be improved.

EXAMPLE 1 (Murty [10])

A nonferrous metals corporation produces four different alloys from two basic metals. The daily total supply to be used of these metals is as follows:

SUPPLY

Metal 1	6 tons
Metal 2	5 tons

The proportions of the two metals entering into the four alloys is set forth in the following table:

COMPOSITION

	ALLOY 1	ALLOY 2	ALLOY 3	ALLOY 4
METAL 1	0.5	0.6	0.3	0.1
METAL 2	0.5	0.4	0.7	0.9

Present market prices for the four alloys are per ton in dollars:

PRICE

ALLOY 1	1000
ALLOY 2	1500
ALLOY 3	1800
ALLOY 4	4000

The problem is to determine the optimal product mix to maximize gross revenue. To this end, one may represent the unknown optimal product mix by four variables

$$x_1, x_2, x_3, x_4,$$

which determine the number of tons produced and sold daily of alloys 1-4 in that order. The price of the total daily production is then given by:

$$1000x_1 + 1500x_2 + 1800x_3 + 4000x_4 \ .$$

This is the objective function to be maximized. The constraints are determined by the supply provided for the two basic metals:

$$0.5x_1 + 0.6x_2 + 0.3x_3 + 0.1x_4 = 6 \ .$$

$$0.5x_1 + 0.4x_2 + 0.7x_3 + 0.9x_4 = 5 \ .$$

It is clear that the daily production quotas cannot be negative:

$$x_1 \geq 0 \ , \ldots , \ x_4 \geq 0 \ .$$

Assuming this, the linear program will have the form:

$$\begin{aligned} \text{maximize} \quad & 1000x_1 + 1500x_2 + 1800x_3 + 4000x_4 \\ \text{subject to} \quad & 0.5x_1 + 0.6x_2 + 0.3x_3 + 0.1x_4 = 6 \\ & 0.5x_1 + 0.4x_2 + 0.7x_3 + 0.9x_4 = 5 \ . \end{aligned}$$

The answer can be shown to be

$$x_1 = 0, \ x_2 = 9.8, \ x_3 = 0, \ x_4 = 1.2,$$

i.e., only alloys 2 and 4 are produced.

The above formulation proceeded under the assumption that the daily supply had to be used up. If one does not insist on that, then the constraints take the form of inequalities rather than equations:

$$0.5x_1 + 0.6x_2 + 0.3x_3 + 0.1x_4 \leq 6$$

$$0.5x_1 + 0.4x_2 + 0.7x_3 + 0.9x_4 \leq 5 \ .$$

In this case the linear program will be:

$$\begin{aligned} \text{maximize} \quad & 1000x_1 + 1500x_2 + 1800x_3 + 4000x_4 \\ \text{subject to} \quad & 0.5x_1 + 0.6x_2 + 0.3x_3 + 0.1x_4 \leq 6 \\ & 0.5x_1 + 0.4x_2 + 0.7x_3 + 0.9x_4 \leq 5 \ . \end{aligned}$$

Since not stated otherwise, it is automatically assumed that all variables are nonnegative. The answer is:

$$x_1 = 0, \ x_2 = 0, \ x_3 = 0, \ x_4 = 5.56 \ .$$

Still another linear program results if the condition that not more than 5 tons of alloy 2 can be sold is added to the original linear program. In this case, one would add the bound condition $x_2 \leq 5$:

$$\begin{aligned} \text{maximize} \quad & 1000x_1 + 1500x_2 + 1800x_3 + 4000x_4 \\ \text{subject to} \quad & 0.5x_1 + 0.6x_2 + 0.3x_3 + 0.1x_4 = 6 \\ & 0.5x_1 + 0.4x_2 + 0.7x_3 + 0.9x_4 = 5 \\ & x_2 \leq 5 \ . \end{aligned}$$

The answer now is

$$x_1 = 6, \ x_2 = 5, \ x_3 = 0, \ x_4 = 0,$$

that is, only alloys 1 and 2 are being produced.

1.2 Problem Specification by Keywords

Ways to represent the above examples of linear programs in computer readable form are now explored. We will first discuss six main "keywords":

LPMIN, LPMAX, UNKNOWN, CONSTRAIN, BOUND, COMMENT.

The first two keywords are used to indicate the objective function, which is to be minimized or maximized, respectively. The others specify constraints and bounds. The keyword CONSTRAIN also defines "labels" for constraints, by which these constraints can be identified and referenced. Similarly the variables will be given alphanumeric "variable names" using the keyword UNKNOWN. BOUND instructions indicate deviations from the implicit assumption of nonnegativity for variables. The keyword COMMENT fulfills an obvious function. Additional keywords will be introduced later.

For EXAMPLE 1, we might choose

X1, X2, X3, X4

as variable names, and write the following three separate problem definitions:

```
UNKNOWN(X1, X2, X3, X4)
LPMAX(1000*X1 + 1500*X2 + 1800*X3 + 4000*X4)
CONSTRAIN('1':  0.5*X1 + 0.6*X2  + 0.3*X3 + 0.1*X4 = 6)
CONSTRAIN('2':  0.5*X1 + 0.4*X2  + 0.7*X3 + 0.9*X4 = 5)
```

```
UNKNOWN(X1, X2, X3, X4)
LPMAX(1000*X1 + 1500*X2  + 1800*X3 + 4000*X4)
CONSTRAIN('1':  0.5*X1 + 0.6*X2  + 0.3*X3 + 0.1*X4 <= 6)
CONSTRAIN('2':  0.5*X1 + 0.4*X2  + 0.7*X3 + 0.9*X4 <= 5)
```

```
UNKNOWN(X1, X2, X3, X4)
COMMENT(X1,X2,X3,X4 ARE PRODUCTION QUOTAS)
LPMAX(1000*X1 + 1500*X2 + 1800*X3 + 4000*X4)
CONSTRAIN('1':  0.5*X1  + 0.6*X2  + 0.3*X3 + 0.1*X4 = 6)
CONSTRAIN('2':  0.5*X1  + 0.4*X2  + 0.7*X3 + 0.9*X4 = 5)
BOUND(X2 <= 5)
```

Each of the three problem definitions represents a run statement as defined in the Introduction.

In the above CONSTRAIN statements, the numbers 1 and 2, stated in quotes, serve as constraint labels. Arbitrary alphanumeric names starting with a letter can serve as variable names. The labels are arbitrary strings of letters and numbers. Blank spaces in both variable names and labels are optional but are ignored as far as the identity of two such names is concerned.

In general, blank spaces and ends of lines are ignored in the ULP set-up so that an almost entirely free format results. In particular, an expression of the form

Keyword( )

may stretch over several lines. On the other hand, several expressions of this kind may start in the same line. Semicolons separating such expressions are optional. The following is an equivalent formulation of the first linear program:

```
UNKNOWN(ALLOY1, ALLOY2, ALLOY3, ALLOY4)

LPMAX(1000*ALLOY1 + 1500*ALLOY2 + 1800*ALLOY3 + 4000*ALLOY4)

CONSTRAIN('ALUMINUM': 0.5*ALLOY1 + 0.6*ALLOY2 + 0.3*ALLOY + 0.1*ALLOY4 = 6)

CONSTRAIN('MAGNESIUM': 0.5*ALLOY1 + 0.4*ALLOY2 + 0.7*ALLOY3 + 0.9*ALLOY4 = 5)
```

The symbol ">=" (greater or equal) may be used in a CONSTRAIN statement just as well as "=" or "<=" (less or equal). The notations, "=>" instead of ">=" and "=<" instead of "<=", are also accepted.

In BOUND statements, the variables come first, the relation symbols second, and the constants third. All three relation symbols, ">=, <=, =" are permitted in BOUND statements.

The need for a separate BOUND statement might be questioned, because an equivalent CONSTRAIN statement could apparently be written. The function of the BOUND statement, however, is slightly different in that the setting of a lower bound by a BOUND statement overrides the implicit lower bound of zero. Also the simplex algorithm handles BOUND statements more efficiently than equivalent CONSTRAIN statements. The BOUND statement permits the setting of infinite bounds. In particular,

```
BOUND(X >= -INF)
```

sets a lower bound of $-\infty$, indicating nonexistence of finite lower bounds. This device can be used to neutralize nonnegativity conditions for the purpose of introducing completely unbounded variables.

As is customary, the ULP package introduces nonnegative "slack variables" to turn inequalities into equations. These variables are labeled by the names of their corresponding CONSTRAIN statements except that these names are preceded by

"$" (dollar sign).

In the second version of EXAMPLE 1, the two CONSTRAIN statements give rise to two slack variables

$1, $2,

the introduction of which converts the two inequalities to equations:

$$0.5*X1 + 0.6*X2 + 0.3*X3 + 0.1*X4 + \$1 = 6$$

$$0.5*X1 + 0.4*X2 + 0.7*X2 + 0.7*X4 + \$2 = 5 \quad .$$

To repeat, the definition and inclusion of slack variables is automatic: it requires no action by the user. However, the user may encounter these variables in output reports.

1.3 Tables and Index Ranges

For all but the very smallest of linear programs, it is impossible to write down explicitly all the constraints or even the objective function: the data will have to be drawn from a "database". This database is organized as a collection of "tables". A table is given by a "table name", a sequence of "index ranges", and a number, the "table entry" for each "index combination". An index range is a sequence of indexes, i.e. names of the form described in the previous section: alphanumeric strings with blanks not counting. The table names are alphanumeric strings starting with a letter and blanks not counting. Index ranges are identified by a name, which is again like that of tables: alphanumeric starting with a letter and blanks permitted but not counting.

The three tables in EXAMPLE 1 might be represented as follows:

SUPPLY(METALS)

PRICE(ALLOYS)

COMPOSITION(METALS,ALLOYS).

Here SUPPLY, PRICE, COMPOSITION are table names; METALS and ALLOYS denote index ranges, say:

METALS = {METAL 1, METAL 2}

ALLOYS = {ALLOY 1, ALLOY 2, ALLOY 3, ALLOY 4} .

In particular, we then have

SUPPLY('METAL 1') = 6

PRICE('ALLOY 1') = 1000

COMPOSITION('METAL 1','ALLOY 1') = .5 .

An important warning: the same range cannot appear twice in the same table.

The variables can similarly be tables, e.g.

X(ALLOYS).

The letter X thus acts as the name of an unknown, to-be-determined table. In order to use such "indexed variable names" in our formulation of linear programs, an index summation convention is adopted which is closely related to a convention used in tensor algebra: common indices are summed over.

Our "index summation convention" is based on the distinction between "fixed index ranges" and "free index ranges". Consider the expression

PRICE(ALLOYS)*X(ALLOYS),

with ALLOYS considered to be a free index range. Then the free indices are "summed over", and the above expression turns out to be a short-hand notation for

PRICE(ALLOY 1)*X(ALLOY 1) + PRICE(ALLOY 2)*X(ALLOY 2)

+ PRICE(ALLOY 3)*X(ALLOY 3) + PRICE(ALLOY 4)*X(ALLOY 4),

which, upon substitution of the known table entries for PRICE(ALLOY 1), and so on, becomes the objective function in our last formulation of the linear program for EXAMPLE 1.

Considering ALLOYS as a free index range and METALS as a fixed index range, the expression

COMPOSITION(METALS,ALLOYS)*X(ALLOYS)

will stand for two different sums, one for each of the two fixed indexes in the range METALS:

COMPOSITION(METAL 1,ALLOY 1)*X(ALLOY 1) +

... + COMPOSITION(METAL 4,ALLOY 1)*X(ALLOY 4)

COMPOSITION(METAL 1,ALLOY 2)*X(ALLOY 1) +

... + COMPOSITION(METAL 4,ALLOY 2)*X(ALLOY 4).

If table entries for COMPOSITION are entered into the above expression, the linear functions result which form the essential part of the constraints for EXAMPLE 1.

Employing these conventions, one may formulate a linear program for EXAMPLE 1 as follows:

```
UNKNOWN(X(ALLOYS))

LPMAX(PRICE(ALLOYS)*X(ALLOYS))

CONSTRAIN(METALS: COMPOSITION(METALS,ALLOYS)*X(ALLOYS)=SUPPLY(METALS))
```

This follows the rules: a) all index ranges in the LPMAX, LPMIN statements are free; b) in the CONSTRAIN statements, the ranges preceeding the colon are fixed; c) in BOUND statements, all ranges are fixed.

The fixed index ranges signal repetition: for each fixed index or index combination, create the indicated statement. By contrast the free index ranges signal summation.

Table names referring to data tables such as COMPOSITION, SUPPLY and PRICE may be preceded by a plus sign "+" or minus sign "-" in LPMAX, LPMIN, CONSTRAIN and BOUND instructions. For instance,

```
LPMIN(-PRICE(ALLOYS)*X(ALLOYS))
```

would be equivalent to the previous LPMAX instruction.

EXAMPLE 2 (Murty [10])

A farmer can mix two different grains in his chicken feed. Given the cost of these grain types and the nutrients they contain, the cost of the grain mix is to be minimized while maintaining specified nutritional levels. More precisely, there are two index ranges:

NUTRIENTS = {STARCH,PROTEIN,VITAMINS}

GRAINS = {1,2},

and three tables:

COST (per kg)

GRAINS	1	$0.60
	2	$0.35

SUPPLY (units per kg)

		NUTRIENTS		
		STARCH	PROTEIN	VITAMINS
GRAINS	1	5	4	2
	2	7	2	1

REQUIREMENT (units per day)

NUTRIENTS	STARCH	8
	PROTEIN	15
	VITAMINS	3

The resulting linear program can be written:

```
UNKNOWN(X(GRAINS)),

LPMIN(COST(GRAINS)*X(GRAINS))

CONSTRAIN(NUTRIENTS: SUPPLY(GRAINS,NUTRIENTS)*X(GRAINS)>=REQUIREMENT(NUTRIENTS))
```

The variables X(1) and X(2) indicate the amounts of each grain type to be fed every day. The nonnegativity of the variables is again assumed automatically. Note that three slack variables are generated automatically by ULP, indicating the amounts of excess supply for each nutrient. The names of the slack variables will be

$(STARCH), $(PROTEIN),$(VITAMIN).

Without the use of tables, the same linear program can be written as follows:

```
UNKNOWN(X1, X2)

LPMIN(0.60*X1 + 0.35*X2)

CONSTRAIN('STARCH':  5*X1 + 7*X2 >= 8)

CONSTRAIN('PROTEIN': 4*X1 + 2*X2 >= 15)

CONSTRAIN('VITAMINS': 2*X1 + X2 >= 3)
```

The answer in any case is: use only grain type 1. In this case, the slack variables will be named

$STARCH, $PROTEIN, $VITAMIN.

The next example demonstrates a somewhat unusual application of the index summation convention.

EXAMPLE 3 (Murty [10])

A steel company has two mines and three steel plants. It wants to minimize the cost of shipping the ore from the mines to the plants. The unit costs of shipping from each mine to any of the plants are as follows:

COSTS ($ per ton)

		PLANT 1	PLANT 2	PLANT 3
MINE	1	9	16	28
	2	14	29	19

The following amounts of ore are available:

SUPPLY (tons)

MINE	1	103
	2	197

On the other hand, the following amounts are required at the plants:

DEMAND (tons)

PLANT	1	71
	2	133
	3	96

Denoting by X11, X12, ... the shipment sizes from mines 1 and 2 to plants 1, 2, and 3, we write

```
UNKNOWN(X11,X12,X13,X21,X22,X23)
LPMIN(9*X11 + 16*X12 + 28*X13 + 14*X21 + 29*X22 + 19*X23)
CONSTRAIN('MINE1':  X11 + X12 + X13 <= 103)
CONSTRAIN('MINE2':  X21 + X22 + X23 <= 197)
CONSTRAIN('PLANT1':  X11 + X21 >= 71)
CONSTRAIN('PLANT2':  X12 + X22 >= 133)
CONSTRAIN('PLANT3':  X13 + X23 >= 96)
```

With index ranges

MINE = {1,2} , PLANT = {1,2,3},

and tables

COST(MINE,PLANT), SUPPLY(MINE), DEMAND(PLANT),

we will write more compactly:

```
UNKNOWN(X(MINE,PLANT))

LPMIN(COST(MINE,PLANT)*X(MINE,PLANT))

CONSTRAIN('MINE'(MINE): X(MINE,PLANT) <= SUPPLY(MINE))

CONSTRAIN('PLANT'(PLANT): X(MINE,PLANT) >= DEMAND(PLANT)
```

The interpretation of the expression

COST(MINE,PLANT)*X(MINE,PLANT)

is straightforward: since it denotes an objective function, all index ranges are free. Consequently, the products "COST*X" are to be formed for all index combinations where the first index is in range MINE and the second index is in range PLANT. All six resulting products,

COST(1,1)*X(1,1) + COST(1,2)*X(1,2) + COST(1,3)*X(1,3)

+ COST(2,1)*X(2,1) + COST(2,2)*X(2,2) + COST(2,3)*X(2,3),

are then to be summed.

The expression

X(MINE,PLANT)

where the index range MINE is fixed and the index range PLANT is free, is understood to translate into the two straight sums:

X(1,1) + X(1,2) + X(1,3)

X(2,1) + X(2,2) + X(2,3).

This straight summation convention is somewhat unusual and extends the convention used in tensor algebra.

As to the use of table-like expressions

'MINE'(MINE), 'PLANT'(PLANT),

in the label portion of the CONSTRAIN statements: they generate index driven names,

MINE(1), MINE(2), PLANT(1), PLANT(2), PLANT(3),

which label the constraints. The need for this is due to the fact that the index ranges MINE and PLANT were defined above in such a fashion that they contained some identical indexes. As a result, they cannot be used by themselves to distinguish between different constraints. The slack variables are correspondingly named

$MINE(1), $MINE(2), $PLANT(1), $PLANT(2), $PLANT(3).

The following example illustrates the use of index ranges within a bound statement. As was mentioned before, such index ranges are considered fixed ranges, creating a bound constraint for each index in that range.

EXAMPLE 4 (Murty [10])

Technological Hotels, Inc., has placed an order for 1000 pounds of ground meat loaf (mixed ground beef, pork, veal) with Quantitative Butchers, Inc., requesting that:

a) Ground beef is to be no less than 400 pounds and no more than 600 pounds.

b) The ground pork must be between 100 and 400 pounds.

c) The ground veal must weigh between 100 and 400 pounds.

d) The weight of ground pork must be no more than one and one half times the weight of veal.

Technological Hotels will pay Quantitative Butchers $1200 for the entire order. The costs, based on percent useable meat and labor, are estimated to run for Quantitive Butchers at the rates of $0.70, $0.60. $0.80 for beef, pork, veal, respectively. Which blend of meats is most profitable to Quantitative Butchers?

The index range

MEAT = {BEEF, PORK, VEAL},

and the variables

X(BEEF), X(PORK), X(VEAL)

are used to denote the amounts used of the respective meats in the shipment and to set up three tables

COST(MEAT), LOW(MEAT), HIGH(MEAT)

with the last two containing the maximum and the minimum amounts permissible. The linear program can then be written:

```
UNKNOWN(X(MEAT))

LPMIN(COST(MEAT)*X(MEAT))

CONSTRAIN('TOTAL': X(MEAT) = 1000)

BOUND(X(MEAT) >= LOW(MEAT))

BOUND(X(MEAT) <= HIGH(MEAT))

CONSTRAIN('PORK RATIO': 1.5*X('VEAL') - X('PORK') >= 0)
```

The answer is: 400 pounds of beef, 360 pounds of pork, 240 pounds of veal.

1.4 Using Several Variable Names in the Objective Function

Several different indexed variable names can be used at the same time in LPMIN and LPMAX statements. It is often handy to work with both indexed and nonindexed variable names simultaneously.

EXAMPLE 5 (Murty [10])

A skyscraper is to be painted. The paint to be used can be obtained by blending four raw paints and two thinners. The assumption is that the physical properties of the paint vary in linear proportion with those of the basic materials used. Data on them is presented in two tables:

PAINT DATA

	PAINTS			
	1	2	3	4
Cost ($/gallon)	9	7	575	4
Viscosity (CP)	900	780	620	375
Vapor pressure (PSI)	0.2	0.4	0.6	0.8
Brilliance content (grams/gallon)	30	20	50	10
Durability content (grams/gallon)	2000	1500	1000	500

THINNER DATA

	THINNERS	
	1	2
Cost ($/gallon)	3	1.85
Viscosity (CP)	2	25
Vapor pressure (PSI)	12.0	8.0

Total viscosity should be above 400, brilliance between 15 and 30, vapor pressure between 2 and 4, durability above 575. Minimize cost.

Consider the index ranges

PROPERTIES = {COST, VISCOSITY, VAPOR PRESSURE,BRILLIANCE,DURABILITY}

NONCHEMICAL = {COST,VISCOSITY,VAPOR PRESSURE}

They are associated with the given tables

PAINT DATA(PROPERTIES,PAINTS)

THINNER DATA(NONCHEMICAL,THINNERS)

where

PAINTS = {1,2,3,4} , THINNERS = {1,2}.

We use these ranges and tables to set up a linear program as follows:

```
UNKNOWN(X(PAINTS),Y(THINNER))
LPMIN(PAINT DATA('COST',PAINTS)*X(PAINTS)
  + THINNER DATA('COST',THINNERS)*Y(THINNERS))
CONSTRAIN('VISCOSITY LOW':
  PAINT DATA('VISCOSITY',PAINTS)*X(PAINTS)
  + THINNER DATA('VISCOSITY',THINNERS)*Y(THINNERS) >= 400)
CONSTRAIN('BRILLIANCE LOW':
  PAINT DATA('BRILLIANCE',PAINTS)*X(PAINTS) >= 15)
CONSTRAIN('BRILLIANCE HIGH':
```

```
    PAINT DATA('BRILLIANCE',PAINTS)*X(PAINTS) <= 30)
  CONSTRAIN('VAPOR PRESSURE LOW':
    PAINT DATA('VAPOR PRESSURE',PAINTS)*X(PAINTS)
    + THINNER DATA('VAPOR PRESSURE',THINNERS)*Y(THINNERS) >= 2)
  CONSTRAIN('VAPOR PRESSURE HIGH':
    PAINT DATA('VAPOR PRESSURE',PAINTS)*X(PAINTS)
    + THINNER DATA('VAPOR PRESSURE',THINNERS)*Y(THINNERS) <= 4)
  CONSTRAIN('DURABILITY LOW':
    PAINT DATA('DURABILITY',PAINTS)*X(PAINTS) >= 575)
  CONSTRAIN('NORMALIZE': X(PAINTS) + Y(THINNERS) = 1)
```

In the above formulation, we have used two names "X" and "Y" to denote the unknown variables. They stand for the percentages used of paints and thinners, respectively, and must therefore add up to 1.

The introduction of additional variables often simplifies problem specification. These variables will typically not appear in the LPMIN or LPMAX statements. Default coefficients of zero are then assumed for such variables in the objective function.

```
UNKNOWN(X(PAINTS),Y(THINNERS),U,V)
LPMIN(PAINT DATA(COST,PAINTS)*X(PAINTS)
  + THINNER DATA(COST,THINNERS)*Y(THINNERS))
CONSTRAIN('VISCOSITY':
  PAINT DATA('VISCOSITY',PAINTS)*X(PAINTS)
  + THINNER DATA('VISCOSITY',THINNERS)*Y(THINNERS) >= 400)
CONSTRAIN('BRILLIANCE':
  PAINT DATA('BRILLIANCE',PAINTS)*X(PAINTS) - U = 0)
CONSTRAIN('VAPOR PRESSURE':
  PAINT DATA('VAPOR PRESSURE',PAINTS)*X(PAINTS)
  + THINNER DATA('VAPOR PRESSURE',THINNERS)*Y(THINNERS) - V = 0)
```

```
CONSTRAIN('DURABILITY':

  PAINT DATA('DURABILITY',PAINTS)*X(PAINTS) >= 575)

BOUND(U >= 15)  BOUND(U <= 30)

BOUND(V >= 2)   BOUND(V <= 4)

CONSTRAIN('NORMALIZE':  X(PAINTS) + Y(THINNERS) = 1)
```

In the above formulation two non-indexed variables have been introduced,

U , V

for viscosity and vapor pressure, respectively. BOUND statements are then used to bound these new variables, and thereby viscosity and vapor pressure, both from above and below. This saves writing down twice the expressions for viscosity and vapor pressure.

1.5 Subranges

We return to EXAMPLE 5 in the previous section. We assume that the data are given in a somewhat different tabular structure, involving the ranges:

PROPERTIES = {VISCOSITY,VAPOR PRESSURE,BRILLIANCE,DURABILITY}

NONCHEMICAL = {VISCOSITY,VAPOR PRESSURE}

PAINTS = {1,2,3,4} , THINNERS = {1,2} ,

and the tables:

PAINT COST(PAINTS), THINNER COST(THINNERS)

PAINT DATA(PROPERTIES,PAINTS), THINNER DATA(NONCHEMICAL,THINNERS).

The construct

NONCHEMICAL()PROPERTIES

is introduced to characterize the intersection of the two ranges, that is, those indexes in PROPERTIES which are also to be found in the index range NONCHEMICAL. We call NONCHEMICAL the "screening range" and PROPERTIES the "leading range". As we run through the indexes in PROPERTIES we either encounter an index which is also valid in NONCHEMICAL, or an index which is not, and which is therefore to be screened out. It follows that a subrange of the screening range is constructed. For this reason, we call a construct of the above kind a "subrange construct". Note that while in the above example the screening range, NONCHEMICAL, is a subset of the leading range, namely PROPERTIES, this need not be the case in general; the leading range may well

be a subset of the screening range or neither range may contain the other.

Whenever the subrange construct is used, the leading range (second position) is thought of as the "driver", that is, the range whose indexes are traversed in the order of that range. The order of the leading range is imposed on the resulting subrange. The screening range then ensures that only indexes which are desired in the given situation are considered. In particular, this holds if the subrange construct is used for the extraction of entries from the table. Here corresponding table entries will be sought out for valid indexes in the screening range or will be ignored (considered zero) otherwise. To ensure the validity of the indexes, we impose the rule: the screening range must coincide with the range associated with the table in the position in which the subrange construct appears.

The use of the subrange construct is to be interpreted in this fashion in the following formulation of EXAMPLE 5:

```
UNKNOWN(X(PAINTS),Y(THINNERS))
LPMIN(PAINT COST(PAINTS)*X(PAINTS)
  + THINNER COST(THINNERS)*Y(THINNERS))
CONSTRAIN(PROPERTIES:
  PAINT DATA(PROPERTIES,PAINTS)*X(PAINTS)
  + THINNER DATA(NONCHEMICAL()PROPERTIES,THINNERS)*Y(THINNERS)
  - Z(PROPERTIES) = 0)
BOUND(Z('VISCOSITY') >= 400)
BOUND(Z('BRILLIANCE') >= 15)
BOUND(Z('BRILLIANCE') <= 30)
BOUND(Z('VAPOR PRESSURE') >= 2)
BOUND(Z('VAPOR PRESSURE') <= 4)
BOUND(Z('DURABILITY') >= 575)
CONSTRAIN('NORMALIZE':  X(PAINTS) + Y(THINNERS) = 1)
```

In the first CONSTRAIN statement, entries from the table "THINNER DATA" are only considered for entries in the subrange NONCHEMICAL( )PROPERTIES.

Since the subrange is in effect formed as the intersection of two ranges with indexes ordered according to their order in the leading range,

NONCHEMICAL ∩ PROPERTIES

might be a more desirable notation. However, the symbol ' ' is not one of the available characters. We chose the two parentheses instead because that notation will be compatible with a generalization of the subrange construct to be described in the next section.

Why not simply write

THINNER DATA(NONCHEMICAL,THINNERS)

in the above run statement? The answer is that then the range NONCHEMICAL would be treated as a free range to be summed over rather than a fixed range, the latter being the intent in the above example. Using the fixed range PROPERTIES as the leading range indicates that the range NONCHEMICAL is also fixed and it links these indexes with their associated constraints which are arranged by indexes from the range PROPERTIES. Another possibility is to write

THINNER DATA(PROPERTIES,THINNERS),

with the interpretation that indexes in PROPERTIES which do not belong to the range NONCHEMICAL are simply skipped, since the table THINNER DATA is only defined for indexes in the range NONCHEMICAL. Serious consideration should be given to permitting this notation as an abbreviation.

The subrange construct can be used for the purpose of reordering an index range. This is demonstrated in the following.

EXAMPLE 6: (Murty [10])

A forestry company has four sites on which it grows trees. It is considering four species of trees: pines, spruces, walnuts and oaks. There are four sites whose available areas are given below:

		AREA (kiloacres)
SITES	A	1500
	B	1700
	C	900
	D	600

There are minimum required expected annual yields specified which will have to be produced on the combined available area:

MINIMAL REQUIRED YIELD
(cubic meters per kiloacre)

TREES	WALNUT	4.8
	OAK	3.5
	PINE	22.5
	SPRUCE	9

The following tables give the expected annual yield and revenue:

YIELD
(cubic meters per kiloacre)

		SPECIES			
		PINE	SPRUCE	WALNUT	OAK
SITES	A	17	14	10	9
	B	15	16	12	11
	C	13	12	14	8
	D	10	11	8	6

REVENUE
(kilo$ per kiloacre)

		SPECIES			
		PINE	SPRUCE	WALNUT	OAK
SITES	A	16	12	20	18
	B	14	13	24	20
	C	17	10	28	20
	D	12	11	18	17

How much area should be devoted to growing the various species in the various sites?

The data are arranged in the following tables:

AREA(SITES)

MINIMAL REQUIRED YIELD(TREES)

YIELD(SPECIES,SITES)

REVENUE(SPECIES,SITES)

where

SITES = {A,B,C,D}

TREES = {WALNUT,OAK,PINE,SPRUCE}

SPECIES = {PINE,SPRUCE,WALNUT,OAK} .

The index ranges "TREES" and "SPECIES" are identical except for the order of the indexes in them. This suggests the following formulation:

```
UNKNOWN(X(SPECIES,SITES))
LPMAX(REVENUE(SPECIES,SITES)*X(SPECIES,SITES))
CONSTRAIN(SPECIES:
  YIELD(SPECIES,SITES)*X(SPECIES,SITES)
  >= MINIMAL REQUIRED YIELD(TREES()SPECIES))
CONSTRAIN(SITES: X(SPECIES,SITES) <= AREA(SITES))
```

1.6 Index Maps

The subranges introduced in the previous sections are special cases of a more general construct whose purpose it is to assign indexes in a specified image range to some of the indexes of a specified parent range.

In order to define such a "range transformation," we introduce "index maps". Such an index map assigns to each index in the "domain" an index in the "image range". While each index in the domain must be assigned an image, not every index in the image range needs to be the image of some index in the domain. Different domain indexes may have the same image. Domain and image range are part of the specification of each index map. For defining an index map, we use the key words

LET MAP.

Index maps, just like index ranges, have alphanumeric names which start with a letter. Schematically, the definition of index maps takes the form:

```
LET MAP (image range name (map name) domain range name:
        'index name in image range' = 'index name in domain range',
        'index name in image range' = 'index name in domain range',
            ...                              ...
        'index name in image range' = 'index name in domain range',
        'index name in image range' = 'index name in domain range')
```

We realize that the LET MAP construct in its present form has a drawback in that the quantity to be defined, namely the index map, is nested in the middle of a string and therefore not readily visualized. We feel, however, that the similarity of the definition string with the string in which the index is actually used provides a compensating advantage. Also the image range and domain of the map can be immediately identified.

The LET MAP statement is a first example of a "specification instruction". Such instructions define a quantity, in this case an index map without, however, including this quantity in the database. The specification thus remains valid only within a particular set of run instructions. Other specification instructions for defining tables and ranges will be discussed in Chapter 2.

Index maps define range transformations when used as follows:

image range name (map name) domain range name.

Suppose, for example, that in EXAMPLE 6 of the previous section, the table

MINIMAL REQUIRED YIELD(TREES)

would be indexed in digits rather than names of tree species. In other words, the index range TREES, which previously contained names, is now assumed to be

TREES = {1,2,3,4},

whereas we still have

SPECIES = {PINE, SPRUCE, WALNUT, OAK}

in tables YIELD and REVENUE.

In order to make the tables compatible, we introduce an index map DIGITS. We can then write:

```
UNKNOWN(X(SPECIES,SITES))

LPMAX(REVENUE(SPECIES,SITES)*X(SPECIES,SITES))

CONSTRAIN(SITES: X(SPECIES,SITES) <= AREA(SITES))

LET MAP(TREES(DIGITS)SPECIES:

   '1'='WALNUT', '2'='OAK', '3'='PINE', '4'='SPRUCE')

CONSTRAIN(SPECIES:

   YIELD(SPECIES,SITES)*X(SPECIES,SITES)

   >= MINIMAL REQUIRED YIELD(TREES(DIGITS)SPECIES))
```

The construct

TREES(DIGITS)SPECIES

is a generalization of the subrange construct introduced in the previous section, and is therefore called a "generalized subrange construct". The ranges TREES and SPECIES are screening and leading ranges, respectively. The index map DIGITS, as specified by the LET MAP instruction, assigns to every index of range SPECIES its counterpart in DIGITS. The latter index is then screened as to its occurrence in the range TREES.

In this case, the image and domain ranges of the index map used in the generalized subrange construct coincide with the screening and leading ranges, respectively. In general, this need not be the case. Consider

S(M)T

with ranges A and B being the image and domain ranges of the index map M. This construct is to be interpreted as follows: For each index in the leading range T, determine whether it lies in the domain B of map M. If so, substitute the corresponding image index in A. If the leading index does not lie in the domain B, then do not transform the index. In both cases, transformation or no transformation, determine whether the resulting index occurs in the screening range S, in which case the index is accepted as a member of the generalized subrange. If not, then skip the leading index in question.

There is a natural alternative to this interpretation, namely to skip leading indexes not in the domain B of the map M. An advantage of that alternative interpretation would be that the construct becomes a straightforward composition of the subrange constructs S( )A and B( )T with the index map M. The advantage of the chosen interpretation is that in this case the generalized subrange construct with the "empty map" ϕ achieves the same effect as the nongeneralized subrange construct: $S(\phi)A = S()A$. This represents more than a formal advantage as we will explain below.

First we introduce a generalized subrange construct in which the index map definition rather than the index map name is employed. For example, the above run statment can be written as follows:

```
UNKNOWN(X(SPECIES,SITES))

LPMAX(REVENUE(SPECIES,SITES)*X(SPECIES,SITES))

CONSTRAIN(SITES: X(SPECIES,SITES) <= AREA(SITES))

CONSTRAIN(SPECIES:

     YIELD(SPECIES,SITES)*X(SPECIES,SITES)

     >= MINIMAL REQUIRED YIELD(TREES

     ('1'='WALNUT', '2'='OAK', '3'='PINE', '4'='SPRUCE')SPECIES))
```

Related ranges frequently differ by only a few indexes, perhaps due to an ambiguity in spelling. Consider, for instance, the ranges

$$S = \{A1,A11,A12,A21,A22,A23,A31,A32,A33,A34\}$$

$$T = \{A00,A01,\ldots,A99\} .$$

Then writing

```
S('A'='A01')T
```

is a convenient way to use the generalized subrange construct to compensate for the spelling discrepancy concerning the indexes A1 and A01. Note that only the exception is specified: the identity mapping is assumed as the default for all other indexes. This demonstrates the usefulness of this default convention.

However, there are also circumstances under which the user would like to enforce the skipping of a particular index in the sequencing range of a subrange construct. Therefore, we introduce the "non-index" symbol, which consists of two successive single quotes, possibly separated by a string of blank characters. Within the index map specification, writing

```
' ' = 'I'
```

for some index I in the index map domain will indicate that the index I does not have an image. In the context of a generalized map construct, it will cause the leading index I to be skipped no matter what screening range is used. The non-index feature is available for index map definitions as well as for generalized subrange constructs.

The non-index feature will be useful for discussions of the "NEXT" operator in the following section.

1.7 Time-periods in Linear Programs

In many applications, index ranges consist of indexes which denote time periods. Linear programs based on such time periods often relate information of the next time period to the present one. It will then be useful to associate with a given index range, say,

$$\text{TIME} = \{T1,T2,T3,T4,T5\}$$

a generalized subrange construct which assigns to each index its successor in the index range

$$T1 \to T2,\ T2 \to T3,\ \ldots,\ T4 \to T5$$

with no assignment to the last index, T5. For this purpose, we introduce the operator NEXT and write

$$\text{NEXT(TIME)}.$$

To arrive at an equivalent generalized subrange construction, we would have to write, using the nonindex symbol "'",

$$\text{TIME('T2'='T1','T3'='T2', 'T4'='T3','T5'='T4', ' '='T5')TIME},$$

or introduce an equivalent map using a similarly lengthy map definition. The

$$\text{PREVIOUS(TIME)}$$

operator is analogously defined for the purpose of "lagging".

The operators NEXT and PREVIOUS will also apply to tuples of ranges. For example, let

$$\text{YEARS} = \{80,81\}$$

$$\text{QUARTERS} = \{Q1,Q2,Q3,Q4\}.$$

Then

$$\text{NEXT(YEARS,QUARTERS)}$$

consists of the following assignments in "odometer" or "lexicographic" order (See Section 2.3):

$$\text{NEXT}(80,Q1) = (80,Q2),\ \ldots,\ \text{NEXT}(80,Q4) = (81,Q1),\ \ldots,\ \text{NEXT}(81,Q3) = (81,Q4).$$

This extends the concept of index maps. We illustrate the use of the NEXT operator in the following example

EXAMPLE 7:

A manufacturing firm agrees to produce and deliver specified quantities of products A, B, C at the end of calendar years 1980 and 1981:

CONTRACT

	80	81
A	2000	3000
B	3500	5000
C	5000	6000

The firm draws up a manufacturing plan, specifying for each quarter the number of units to be produced, labor employed and raw materials purchased. There are three kinds of raw materials: plastic, wire, and coating. Unit raw materials needed per unit product are:

RAW MATERIALS

	A	B	C
PLASTIC	.513	.197	.114
WIRE	.032	.021	.019
COATING	.013	.011	.006

Besides raw materials there are labor and machinery requirements. The latter represent the time for tying up manufacturing machinery.

WORK

	A	B	C
LABOR	.0022	.0024	.0015
MACHINERY	.0010	.0010	.0005

The costs of raw materials as well as the above work requirements are time dependent:

REQUIREMENT COSTS

		PLASTIC	WIRE	COATING	LABOR	MACHINERY
80	Q1	1.2	2.0	9.5	1000	1100
	Q2	1.3	2.1	10.1	1050	1000
	Q3	1.3	2.1	10.1	1100	1250
	Q4	1.0	1.9	10.1	1150	1000
81	Q1	1.3	2.1	9.9	1050	1150
	Q2	1.4	2.2	10.5	1100	1050
	Q3	1.4	2.2	10.5	1150	1300
	Q4	1.1	2.0	10.5	1200	1050

Raw material requirements differ from labor and machinery requirements in that the former can be utilized during later time-periods, at the expense, however, of quarterly inventory costs. We assume that these quarterly inventory costs are time independent.

MATERIAL INVENTORY COST

PLASTIC	.06
WIRE	.04
COATING	.20

Finished products, when stored, incur similar inventory costs:

PRODUCT INVENTORY COST

A	.11
B	.09
C	.05

Continuity of labor over project time has to be ensured to avoid unnecessary learning expenses. We stipulate that, after each quarter in year 80, labor used shall not decrease and, after each quarter in year year 81, labor used shall not increase. Thus we have the labor carry-over conditions:

$$\text{Labor}(80,Q1) < \text{Labor}(80,Q2)$$

$$\text{Labor}(80,Q2) < \text{Labor}(80,Q3)$$

$$\text{Labor}(80,Q3) < \text{Labor}(80,Q4)$$

$$\text{Labor}(80,Q4) < \text{Labor}(81,Q1)$$

$$\text{Labor}(81,Q1) > \text{Labor}(81,Q2)$$

$$\text{Labor}(81,Q2) > \text{Labor}(81,Q3)$$

$$\text{Labor}(81,Q3) > \text{Labor}(81,Q4)$$

Company planning also determines minimum and maximum utilization levels for machinery:

UTILIZATION

		MINIMUM	MAXIMUM
80	Q1	0	40
	Q2	20	50
	Q3	10	40
	Q4	5	30
81	Q1	5	40
	Q2	10	50
	Q3	0	60
	Q4	0	60

Finally, a policy decision states that a minimim portion of plastic be purchased in the first time period 80,Q1 as a safeguard against possible supply difficulties which might develop later in the year 80:

$$\text{Plastic purchased } (80,Q1) > 800.$$

Determine an optimal production plan.

We assume that the data of the above problem are contained in the following index ranges and tables:

```
PRODUCTS = {A,B,C}

MATERIALS = {PLASTIC,WIRE,COATING}

ITEMS = {PLASTIC,WIRE,COATING,A,B,C}

QUARTERS = {Q1,Q2,Q3,Q4}

YEARS = {80,81}

POTENTIALS = {LABOR,MACHINERY}

RESOURCES = {PLASTIC,WIRE,COATING,LABOR,MACHINERY}

LEVELS = {MINIMUM,MAXIMUM}

CONTRACT(PRODUCTS,YEARS)

RAW MATERIALS(MATERIALS,PRODUCTS)

WORK(POTENTIALS,PRODUCTS)

COSTS(RESOURCES,YEARS,QUARTERS)

MATERIAL INVENTORY COST(MATERIALS)

PRODUCT INVENTORY COST(PRODUCTS)

UTILIZATION(LEVELS,YEARS,QUARTERS)
```

Based on this information, we formulate a linear program as follows:

```
UNKNOWN(PURCHASED(MATERIALS,YEARS,QUARTERS),
        AVAILABLE(POTENTIALS,YEARS,QUARTERS),
        INVENTORY(ITEMS,YEARS,QUARTERS),
        PRODUCED(PRODUCTS,YEARS,QUARTERS),
        DELIVERED(PRODUCTS,YEARS,'Q4'))

 LPMIN(COSTS(RESOURCES()MATERIALS,YEARS,QUARTERS)
      *PURCHASED(MATERIALS,YEARS,QUARTERS)
     + COSTS(RESOURCES()POTENTIALS,YEARS,QUARTERS)
      *AVAILABLE(POTENTIALS,YEARS,QUARTERS)
```

```
      + MATERIAL INVENTORY COST(MATERIALS)
       *INVENTORY(ITEMS()MATERIALS,YEARS,QUARTERS)
      + PRODUCT INVENTORY COST(PRODUCTS)
       *INVENTORY(ITEMS()PRODUCTS,YEARS,QUARTERS))

 CONSTRAIN('PRODUCT BALANCE'(PRODUCTS,NEXT(YEARS,QUARTERS)):
            PRODUCED(PRODUCTS,YEARS,QUARTERS)
          + INVENTORY(ITEMS()PRODUCTS,YEARS,QUARTERS)
          - INVENTORY(ITEMS()PRODUCTS,NEXT(YEARS,QUARTERS))
          - DELIVERED(PRODUCTS,YEARS,'Q4'()QUARTERS) = 0)

 BOUND(DELIVERED(PRODUCTS,YEARS,'Q4') >=
        CONTRACT(PRODUCTS,YEARS))

 CONSTRAIN('MATERIAL BALANCE'(MATERIALS,NEXT(YEARS,QUARTERS)):
            PURCHASED(MATERIALS,YEARS,QUARTERS)
          + INVENTORY(ITEMS()MATERIALS,YEARS,QUARTERS)
          - INVENTORY(ITEMS()MATERIALS,NEXT(YEARS,QUARTERS)
          - RAW MATERIALS(MATERIALS,PRODUCTS)
          * PRODUCED(PRODUCTS,YEARS,QUARTERS) = 0)

 BOUND(PURCHASED('PLASTIC','80','Q1') >= 800)
CONSTRAIN('RESERVE'(POTENTIALS,YEARS,QUARTERS):
           AVAILABLE(POTENTIALS,YEARS,QUARTERS)
         - WORK(POTENTIALS,PRODUCTS)
        * PRODUCED(PRODUCTS,YEARS,QUARTERS) >= 0)
BOUND(AVAILABLE('MACHINERY',YEARS,QUARTERS) <=
      UTILIZATION('MAXIMUM',YEARS,QUARTERS))
BOUND(AVAILABLE('MACHINERY',YEARS,QUARTERS) >=
      UTILIZATION('MINIMUM',YEARS,QUARTERS))
```

```
CONSTRAIN('CARRYOVER'('80',NEXT(QUARTERS)):
          AVAILABLE('LABOR','80',NEXT(QUARTERS))
        - AVAILABLE('LABOR','80',QUARTERS) >= 0)
CONSTRAIN('CARRYOVER'('81','Q1'):
          AVAILABLE('LABOR','81','Q1')
        - AVAILABLE('LABOR','80','84') >= 0)
CONSTRAIN('CARRYOVER'('81',NEXT(QUARTERS)):
          AVAILABLE('LABOR','81',NEXT(QUARTERS))
        - AVAILABLE('LABOR','81',QUARTERS) <= 0)
```

The constraint name specification

```
'CARRYOVER'('81',NEXT(QUARTERS))
```

will produce, as the four indexes of the range QUARTERS are being stepped through, only the following three names:

```
CARRYOVER(81,Q2)
CARRYOVER(81,Q3)
CARRYOVER(81,Q4)
```

Thus only three constraints are being generated even though the range QUARTERS contains four indexes. Similarly, the notation

```
'PRODUCT BALANCE'(PRODUCTS, NEXT(YEARS, QUARTERS))
```

causes the constraint to be skipped for the last quarter of year 81. The construct

```
'Q4'()QUARTERS
```

causes terms to be suppressed for all quarters other than Q4.

1.8 Network Flow Problems

A "network" or "directed graph" consists of a set of "nodes", a set of "arcs", and specifications which associate with each arc two different nodes: the "origin" and the "destination" of the arc. Figure 1 illustrates such a network. The nodes are represented by small circles and the arcs by connecting lines.

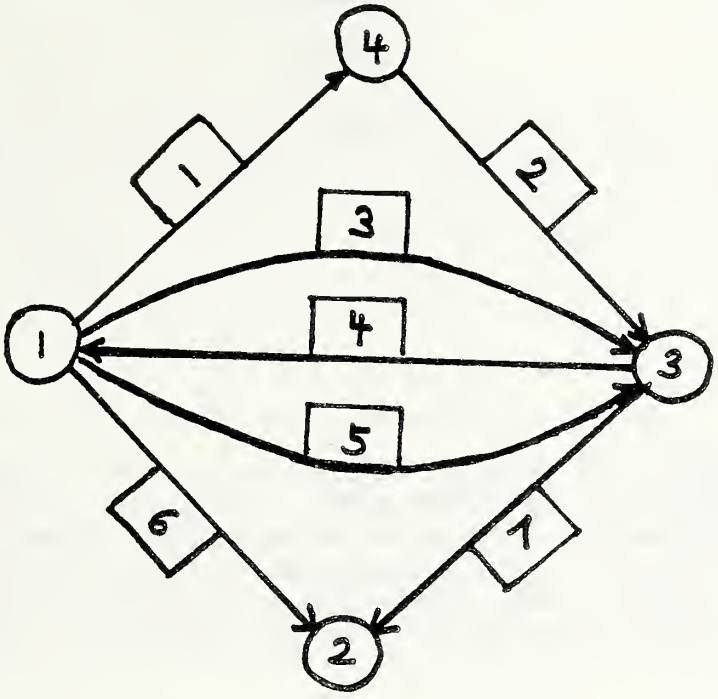

Figure 1

The numbers in the circles identify the nodes. Similarly, arc numbers are written next to their lines.

A network is characterized by its "adjacency matrix" A, whose rows correspond to the nodes and whose columns correspond to the arcs in the network. The elements of this matrix are defined as:

$$A(n,a) = \begin{cases} -1 & \text{if node } n = \text{origin of arc } a \\ +1 & \text{if node } n = \text{destination of arc } a \\ 0 & \text{otherwise.} \end{cases}$$

EXAMPLE 8:

With the ranges

$$\text{NODES} = \{1, 2, 3, 4\}$$

$$\text{ARCS} = \{1, 2, 3, 4, 5, 6, 7\},$$

the adjacency matrix for the network in Figure 1 may be represented by the 2-dimensional table:

NETWORK

ARCS

NODES	1	2	3	4	5	6	7
1	-1	0	-1	+1	-1	-1	0
2	0	0	0	0	0	+1	+1
3	0	+1	+1	-1	+1	0	-1
4	+1	-1	0	0	0	0	0

In what follows, we describe an important class of linear programs, each of which is associated with a particular network such as the one in Figure 1. These linear programs are based on a notion of "flow" along arcs, which can be thought of as a flow of a single homogeneous material subject to the law of conservation of matter. Each node may be a "source" or a "sink" of given strength. The law of conservation then requires that at each node the condition

$$\left[\begin{array}{l}\text{sum of flows}\\ \text{on incoming arcs}\end{array}\right] - \left[\begin{array}{l}\text{sum of flows}\\ \text{on outgoing arcs}\end{array}\right] = \left[\begin{array}{l} - \left[\begin{array}{l}\text{source strength}\\ \text{at the node}\end{array}\right] \\ \\ + \left[\begin{array}{l}\text{sink strength}\\ \text{at the node}\end{array}\right]\end{array}\right]$$

hold depending on whether the node in question is a source or a sink. The objective is to route the flow through the network in the least expensive way, given, for each arc, a particular cost for routing a unit of flow along this arc, as well as upper and lower bounds on the flow which limit the capacity of the arc. This is the well-known "minimum cost flow problem". We will describe it in terms of our specification language for the particular network in Figure 1.

Additional information is provided by the tables

SOURCE

NODES	
1	5
2	0
3	-5
4	0

COST

ARCS	
1	1
2	-2
3	2
4	0
5	1
6	1
7	3

LOW

ARCS	
1	0
2	-5
3	1
4	-1
5	0
6	0
7	0

HIGH

ARCS	
1	3
2	5
3	2
4	1
5	1
6	2
7	3

The first array, SOURCE, associates with each node the amount of flow originating at this node. Negative values indicate that the node is acting as a sink. It follows from the conservation law that the sum of all entries in SOURCE must vanish. The array COST specifies the costs per unit of flow along each arc. Negative costs indicate revenues. Finally, the arrays HIGH and LOW contain the capacity limits. The minimum cost flow problem can now be written as follows:

```
UNKNOWN(FLOW(ARCS))

LPMIN(COST(ARCS)*FLOW(ARCS))

CONSTRAIN(NODES: NETWORK(NODES,ARCS)*FLOW(ARCS) = SOURCE(NODES))

BOUND(FLOW(ARCS) > LOW(ARCS))

BOUND(FLOW(ARCS) < HIGH(ARCS))
```

The constraint equations are precisely the conditions that flow be conserved at each node. It should be mentioned that at least one of the constraint equations is superfluous. Indeed, every column in the adjacency matrix contains by definition exactly two nonzero entries with values +1 and -1. The sum of all rows of the adjacency matrix therefore vanishes, which means that the constraint equations are linearly dependent. In other words, at least one constraint equation can be expressed as a linear combination of the remaining constraint equations, and is therefore satisfied whenever the latter constraints are satisfied. If the network is "connected" (in some fashion not to be discussed here), then deleting an arbitrary single row will remove the linear dependence. For this reason the user may want to redefine the range NODES and the tables NETWORK and SOURCE to avoid linear dependence in the above problem specification. In general, this should not be necessary, as the linear programming algorithm is capable of detecting linear dependence and, consequently, superfluity of constraint equations, and can therefore be expected to take suitable default measures.

It must be pointed out that special methods for solving minimum cost flow problems in networks exist that are much more efficient than general linear programming algorithms. Well-crafted general methods may be able to take some advantage of the general structure, but are not yet competitive with special network methods. Why then bother with network problems in this context? The answer is that many linear programming problems have network problems imbedded, that is, a portion of their variables may be flows in a network and, consequently, satisfy the network constraints in addition, perhaps, to other constraints. Such problems are no longer the kind of network problems to which the special methods apply.

Unfortunately, the adjacency matrices of networks tend to be very large and difficult to prepare for input into the database. As a saving grace, the adjacency matrices, especially the big ones, are usually "sparse", that is, the number of their nonzero elements is small compared to the total number of elements. The problem of specifying and storing adjacency matrices may thus be alleviated by providing special features for their input into the database and by permitting condensed (nonzero elements only) representations of tables within the database. Such features, however, are not part of the problem specification process; they belong in the realm of generating the database.

In addition to the above measures, however, special capability to define network problems within the problem specification process appears to be useful. In particular, two index maps (see Section 1.6), one relating origins to arcs and one relating destinations to arcs, may be specified. In the case of the network in Figure 1 and again with the ranges

NODES = {1, 2, 3, 4}

ARCS = {1, 2, 3, 4, 5, 6, 7} ,

one might thus introduce the index maps

```
LET MAP(NODES(ORIGIN)ARCS: '1'='1', '4'='2', '1'='3', '3'='4', '1'='5',
                           '1'='6', '3'='7')
```

```
LET MAP(NODES(DESTINATION)ARCS: '4'='1', '3'='2', '3'='3', '1'='4', '3'='5',
                                '2'='6', '2'='7')
```

The first map associates the origin NODES, represented by their corresponding indexes in range NODES, with the arcs represented by indexes in range ARCS. The former indexes are to the left, the latter to the right of the corresponding equal sign. The second map associates the destination nodes with the arcs in an analogous fashion.

We then propose to introduce the operator word

```
NETWORK,
```

which, when applied to the two mappings defined by the MAP statement, creates the corresponding adjacency table:

```
NETWORK(ORIGIN,DESTINATION)(NODES,ARCS) .
```

The names ORIGIN and DESTINATION are the names of the maps with domain range ARCS and image range NODES. These maps are to spell out the adjacency relations between nodes and arcs in the network. The names are of course arbitrary as long as the ranges of the so defined adjacency table coincide with the map ranges. Thus, in the following problem specification we have chosen the shorter names "O" and "D" instead of "ORIGIN" and "DESTINATION", and the name N and A for NODES and ARCS:

```
UNKNOWN(X(A))
LET MAP(N(O)A: '1'='1','4'='2','1'='3','3'='4','1'='5','1'='6','3'='7')
LET MAP(N(D)A: '4'='1','3'='2','3'='3','1'='4','3'='5','2'='6','2'='7')
CONSTRAIN(N: NETWORK(O,D)(N,A)*X(A)=S(N))
BOUND(X(A)<H(A)); BOUND(X(A)>L(A))
LPMIN(C(A)*X(A))
```

Here

```
S(N), C(A), H(A), L(A)
```

represent sources, costs, and capacity limits, whereas the notation

```
X(A)
```

has been chosen for the flow, which is to be optimized.

The above considerations suggest that index maps should join tables and ranges as a class of items to be stored on the database. Then the origin and destination maps could be input directly into the database at the outset, obviating the need for the LET MAP statements. In some cases, such maps might provide a more convenient way for presenting network information than adjacency matrices.

1.9 Index Range Sequences and Their Inverses

In this section, we will discuss yet another feature that can be employed for the specification of networks, but which is more generally useful for the formulation of linear programs of a combinatorial nature. This feature is an additional construct, namely, "index range sequences". The definition of such an index range sequence is based on the selection of two index ranges, the "domain range" or "domain" and the "universal range". An index range sequence then assigns to each index in the domain range a subrange of the universal range.

Consider the ranges

$$\text{NODES} = \{1,2,3,4\}$$

$$\text{ARCS} = \{1,2,3,4,5,6,7\}$$

in EXAMPLE 8. By assigning to the indexes in NODES the following subranges of ARCS:

$$1 \rightarrow \{1,3,5,6\} = F(1)$$

$$2 \rightarrow \phi = F(2) \text{ (the "empty" subrange)}$$

$$3 \rightarrow \{4,7\} = F(3)$$

$$4 \rightarrow \{2\} = F(4),$$

we define an index range sequence F with NODES as domain range and ARCS as universal range. Each of the assigned subranges of ARCS describes what is commonly called the "forward star" of a node in the network, namely, the set of arcs which originate at this node. Similarly, one might assign the following subranges of ARCS

$$1 \rightarrow \{4\} = B(1)$$

$$2 \rightarrow \{6,7\} = B(2)$$

$$3 \rightarrow \{2,3,5\} = B(3)$$

$$4 \rightarrow \{1\} = B(4) .$$

This again defines an index range sequence B with universal range ARCS and sequenced by NODES. It describes the "backward stars" in a network. Forward and backward stars together define the network. Using these index range sequences, we can write the minimum cost flow problem of EXAMPLE 8 as follows:

```
UNKNOWN(X(ARCS))

LPMIN(COST(ARCS)*X(ARCS))

CONSTRAIN(NODES: + X(ARCS()F(NODES))

                 - X(ARCS()B(NODES))=SOURCE(NODES))

BOUND(X(ARCS)<=HIGH(ARCS))

BOUND(X(ARCS)>=LOW(ARCS))
```

For each index n in NODES, F(n) and B(n) denote an index range. Since these ranges are not mentioned before the colon ":" in the CONSTRAIN statement, they are free index ranges (see Section 1.3) to be summed over. Summation over F(n) yields the total outflow from node n (along arcs), whereas summation over B(n) yields the total inflow (along arcs) into node n. The resulting constraint equations are identical to those derived from the adjacency matrix.

The above problem specification is based on the assumption that the two index range sequences are contained in the database. That means that yet another construct has been added to the list of constructs, such as tables, index ranges and maps, which the database is expected to handle. An alternative option is to go the route of a specification statement. Such a statement might take the form

```
LET RANGE SEQUENCE(ARCS()F(NODES):

        F('1') = '1','3','5','6',

        F('2') = ,

        F('3') = '4','7',

        F('4') = '2')
```

Note, that F('2') is assigned the empty range. The same action will be taken by default, if F('2') is not mentioned in the above instruction.

Of more interest is the question whether index range sequences can be derived from other information. Indeed, we observe that the forward star assignment F is the "inverse" of the origin map O in a network (see Section 1.8) in the following sense: if the index map O is applied to any element in F(n), then node n is reproduced:

$$n = O(F(n)).$$

Conversely, every arc lies in the forward star of its origin:

$$a \in F(O(a)).$$

The backward star assignment B is analogously the inverse of the destination map D. Given the index maps O and D, defined either by a LET MAP statement or contained in the database, the following statement using the operator word

INVERSE,

could be employed to define the index sequences F and B:

LET RANGE SEQUENCE(ARCS()F(NODES): INVERSE(O))

LET RANGE SEQUENCE(ARCS()B(NODES): INVERSE(D)).

In a similar vein, we define the inverses of index range sequences. If F is any index range sequence with universal range, say, ARCS and domain NODES, then the inverse F^{-1} of F is an index range sequence with universal range NODES and domain range ARCS defined by

$$F^{-1}(a) = \{n \in N: a \sim F(n)\}.$$

In other words, $n \varepsilon F^{-1}(a)$ if and only if $a \varepsilon F(n)$. It follows that

$$(F^{-1})^{-1}=F.$$

Consider the inverse of the index map O. This index map clearly corresponds to the range sequence which assigns to each index in the domain range a sequence consisting of a single index in the image range. The domain thus becomes the sequencing range and the image range the universal range. The inverse F of the index map O is identical to the inverse of its corresponding index range sequence, and the latter is reproduced as the inverse of F.

Many combinatorial configurations can be characterized by index range sequences. Consider for example an "undirected network" or "graph" as shown in Figure 2.

Again there are nodes linked by arcs. However, among the two "ends" of an arc, no distinction is made between origin and destination. Obviously, the graph in Figure 2 is defined by the ranges:

NODES = {1,2,3,4,5,6,7,8,9,10}

ARCS = {A13,A14,A16,A24,A25,A27,A35,A38,A49,A50,A67,A60,A78,A89,A90}

and the index range sequence ENDS:

LET RANGE SEQUENCE(ARCS()ENDS(NODES):

'A13'='1','3', 'A14'='1','4', 'A16'='1','6', 'A24'='2','4', 'A25'='2','5',

'A27'='2','7', 'A35'='3','5', 'A38'='3','8', 'A49'='4','9', 'A50'='5','10',

'A67'='6','7', 'A60'='6','10','A78'='7','8', 'A89'='8','9', 'A90'='9','10').

The inverse of the index range sequence ENDS is that index range sequence which assigns to each node its "star" of all adjacent arcs.

A graph can also be defined by its adjacency matrix:

$$A(n,a) = \begin{cases} 1 \text{ if node n is an end of arc a} \\ 0 \text{ otherwise.} \end{cases}$$

For the graph in Figure 2, it can be written in tabular form:

	A13	A14	A16	A24	A25	A27	A35	A38	A49	A50	A67	A60	A78	A89	A90
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0
5	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0
6	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0
7	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0
9	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1
10	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

This representation has the same disadvantages as the representation of a directed network in terms of its adjacency matrix. The index range sequence ENDS, of course, indicates for each column the two rows with nonzero entries, which is precisely the information needed for a compressed representation of the above table. The inverse $ENDS^{-1}$ accomplishes the same for the rows of the table.

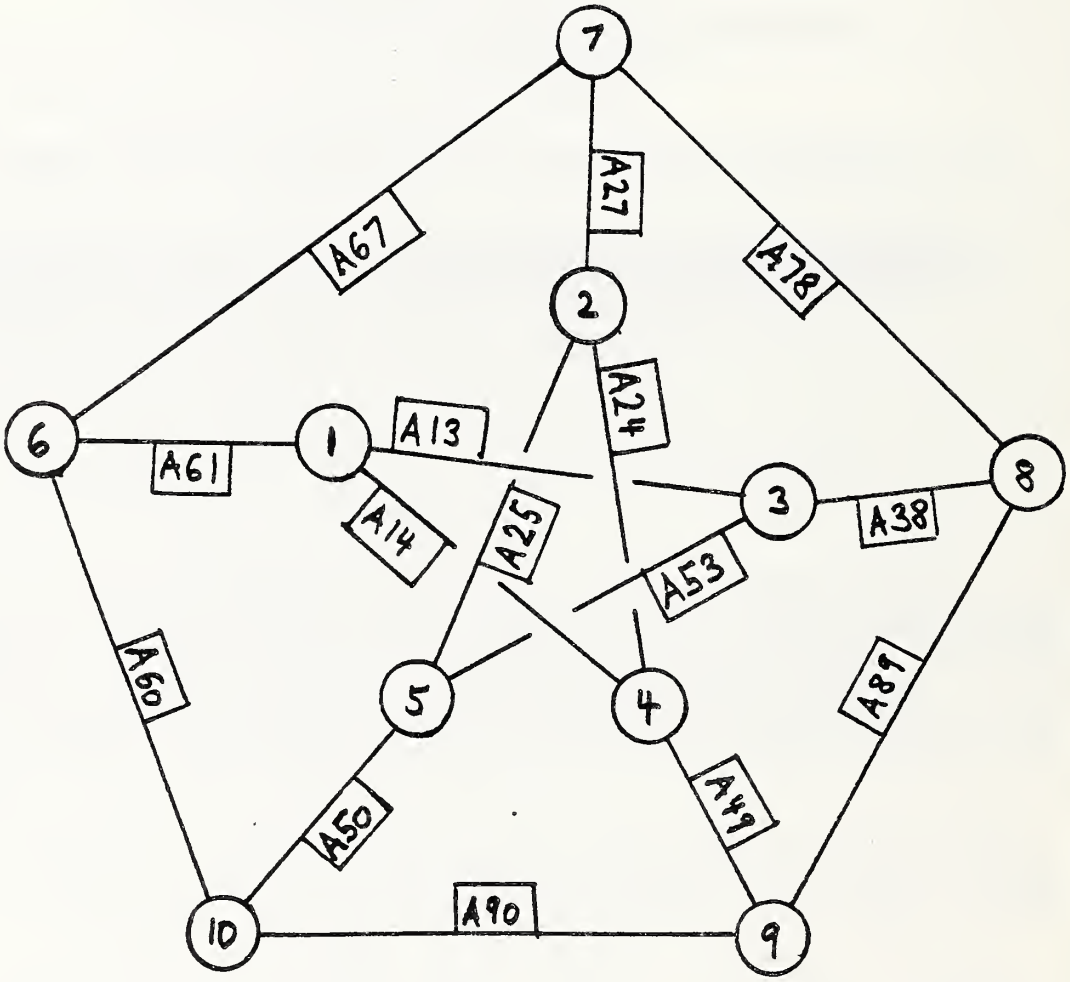
7
A67
A27
A78
2
6
1
A13
A24
8
A61
3
A38
A14
A25
A53
A60
5
4
A89
A49
A50
10
A90
9

FIGURE 2

EXAMPLE 9

A matching in a graph is a set M of arcs such that no two arcs in M have an end in common. We consider the problem of finding a matching of maximum cardinality in the graph of Figure 2. To this end, we introduce unknowns X(ARCS) bounded between 0 and 1. A zero value of X indicates that the arc does not belong to the matching M, whereas a value of 1 indicates that it does. Assuming that the database contains the index range sequence ENDS, we can formulate the linear program:

```
UNKNOWN(X(ARCS))

LPMAX(X(ARCS))

LET RANGE SEQUENCE(ARCS()STAR(NODES):=INVERSE(ENDS))

CONSTRAIN(NODES:X(ARCS()STAR(NODES))=1)

BOUND(X(ARCS)<=1); BOUND(X(ARCS)>=0)
```

This program is known as the "linear programming relaxation" of the maximum cardinality matching problem. Indeed, if all X values are integers for an optimal solution of the above linear program, then the values of X characterize a matching M. This matching will be maximum. In general, however, the linear programming solution will yield some fractional values for X, which thus cannot be associated with a matching. In this case, integer programming techniques are called for. The solution of the linear programming relaxation, however, often plays an important role as part of such techniques.

1.10 Conventions for Deleting Quotes

Putting quotes around every index name in, say, a LET MAP instruction or a generalized subrange construct as described in Section 1.6, may become tiresome. It also takes up space and at times will clutter the visual image. For these reasons, we propose conventions under which quotes may be deleted. For instance, instead of

```
LET MAP(TREES(DIGITS)SPECIES: '1'='WALNUT','2'='OAK','1'='PINE','4' = 'SPRUCE')
```

it appears desirable to write, say,

```
LET MAP(TREES(DIGITS)SPECIES: '1 = WALNUT, 2=OAK, 3=PINE, 4=SPRUCE')
```

Similarly, the instruction

```
LET RANGE SEQUENCE (ARCS()ENDS(NODES)):
'A13'='1','3', 'A14'='1''4', 'A16'='1','6', 'A24'='2','4', 'A25'='2','5',
'A27'='2','7', 'A35'='3','5', 'A38'='3','8', 'A49'='4','9', 'A50'='5','10',
'A67'='6','7', 'A60'='6','10', 'A78'='7','8', 'A89'='8','9', 'A90'='9','10')
```

becomes more readable if only few quotes are retained:

```
LET RANGE SEQUENCE (ARCS()ENDS(NODES)):
'A13=1,3, A14=1,4, A16=1,6, A24=2,4, A25=2,5',
'A27=2,7, A35=3,5, A38=3,8, A49=4,9, A50=5,10',
'A67=6,7, A60 = 6,10, A78=7,8, A89=8,9, A90=9,10').
```

The idea is that a single set of quotes should be sufficient, but that intermediate options should also be available.

In order to formulate suitable conventions for deleting quotes, we introduce several new terms. A sequence of index names separated by commas is an "index string". An index string between two quotes is a "quoted index string". A single index name in quotes is a special case of a quoted index string.

> Quote-deletion convention I: If two quoted index strings are separated by a comma, and if the second index string is not followed by an equal sign, then the two quotes around the separating comma may be deleted.

Application of convention I results in the merger of two quoted index strings into a simple one.

In LET MAP and LET RANGE SEQUENCE instructions, as well as in generalized subrange constructs, more complicated expressions involving equal signs are encountered. An expression of the form

'index name = index sequence'

is a "quoted index equivalence". Such quoted index equivalances are created using the following

> Quote-deletion convention II: If a quoted index sequence is preceded by an equal sign "=", then the two quotes around the equal sign may be deleted, creating a quoted index equivalance.

As a next step, adjacent quoted index equivalances may be merged, resulting in "quoted index equivalance strings".

Quote-deletion convention III: If two quoted index equivalance strings are separated by a single comma, then the two quotes around that comma may be deleted, resulting in the merger of the two strings into a single index equivalance string.

The above three quote-deletion conventions are sufficient to effect all the quote deletions discussed so far. They permit several intermediate deletion strategies such as writing

'1 = WALNUT', '2=OAK', '3=PINE', '4=SPRUCE'.

We need, however, one more convention in order to handle the non-index case.

Quote-deletion convention IV: For purposes of quote deletion, two quotes in succession or separated by blanks are treated in the same fashion as an index name enclosed in quotes.

As a result of this last convention, we may carry out, for instance, the following sequence of successive simplifications

'T2'='T1', 'T3'='T2', ' '='T3'

'T2 = T1', 'T3 = T2', '= T3'

'T2 = T1, T3 = T2, = T3'

CHAPTER 2: The Database

This chapter deals with issues concerning the database separately from the issues of the problem specification language. After describing the general structure of the database, input via a data statement (see Introduction) is discussed. Subsequently instructions for enlarging, purging, and modifying the database are proposed.

The reader will notice an apparent duplication of purpose: on the one hand a general input format for the database is described, while on the other hand mechanisms for enlarging the database via specification statements are proposed which could obviously also be used for setting up the database in the first place. This dualism has historical reasons, in that the early stages of the project only the input capability via the data statement was available. Editing and modifying the corresponding data file was envisioned to be handled directly or by extraneous data handling capabilities, but not as part of the problem specification language. This point of view was obviously too narrow, and NBS was encouraged to consider a corresponding broadening of the specification language. The question now arises whether the data statement is not rendered obsolete by the development of the additional specification algorithms. Also, there is the additional concern whether the two different modes of creating a database, with the slightly different conventions may not cause confusion. We reply that having the two modes of database creation may provide a useful flexibility. The direct input can certainly be handled more efficiently as there is less paring involved. In the present proposal, the capabilities of the two methods do not fully match: the data statement contains file manipulation capabilities which are already in the specification language and, vice versa, the data statement may not refer to previously defined tables when defining new ones. We welcome the opinions of the reader on this topic.

2.1 The Data Statement: General Conventions

As mentioned in the Introduction, the data statement represents a body of information headed by an &DATA card. Input is in the traditional "card format", that is, 72 characters of any line (card) following the &DATA card are read. All characters, numerical, alphabetical, and otherwise, are read in continuous sequence, that is, "ends-of-lines" are essentially ignored so that the last character on each line immediately precedes the first character on the next line. A line of all blank characters terminates the data statement.

The data statement may contain "comment lines" or "comment clauses". A comment line is any line whose first non-blank character is an

"*" (asterisk),

and where the character immediately succeding the asterisk is not an asterisk. We call such an asterisk a "single asterisk". A comment line is only for comment within the data statement and its content will be discarded upon input. A comment clause also starts with a single asterisk, but is preceded in its line by non-blank characters. The comment clause is terminated by the end-of-line, and its content is again discarded upon input. A "double asterisk", that is an asterisk followed immediately by another asterisk on the same line and without blank characters separating them, indicates a "text clause" to be described at the end of this section.

Terminating comment lines and clauses, delimiting the line of blanks which terminates the data statement, and possibly separating asterisks are the only roles played by end-of-lines in the data statement.

Input is composed of "names" and "numbers". All numbers are considered to be in single precision and are accepted in standard floating point notation. Decimal points are optional following integers. A number may carry a plus "+" or a minus "-" sign, but at most one sign per number. Names are arbitary alphanumeric strings. "Alphanames" are names that start with a letter. "Normal names" begin and end with non-blank characters, and contain no successive intermediate blanks.

The role of blank characters needs to be clarified. Blank characters can be used for separating successive numbers, although commas are recommended for this purpose. As a consequence, intermediate blanks are not permitted in a number. Such blanks would cause an intended single number to read as several numbers. Similarly, plus and minus signs as well as decimal points should not be separated by blanks from the rest of the number.

Intermediate blanks are, however, permitted in names. Names containing several successive blanks are normalized by suppressing blanks which succede intermediate blanks. The resulting normal names are stored for eventual report generation. For name recognition, only non-blank characters are taken into account. Names are limited to 24 characters after normalization.

The following simple general rules can be stated: blank characters within the data statement which are adjacent to one of the "special characters"

: , ( ) @ / + - #

or which succede another blank character, may be deleted without a change in content.

Not counting comment lines, the input is structured as a sequence of self-contained "blocks" of information. These blocks are separated by

";" (semicolon).

Termination of the last block by a semicolon is optional. A block is a "range block", a "table block", a "map block", or a "range sequence block" defining, respectively, an index range (Section 1.3), table (Section 1.3), index map (Sections 1.6, 1.8), and index range sequence (Section 1.9), respectively. A block can also be a "file block".

A file block simply associates a name with a read channel number. The so defined "file name" ("channel name" might be a more appropriate term) may appear in tables indicating that part or all of the table's content is to be read from a designated file rather than from the database file directly. This capability of conceptually inserting separate files into the database is important: it permits the database to act as a superstructure, integrating a collection of data files. In order to provide an immediate overview of those read channels which are utilized by the database, we require that file blocks precede all other blocks in the data statement.

Defining a table requires indicating the index ranges with which the table is indexed. The blocks defining these index ranges must therefore precede the block which defines the table. Index maps and index range sequences similarly require that their associated ranges precede them in the data statement.

At the end of any block, before the semicolon that terminates this block or before the blank line that terminates both the block and the data statement, a "text clause" may be inserted. This text clause is part of the block and associated with whatever index range, table, index map, or index sequence is being defined by the block. The text clause is headed by a double asterisk

'**' (double asterisk),

that is, two asterisks with no blank character in between. Contrary to comment lines and comment clauses, text clauses are saved upon input, and entered into a separate data directory file, where the text is available for reporting and querying.

2.2 Range Blocks

The index ranges in EXAMPLE 1 (Section 1.3),

METALS = {METAL 1, METAL 2} ,

ALLOYS = {ALLOY 1, ALLOY 2, ALLOY 3, ALLOY 4} ,

can be input using two blocks:

```
METALS: 'METAL 1', 'METAL 2';

ALLOYS: 'ALLOY 1', 'ALLOY 2', 'ALLOY 3', 'ALLOY 4'
```

or, using the quote deletion convections of Section 1.10,

```
METALS: 'METAL 1, METAL 2';

ALLOYS: 'ALLOY 1, ALLOY 2, ALLOY 3, ALLOY 4';
```

The following input is equivalent:

```
METALS: 'METAL 1, METAL 2'

;ALLOYS

:'ALLOY 1, ALLOY 2, ALLOY 3, ALLOY 4'
```

Indeed, since ends-of-lines are ignored, the only differences between the two input strings are superfluous blanks.

Formally, every range block consists of the alphaname of the index block, followed by a colon. After the colon a sequence of names, adjacent ones separated by a comma or a blank character, specifies the sequence of indexes (or keys) in the range.

In order to simplify the input of certain structured index names, the following constructs are also permitted.

Index ranges consisting of consecutive whole numbers such as, for instance, the index range

CONSECUTIVE INTEGERS = {0,1,2,3}

can be defined in terms of its first and last entries and can be input accordingly:

CONSECUTIVE INTEGERS: (0-3) .

The "range expression" (0-3) need not start with zero, but could instead start with any (reasonably small) positive integer.

Many indexes consist of a given "prefix" followed by consecutive integers. In this case a more general range expression,

METALS: 'METAL '(1-2)

defines the range METALS as described earlier in this section. Deleting the blank character after METAL leads to a different sequence of index names:

METALS: 'METAL'(1-2)

yields

METALS = {METAL1,METAL2}.

A "postfix" can be handled similarly:

AA: (1-4)'A'

defines the range

AA = {1A, 2A, 3A, 4A} .

Finally prefixes and postfixes can be applied to index ranges that have been defined earlier. For instance, given the range

YEARS = {Y80,Y81}

the range block

FISCAL: 'F'(YEARS)

will input the index range

FISCAL = {FY80,FY81} .

Frequently, the digital portions of index names are desired to contain an equal number of digits, such as in the range BB = {BB01, BB02,...,BB78}. This pattern can be specified by indicating leading zeros:

BB: 'BB'(01-78)

2.3 Table Blocks

We return again to EXAMPLE 1 in Section 1.1. Three tables were given:

SUPPLY

METAL 1	6 tons
METAL 2	5 tons

COMPOSITION

	ALLOY 1	ALLOY 2	ALLOY 3	ALLOY 4
METAL 1	0.5	0.6	0.3	0.1
METAL 2	0.5	0.4	0.7	0.9

PRICE

ALLOY 1	1000
ALLOY 2	1500
ALLOY 3	1800
ALLOY 4	4000

All the data describing EXAMPLE 1 can now be collected in an &DATA statement:

```
METALS: 'METAL '(1-2);

ALLOYS: 'ALLOY '(1-4);

SUPPLY(METALS): 6,5;

COMPOSITION(METALS,ALLOYS): .5, .6, .3, .1,

                            .5, .4, .7, .9;

PRICE(ALLOYS): 1000, 1500, 1800, 4000
```

There are two range blocks and three table blocks, one for each table. Each table block thus consists of an alphaname denoting the table followed by a k-tuple of alphanames in parentheses and separated by commas. These alphanames refer to previously defined index ranges. A colon then signals the beginning of the "table body", namely the set of numerical entries into the table. These entries are sequentially ordered according to the "odometer principle": the rightmost range runs continuously through its indexes in the specified sequence, starting over again after the range has been exhausted, and any other range passes to the next index whenever the range to the right of it starts over again. More precisely, the k-tuples of indexes to which the table entries are assigned are ordered as follows: start with the k-tuple that consists of the first indexes in each of the k index ranges. The first number in the table body is associated with the k-tuple of indexes. Now consider an arbitrary k-tuple of indexes from the k index ranges, respectively. If all the indexes are the last indexes in their ranges, then this k-tuple is the last one. Otherwise, we define the successor to the given k-tuple as follows: starting with the rightmost index in the k-tuple and proceeding to the left, find the first index that is not last in its range. Replace this index by its successor in the range, and reset all indexes to the right to the first indexes in their respective ranges. It is readily seen that this successor definition establishes a sequential order of all k-tuples, and the latter are than matched in this sequence with the number in the table body. This ordering is often called "lexicographic" or, in the case of 2-indexed tables, "row major order".

Instead of commas, blank characters may be used to separate the numbers in the table body, as in the following table block:

```
COMPOSITION(METALS,ALLOYS): .5 .6 .3 .1 .5 .4 .7 .9
```

A table block defining a "singleton table" simply skips the specification of associated ranges:

```
CONSTANT: 5.3
```

If a table calls for more entries than are specified in the content clause, then the values of the unfilled positions of the table default to

zero. Often this may be intentional, but in many other cases it may indicate a counting error during data preparation. For this reason, the user should have the option to indicate whether he expects the table to be fully defined by the content clause. It is proposed to use the word

FULL

preceded by a

"/" (slash)

as in the block

```
COMPOSITION(METALS, ALLOYS)/FULL: .5 .6 .3 .1
                                  .5 .4 .7 .9;
```

If /FULL is used, then an error message is issued whenever the content clause does not match the table size.

2.4 File Blocks, Content Clauses, and Locator Phrases

Writing

```
DISUT: #32 (J); * UTPS FILE TYPE J
```

provides an example of a file block. It associates the alphaname DISUT, the "file designation", with channel number 32. The letter "J" in parentheses indicates the "file type" (see comment clause in the above file block). The general form of a file block is the file designation followed by a colon, followed by a

"#" (number sign)

in front of a positive integer followed by an alphaname representing the file type in parentheses. We proceed to describe the use made of file designations introduced by file blocks.

A colon followed by a list of numbers separated by commas, blanks, or number signs is an instance of a "content clause". A content clause may also take the form of a colon followed by a file designation, such as in the following table block:

```
DISUTILITIES (OBSERVATIONS): DISUT
```

The latter states that a certain "external file" contains the numbers which constitute the table DISUTILITIES. Job control language (not to be discussed in this report) is required to link the external file to channel 32, which was associated with the file designation DISUT in the original file block.

A single table block may contain several content clauses. Such clauses are preceded by a "locator phrase" which describes their location in a table. The first number of the content clause will be entered into this location and the remaining numbers will be entered, in order, into the subsequent locations.

A locator phrase is initiated by an

"@" (at-sign).

This is followed by a sequence of index names separated by

"," (comma).

The first of these indexes must belong to the first index range of the table, the second index to the second index range, and so on. Index ranges of the table from which no index was selected in the locator phrase are considered represented by their first elements. For example, the same table is generated by each of the following table specifications:

```
  COST(YEARS,QUARTERS)

@ FY80, Q1: 30.2, 11.1, 22.4, 20.4

@ FY81, Q1: 18.7, 19.9, 11.9, 30.5
```

```
  COST(YEARS,QUARTERS)/FULL

@ FY80:  30.2, 11.1, 22.4, 20.4

@ FY81:  18.7, 19.9, 11.9, 30.5;
```

The locator phrases must be compatible with the lexicographic ordering of the table entries, that is, the locator phrases must be written in the sequence of the associated index tuples. They may not overlap. However, they are not required to cover every entry of the table. If the user expects the entire table to be defined by the content clauses, then he may indicate this by adding /FULL to the table name (see Section 2.3).

Locator phrases can be used to input sparse matrices. The adjacency matrix for the network in Section 1.8 can be input as follows, given the ranges

```
NODES = {N1,N2,N3,N4}

ARCS = {A1, A2, A3, A4}:
```

```
NETWORK(ARCS,NODES):     @A1,N1:-1   @A1,N4:+1
                         @A2,N3:+1   @A2,N4:-1
                         @A3,N1:-1   @A3,N3:+1
                         @A4,N1:+1   @A4,N3:-1
                         @A5,N1:-1   @A5,N3:+1
                         @A6,N1:-1   @A6,N2:+1
                         @A7,N2:+1   @A7,N3:-1
```

In the above example, each line in the table body corresponds to a column in the adjacency matrix. If the order of the ranges ARCS and NODES were to be interchanged, a row orientied input pattern would emerge:

```
NETWORK(NODES,ARCS):

@N1,A1:-1          @N1,A3:-1 @N1,A4:+1 @N1,A5:-1 @N1,A6:-1

                                                  @N2,A6:+1 @N2,A7:+1

      @N3,A2:+1 @N3,A3:+1 @N3,A4:-1 @N3,A5:+1              @N3,A7:-1

@N4,A1:+1  @N4,A2:-1
```

Thus locator phrases can be used for table specifications in which only nonzero elements are given. For large "sparse tables", this may save considerable work in input preparation. The use of locator phrases may also lay the ground work for compressed (nonzero elements only) storage within the database .

Finally, file designations may be used in content clauses following locator clauses as in the data statement below:

```
DISUT  1: #41(J)  DISUT 2:  #23(J)
OBSERVATIONS:  (001-573);
YEARS:  1981, 1982;
DISUTILITIES (YEARS, OBSERVATIONS):
                  @ 1981: DISUT 1,
                  @ 1982: DISUT 2;
```

2.5 Map Blocks and Range Sequence Blocks

Consider the origin map of the network in Section 1.8. Its image and domain ranges, respectively, were

NODES = {1, 2, 3, 4}

ARCS = {1, 2, 3, 4, 5, 6, 7}.

A map block for inputting this information through a data statement reads as follows:

```
NODES(ORIGIN)ARCS: '1', '4', '1', '3', '1', '1', '3' .
```

Here ARCS is the domain, NODES the image range. Their definition needs to precede the map block. ORIGIN is the name of the index map. In the remainder of the block, the image indexes from the range NODES appear in the order of the domain indexes from the range ARCS to which the image indexes are assigned. Thus index 1 in NODES is assigned to index 1 in ARCS, index 4 in NODES to index 2 in ARCS, and so on. Using two quotes with nothing or only blanks inbetween indicates a non-index (see Section 1.8).

If the user prefers a more redundant format for checking purposes, he may use an alternate form of the map block:

```
NODES(ORIGIN)ARCS:'1'='1','4'='2','1'='3','3'='4','1'='5','1'='6','3'='7';
```

Here the associated pairs of indexes are listed explicitly. Indexes from the image range NODES are to the left, and indexes from the domain range ARCS are to the right of their respective

"=" (equal)

signs.

In Section 1.9, the network of Section 1.8 was characterized by two index range sequences F and B, representing forward and backward stars in the network. A range sequence block may be used to include, for instance, the range sequence F in the data statement:

```
ARCS()F(NODES):  ('1', '3', '5', '6') () ('4') ('2')
```

Here ARCS is the universal range of the index range sequence, and NODES is its domain. 'F' is the name of the index range sequence. The image ranges, consisting of indexes in the universal range ARCS, are listed in the sequence of the indexes in the domain range to which they correspond. Again, there is an alternate more redundant notation,

```
ARCS()F(NODES):  ('1','3','5','6') = '1',  ()='2',
                 ('4') = '3', ('2') = '4'
```

in which the indexes in the domain are also represented. In addition, quote deletion convertions (see Section 1.10) may be applied in both instances, replacing ('1', '3', '5', '6') by ('1,3,5,6').

2.6 Database Manipulations: Specification

For the database system to be useful in a broader context, two classes of additional features are needed: 1) a set of "specification instructions", which permit the conceptual definition of new tables in terms of existing tables; 2) a set of "disposition instructions", which permit the actual generation of new tables and ranges as well as the modification of old ones. In this category belong instructions which create new databases in addition to the original one and which generate files outside the database.

At a minimum, specification instructions should provide the following capabilities: defining small tables and ranges from scratch, modifying tables or ranges, renaming tables and ranges, extracting smaller tables from larger ones, as well as aggregating table entries, combining tables. In accordance with an analogous usage in the UTPS package UFIT, we propose to use the keyword

LET

in conjunction with keywords TABLE, RANGE and RANGE SEQUENCE for specification instructions.

For instance by writing

LET RANGE(R:='1','2','3')

a new range

R = {1,2,3}

is introduced. The equal sign "=" after the colon is optional. The following instruction defines the same range as the instruction above:

LET RANGE (R: '1','2','3').

This equivalence of the symbols ":" and ":=" holds for all specification instructions. Deletion of indexes from a range is indicated by a slash "/" preceding the indexes to be deleted. For ranges

U = {'1','2','3','4','5'} V = {'4','5'},

each of the following two instructions,

LET RANGE (R: = U/'4', '5'),

LET RANGE (R: = U/V),

produces the above range R. The first of the above instructions is equivalent to using a generalized subrange construct

LET RANGE (R: = U(''='4',''='5')U).

In addition LET RANGE instructions may utilize range expressions involving prefixes and consecutive integers as described in Section 2.2:

LET RANGE (METALS: 'METAL'(1-2))

In general, any expression that is legal for range blocks in a data statement can also be used in a LET RANGE instruction. However, the converse is not true.

A table T with range R can be defined from scratch by writing

LET TABLE (T(R): T('1') =-3.8, T('3')=4.2).

Entries into table T that are omitted in this definition default to zero. Thus only nonzero elements need to be specified. As an abbreviation, we permit instructions of the form

LET TABLE (T(R):=-3.8,0,4.2)

for defining the identical table. The two notations may be combined. For the subrange

S = {1,2}

of R, we write:

LET TABLE (T(R): T(R()S)=-3.8,0, T('3') = 4.2).

We write T(R()S) in the above instruction because of the general principle that either an associated range or a subrange construct with the associated range as screening range should be used for range identification following a table name. The range S drives the portion

T(R()S) = -3.8, 0

of the LET TABLE instruction, that is, for each consecutive index in S the corresponding numerical quantity from the latter's sequence is entered into the table T(R). Between the parenthesis an index map or directly specified index equivalences could be employed legally, although the utility of such a construct is questionable under these particular circumstances.

When defining a table of several ranges, the entries are arranged according to a lexicographic or odometer-like (see Section 2.3) ordering of the corresponding index tuples. For instance, if

LET TABLE(TAB(R1,R2):= .5, -1.3, 0., 2.7, .9, 1.5, -2.1, -.8)

where

R1={R11,R12}, R2={R21,R22,R23,R24},

then

TAB(R11,R21)=.5, TAB(R11,R22)= -1.3, TAB(R11,R23)=0., TAB(R11,R24) = 2.7

TAB(R12,R21)=.9, TAB(R12,R22)=1.5, TAB(R12,R23)=-2.1, TAB(R12,R24)=.8.

The use of specification instructions for the fully numerical definition of tables is only recommended for a small number of tables of small size. In all other cases, a data statement should be used to enter larger data sets into the database.

Table definitions may also rely on other tables, which are assumed to be already defined or which are part of the database. The simplest definitions of this kind are renaming instructions.

For example, suppose a database contains the ranges

VEHICLES = {AUTOMOBILE,MOTORCYCLE}

MODES = {BUS,AUTOMOBILE,MOTORCYCLE}

SECTIONS = {NW,NE,SE,SW}

YEARS = {80,81,82,83}

and the table

SURVEY(MODES,SECTIONS,YEARS).

A straightforward renaming of a table would be effected by the following instruction, which introduces the new table name S :

LET TABLE(S(SECTIONS,MODES,YEARS):=SURVEY(MODES,SECTIONS,YEARS)).

The reader notices that the sequence of ranges has been changed. Similarly the instructions

LET RANGE(MS:=MODES); LET RANGE(SS:=SECTIONS); LET RANGE(YS:=YEARS)

LET TABLE(S(SS,MS,YS):=SURVEY(MODES()MS,SECTIONS()SS,YEARS()YS))

will introduce new range names as well as a new table name.

Extraction can be handled in a similar fashion. For instance,

LET TABLE(E(MODES,SECTIONS):=SURVEY(MODES,SECTIONS,'83')

extracts the 2-dimensional subtable for 1983 data from the 3-dimensional SURVEY table. The subrange constructs as described in Sections 1.5 and 1.6 may be used for the extraction of subtables:

```
LET TABLE(T(VEHICLES,SECTIONS):=SURVEY(MODES()VEHICLES,SECTIONS,'83').
```

VEHICLES is, of course, a subrange of MODES. If V was not a subrange of MODES, but related to the latter by an index map, say,

```
V = {V1,V2},

LET MAP (MODES(REF)V:'AUTOMOBILE'='V1','MOTORCYCLE'='V2'),
```

then the instruction

```
LET TABLE(T(V,SECTIONS):=SURVEY((MODES(REF)V,SECTIONS,'83'))
```

will produce the same extraction as the subrange construction above.

Combination of tables may result in new tables of the same or of higher dimensions. Consider, for instance, the three tables

```
A(R1,R2,RA),  B(R1,RB,R2),  C(RC,R2,R1),
```

where two ranges, R1 and R2, are a part of all three tables, albeit in different positions, and where the other ranges, RA, RB, and RC, have no keys in common. In order to define a combination of the three tables, we first introduce a new operator symbol, the "underline", to indicate "concatenation" of ranges:

```
LET RANGE(R3:=RA_RB_RC)
```

Here a new range is formed using the indexes of RA followed by the indexes in RB and subsequently RC, all indexes being in the order of their respective ranges. We can now write

```
LET TABLE(D(R1,R2,R3): D(R1,R2,R3()RA)= A(R1,R2,RA),

                       D(R1,R2,R3()RB) = B(R1,RB,R2),

                       D(R1,R2,R3()RC) = C(RC,R1,R2))
```

In its structure, this instruction resembles the previously discussed instruction

```
LET TABLE(T(R): T(R()S)=-3.8,0, T('3')=4.2)
```

Again subrange constructs, in the present case

R3()RA, R3()RB, R3()RC,

are employed, in keeping with the general principle that range identification after table names should always tie in with the ranges that are associated with the table in question. Generalized subrange constructs and maps may be used and might be useful, if the relationship between R3 and the ranges RA, RB, RC were less straightforward. Each of the portions

D(R1,R2,R3()RA) = A(R1,R2,RA)

D(R1,R2,R3()RB) = B(R1,RB,R2)

D(R1,R2,R3()RC) = C(RC,R1,R2)

of the LET TABLE instruction is executed independently. In the first case, the index triples defined by

(R1,R2,RA)

are passed through in lexicographical (odometer) order, causing the entries of table A to be distributed into the new table D.

For a similar construction consider the three tables over a common range R

A1(R), A2(R), A3(R)

and the range

S = {1,2,3}.

The latter can be used to combine the three tables A1, A2, A3 into a 2-dimensional table:

LET TABLE(A(R,S): A(R,'1')=A1(R),A(R,'2')=A2(R),A(R,'3')=A3(R))

The two combination devices may be used simultaneously. Consider the tables

L(R1,RA), M(R1), N(RB,R1).

The instruction

LET RANGE(R2:=RA_'MI'_RB)

defines a new range, provided the index MI occurs neither in RA nor RB, and RA and RB have no index in common. The new range R2 consists obviously of the indexes in RA followed by MI and the indexes in RB in that order. Then

LET TABLE (U(R1,R2): U(R1,R2()RA) = L(R1,RA)

U(R1'MI') = M(R1),

U(R1,R2()RB) = N(RB,R1))

defines the table in which the vector M is appended to the matrix L, and the result is concatenated with the matrix N.

Free index ranges as introduced in Section 1.3 can be employed to effect aggregation. In this spirit, the statement

```
LET TABLE(AGG(MODES,SECTIONS):=SURVEY(MODES,SECTIONS,YEARS))
```

is interpreted as providing table entries which are aggregations over the number of years as given by the range YEARS. The summation over YEARS, keeping MODES and SECTIONS fixed, is indicated by the fact that the range YEARS does not occur among the ranges MODES and SECTIONS of the table to be defined.

Aggregation may be combined with extraction as in the instructions

```
LET TABLE(BUS USE(SECTIONS):=SURVEY('BUS',SECTIONS,YEARS))

LET TABLE(VEH AGG(VEHICLES):=SURVEY(MODES()VEHICLES,SECTIONS,YEARS)).
```

The second instruction uses a subrange construct for extraction while at the same time aggregating both over sections and years.

Provisions have been made for defining a new table as a modification of some previously defined table, for instance, by writing

```
LET TABLE(S(SECTIONS,MODES,YEARS):= SURVEY(MODES,SECTIONS,YEARS):
                                    S('BUS','NE','81')= .375,
                                    S('BUS','SE','82')= .125)
```

The resulting table S will agree with the table SURVEY in all entries except the two entries characterized by the index triples BUS, NW, 81 and BUS, SE, 82, respectively. Those two entries in table S will have the specified values. Such "modification clauses" can be attached to any specification instruction as indicated by the following two examples:

```
LET TABLE(BUS USE(SECTIONS):=SURVEY('BUS',SECTIONS,YEARS):BUS USE('NW'):=279)
```

```
LET TABLE (D(R1,R2,R3): D(R1,R2,R3()RA)=A(R1,R2,RA)
                        D(R1,R2,R3()RB)=B(R1,RB,R2)
                        D(R1,R2,R3()RC)=C(RC,R1,R2):
                        D('R11','R21','C1')=5.3)
```

Here we have assumed that NW is an index in range SECTIONS, and that R11,R21,C1 are indexes in R1,R2,R3, respectively.

The modification clauses may take more general forms. If, for instance,

R={1,2}, S={1,2}, T={1,2,3}, U={1,2,3}

one may want to write

```
LET TABLE(A(R,S,T,U):=B(R,S,T,U):A('1',S,'3',U):=7,4,5,5,3,6).
```

As a result,

A('1','1','3','1')=7, A('1','1','3','2')=4, A('1','1','3','3')=5,

A('1','2','3','1')=5, A('1','2','3','2')=3, A('1','2','3','3')=6.

Note that the above instructions do not provide "modifications" in the strict sense of the word. Every table, once defined, remains unchanged. If changes are made, then the result must be given a new name. In other words, an instruction of the form

```
LET TABLE (T(S):=T(S): T('1')=0)
```

is <u>not</u> permitted. ULP also checks for inadvertent circular definitions, which might have the same effect. True modifications are ruled out, because they would violate the principle that the sequence in which the instructions are written in the run statement should not matter.

In Appendix B, we provide a BNF (Backus-Naur Form) description of the general LET TABLE instruction.

2.7 Database Manipulations: Disposition

Disposition instructions either modify an existing database or create new files. It may also be desirable to create an entirely new database in addition to one that already exists. Disposition instructions which create new files or new databases should tie in closely with UTPS file conventions and reflect general UTPS needs. The creation of suitable J-files or Z-files, for instance, should be an option. Any disposition instruction is an execution instruction just as the LPMIN and LPMAX instructions, which trigger the execution of linear programming code.

The computational effort associated with a disposition instruction may be considerable, particularly if a rearrangement of the database is required, for instance by increasing the size of a particular table. Even changing the value of a single table entry may require such a rearrangement if a zero entry is replaced by a nonzero entry and the corresponding table had been stored in a compressed form listing only nonzero elements.

Two main database modifications involve appending and deleting new tables, ranges, maps, and range sequences from the database in question. Tables and ranges which are to be appended must be defined prior to that with the help of specification instructions (see Section 2.6). We use the keywords

APPEND, DELETE

combined with key words

TABLE, RANGE, MAP, RANGE SEQUENCE

to write disposition instructions for appending and deleting the corresponding constructs to and from the database, respectively. A run statement may contain several APPEND instructions or several DELETE instructions but may not use APPEND instructions together with DELETE statements. Similarly, a run statement containing an LPMIN or LPMAX instruction may not contain any other disposition instructions. Typical run statements would then read:

```
LET TABLE(S(M,K):=T(K,M,'81'): S('MA','KB'):=1.0)

APPEND TABLE(S(M,K))
```

```
DELETE TABLE(T(K,M,Y))

DELETE RANGE(K,M,Y)
```

In the first run statement, we assume that MA and KB are indexes in ranges M and K, respectively. The second run statement deletes table T together with all its ranges. The command to delete a range is ignored, however, if the range is used by some other still existing table.

In principle, these are all the modification instructions needed for the purpose of database modification. Indeed, modifying an element of a table can be achieved by first specifying a new table with the modified element, then appending this table while deleting the original one, and finally renaming the modified table via specification and going through yet another append-delete cycle. This procedure is complicated. Direct modification instructions may therefore be desirable, but are not discussed in this report because they involve considerable conceptual and implementational difficulties.

CHAPTER 3: Report Generation

3.1 Reporting Requirements for Linear Programming

A report structure for linear programs is now proposed. This report structure follows general UTPS guidelines. For the purpose of displaying information about the solution of a linear program, more flexibility is needed than a straightforward menu-scheme such as provided by the customary REPORT statements. This need is caused by: (i) the potentially very large size of the solution vector, not all portions of which may be of interest to the analyst; (ii) the existence of different desirable display modes, mainly "tabular" versus "vectorial"; (iii) the desirability to vary the representation and juxtaposition of different display elements.

Output might be enhanced by generating various diagrams and summary data. However, such output enhancements should probably be left to general purpose diagramming and analysis packages in UTPS. Output and analysis enhancements are therefore not discussed in this report, except for the recommendation that FILE statements be provided which permit saving solution information for input into other UTPS analysis modules.

As a vehicle for added flexibility, we propose a "FORMAT statement" to be used in conjunction with a particular REPORT number. The function of this FORMAT statement is to specify the content and the layout of the report, while it remains for the REPORT statement to trigger the actual printing or display of the information.

Any method for solving a linear program will produce one of three possible outcomes: (i) (Normal termination) The linear program has optimal solutions. The optimal solution is usually, but not always, unique. (ii) (Infinite Solution) The linear program has feasible but no optimal solutions. The objective function can be made arbitrarily small (big) by selecting far-out feasible solutions. This cannot happen if the feasible solutions are restricted to a finite area. (iii) (Infeasibility) The linear program has no feasible solutions.

It may happen that some equations in a standard linear program are linear combinations of the remaining equations. We call them "linearly superfluous". It may also happen that, for all feasible solutions, a variable assumes only a single value, namely the value of one or the other of its bounds. If the bounds of such a variable differ, in other words if the variable is not already fixed by its bounds, we call the variable "constant at lower (upper) bound". The analyst may wish to be appraised of such occurrences. An equation in a standard linear program is called "superfluous" if it can be deleted without increasing the set of feasible solutions. While it would be most certainly desirable to identify superfluous constraint equations, this identification is not an automatic result of the solution procedure.

In the case of normal termination, the optimal solution, if unique, is found at a vertex of the feasible region. If there are several optimal solutions to the same linear program, then there is at least one optimal solution among them which lies at a vertex. The simplex method of linear

programming will always find an optimal solution which represents a vertex. Such optimal solutions have the following:

> Simplex Property: Let m be the number of nonsuperfluous equations in a standard linear program, and let n be the number of its variables. Then the variables are divided into two sets: m "basic variables" and n - m "nonbasic variables". Each nonbasic variable assumes the value of one of its bounds, upper or lower, whereas the basic variables are permitted to range between their bounds.

The values of the basic variables can be calculated from the constraint equations after the values of the nonbasic variables have been substituted into these equations. The values of the basic variables are thus uniquely determined by the nonbasic variables.

The simplex property furnishes a qualitative interpretation of the optimal solution to a linear program: it characterizes the variables which assume their boundary values. In general, this characterization is insensitive to small perturbations in the input data, and this insensitivity of the qualitative information is one of the reasons for the extraordinary usefulness of linear programming techniques.

Clearly, the limits to which perturbation of the data can be pushed while still yielding an optimal solution with the same "basis", that is, the same split between basic and nonbasic variables, is of considerable interest. Information about such limits is generally called "ranging" information. If arbitrary small perturbations could change the basis of the optimal solution, then we call this optimal solution "degenerate" with respect to this perturbation. Ranging information can also be interpreted as indicating how much change can be sustained before degeneracy is encountered.

3.2 Output Elements

In this section, we describe output elements which can be made available after normal termination and which the analyst may want to display. Of first and foremost interest are, of course, the values which the variables $x_1, \ldots, x_n$ assume for the optimal solution. We refer to these as "optimal values" or just "values". Next, the analyst wants to know the "objective value", that is, the value of the objective function at the optimal solution. Almost equally important is the distinction between basic and nonbasic variables. Those remaining quantities which are of interest as output elements provide mostly sensitivity information. All these output elements fall into three categories: (i) quantities associated with nonbasic variables, (ii) quantities associated with basic variables, (iii) quantities associated with constraints.

We now discuss information of interest about nonbasic variables. Recall that the value of such a variable is at one of its bounds. For the majority of variables encountered in practice, the lower bound of a variable x_i is zero, $L_i = 0$, and the upper bound is infinite, $H_i = +\infty$. In this case, the nonbasic variable has value zero, $x_i = 0$. In particular, if all bound statements of the linear program are of the form $x_i \geq 0$, as is often the

case, then all nonbasic variables have automatically value zero. For this reason, the value of a nonbasic variable is in general not quite as interesting as the value of a basic variable. Indication of which bound's value is being assumed is very often all the information desired.

With every nonbasic variable, the simplex method associates a "reduced cost" c_j. This reduced cost is nonnegative, $c_j \geq 0$, if the variable is not fixed and is at its lower bound, and is nonpositive, $c_j \leq 0$, if the variable is not fixed and is at its upper bound. If positive, the reduced cost c_j for a nonbasic variable x_j is the marginal cost for increasing variable x_j by one unit (holding all other nonbasic variables at their value). It shows how much the objective value could be improved if the lower bound were to be decreased, and this measures the significance of a lower bound constraint for a particular linear program. If the reduced cost c_j is negative, then the absolute value $|c_j|$ is the marginal cost for decreasing the variable x_j, and the same observations hold with respect to upper bounds.

If the reduced cost c_j vanishes, then the value of its corresponding variable x_j can be perturbed without affecting the objective value. In general, this will indicate the presence of multiple optimal solutions. The vanishing of reduced costs for a nonbasic variable is a degeneracy phenomenon. It is referred to as "dual degeneracy". Another degeneracy phenomenon pertains if a basic variable assumes the value of one of its bounds. This phenomenon is called "primal degeneracy". A sufficient condition for multiple optimal solutions to occur is dual degeneracy in the absence of primal degeneracy.

The reduced costs c_j also provide ranging information. If c_j is positive (negative) for the optimal solution, then the original cost coefficient c_j can be increased (decreased) by an arbitrary positive amount without changing the optimal solution. A decrease (increase) of the cost coefficient c_j by any amount less than $|c_j|$ can be similarly sustained. These observations are valuable in practice, because they permit the analyst to assess the limits of cost modifications for which the optimal solution remains valid.

To indicate the output elements to be displayed for nonbasic variables, we propose statements of the form:

```
FORMAT(NONBASIC = VALUE, REDUCED COST)

FORMAT(NONBASIC = VALUE, BOUND INDICATOR)

FORMAT(NONBASIC = REDUCED COST, LOWER BOUND, UPPER BOUND)
```

The first statement will provide the value and the reduced cost for each nonbasic variable in the indicated order. The second statement features values followed by bound indicators: "HIGH", "LOW". The third statement specifies the reduced costs and the bounds for printing.

To sum up: for nonbasic variables, the following output elements suggest themselves:

VALUE
BOUND INDICATOR
LOWER BOUND
UPPER BOUND
REDUCED COST

} for nonbasic variables.

We consider the term "NONBASIC" in the above FORMAT statements a "class specification". The "=" sign followed by the names of output elements to be reported, e.g.,

= VALUE, REDUCED COST,

constitutes an "output specification".

We now turn to basic variables. Here the value is definitely of primary importance, and so are measures of the round-off "errors" incurred during its calculation. It may also be of interest to know the location of the value with respect to its bounds. Cost-ranging information is somewhat more complex than in the case of nonbasic variables: both "lower" and "upper cost ranges" are needed to limit the area of insensitivity. Thus we list the following output elements:

VALUE
ERROR
UPPER BOUND
LOWER BOUND
UPPER COST RANGE
LOWER COST RANGE

} for basic variables.

Using the class specificaion "BASIC", FORMAT statements might read:

FORMAT(BASIC = VALUE, ERROR)

FORMAT(BASIC = VALUE, UPPER BOUND, LOWER BOUND)

FORMAT(BASIC = UPPER COST RANGE, LOWER COST RANGE).

In many instances, the separate display of basic and nonbasic information may be undesirable: one would like the option to write statements like

FORMAT(SOLUTION = VALUE, ERROR, UPPER BOUND, REDUCED COST)

FORMAT(SOLUTION = VALUE, UPPER COST RANGE, LOWER COST RANGE).

If in these instances, the quantities are not defined for all variables, simp defaults are chosen. Thus every output element that can be used with the class specification "BASIC" or the class specification "NONBASIC" can also be used with class specification "SOLUTION".

The most important quantity associated with a constraint is the celebrated "shadow price". Interpretations of shadow prices abound, and this is not the pla for a discussion. Essentially the shadow price measures the price paid implicitl for the restriction caused by the constraint. Aside from shadow prices, the valu of the slack variables associated with the constraints are of interest. Frequentl again it is the sign of the slack variable within the constraint and the question whether the slack value is positive or zero that may be of more immediate interest the analyst than the actual slack value. Ranging information attaches to the constant terms of the constraints: there are "upper" and "lower constant ranges" which these constants or "right hand sides" could be increased or decreased, respectively, without changing the basis of the optimal solution. We thus have identified the following output elements, which are compatible with the class specification "CONSTRAINT":

SHADOW PRICE
SLACK
SLACK INDICATOR
UPPER CONSTANT RANGE
LOWER CONSTANT RANGE

} for constraints.

Corresponding FORMAT statements might read:

FORMAT(CONSTRAINT = SHADOW PRICE, SLACK INDICATOR)

FORMAT(CONSTRAINT = SLACK, LOWER CONSTANT RANGE, UPPER CONSTANT RANGE).

3.3 Output Qualifications

Since the number of variables may be very large, ways of selecting variables for printing must be provided. We recall that the solution is set up as a sequence of tables. A solution, which includes slack variables, might be of the form:

X(FARES,REGIONS), Y(INTERVIEWS), U, V, $(MODES)

Clearly one would like the option to print out only those tables which are of interest. Correspondingly, we propose a "selection clause" to be included in the FORMAT-statement:

FORMAT(SOLUTION = VALUE/X(FARES,REGIONS))

The power of the selection clause can be extended in various ways. An obvious way would be the specification of keys within ranges. Suppose the range FARES would contain the key "MEDIUM". Then the FORMAT-statement

FORMAT(SOLUTION = VALUE/U, V, X('MEDIUM',REGIONS))

would be self-explanatory. Note that the selection clause can also be used to control the sequence in which the solution elements are displayed. In the absence of a "selection clause", the sequence determined by the UNKNOWN statement will be the sequence in which the solution will be exhibited. These options by themselves are probably insufficient. Extracting subtables with the help of subranges of some kind may have to be accomplished in addition to the above options.

FORMAT statements with the class specification "CONSTRAINT" may employ a selection clause in an analogous fashion. Class specifications "BASIC" and "NONBASIC" will operate on an intersection basis, e.g., they will print those basic (nonbasic) variables which are also included in the selection clause.

3.4 Output Format

In this section, we will discuss the actual arrangement of solution information. There are essentially two options, which we call "sequential" and "tabular", respectively. The sequential option is to display, for each selected item of the specified output category, the corresponding values of the output elements indicated in the specification clause of the FORMAT statement. These values are preceded by the item name, and the so-defined output phrases are sequentially listed. The sequence of this listing either defaults to a natural sequence of the items within the category in question or is defined by a selection clause. The tabular option, on the other hand, displays each of the tables which constitute the selected output as a "table". This means that not every output item is separately named but rather identified by its position within a tabular arrangement. Each table entry will consist of all the values of specified output items which are characterized by the same keys. We indicate the output options through "display specifications":

(SEQUENTIAL), (TABULAR)

after the category specifications, e.g.:

```
FORMAT(SOLUTION(SEQUENTIAL) = VALUE, UPPER BOUND/X(FARES,REGIONS))

FORMAT(SOLUTION(TABULAR) = VALUE, UPPER BOUND/X(FARES, REGIONS)).
```

In the absence of specification, the sequential display option is assumed. The outputs corresponding to the above FORMAT statements might read:

Sequential:

```
X(LOW, NEW ENGLAND)        =  3.27  3.00  5.00
X(LOW, MID ATLANTIC)       =  3.00  3.00  5.00
X(LOW, SOUTH ATLANTIC)     =  3.07  3.00  5.00

X(MEDIUM, NEW ENGLAND)     =  3.50  3.50  5.00
X(MEDIUM, MID ATLANTIC)    =  3.50  3.50  5.00
X(MEDIUM, SOUTH ATLANTIC)  =  3.54  3.50  5.00

X(HIGH, NEW ENGLAND)       =  4.35  4.00  5.00
X(HIGH, MID ATLANTIC)      =  4.25  4.00  5.00
X(HIGH, SOUTH ATLANTIC)    =  4.00  4.00  5.00
```

Tabular: X	NEW ENGLAND	MID ATLANTIC	SOUTH ATLANTIC
LOW	3.27	3.00	3.07
	3.00	3.00	3.00
	5.00	5.00	5.00
MEDIUM	3.50	3.50	3.54
	3.50	3.50	3.50
	5.00	5.00	5.00
HIGH	4.35	4.25	4.00
	4.00	4.00	4.00
	5.00	5.00	5.00

Additional display specifications may be needed to govern the numbers of digits carried. We suggest that such specifications be entered after the output item in the output specification and follow FORTRAN usage, e.g.,

FORMAT(SOLUTION = VALUE(F5.2), LOWER BOUND(F5.2), UPPER BOUND(F5.2))

3.5 Non-normal Terminations and Superfluity

In the case of an infinite solution two output items are of interest: a "feasible solution" and an "infinite ray". We introduce the class specification "INFINITE" for FORMAT statements such as

FORMAT(INFINITE = VALUE, RAY).

If there is no feasible solution, then there exist "infeasibility multipliers" for the constraints such that the resulting linear combinations of the constraint equations, the "inconsistency", cannot be satisfied by variables within their bounds. We introduce the class specifications "INFEASIBLE" and "INFEASIBLITY CONSTRAINT" and write:

FORMAT(INFEASIBLE = INCONSISTENCY, LOWER BOUND, UPPER BOUND)

FORMAT(INFEASIBLE CONSTRAINT = MULTIPLIER).

Finally, if there are superfluous equations, then the class specification "SUPERFLUOUS CONSTRAINT" is employed:

FORMAT(SUPERFLUOUS CONSTRAINT = MULTIPLIER).

3.6 REPORT Statements

We propose the following REPORT options:

REPORT 1: Echos linear program specifications including slack variables (standard form). States number of variables, and number of relevant and superfluous constraints.

REPORT 2: Displays run-history: number of pivot steps, number of reinversions, etc. Indicates outcome (normal, infinite solution, infeasibility).

REPORT 3: Displays solution quantities according to FORMAT-statement in case of normal termination and displays objective value.

REPORT 4: Displays feasible solution and infinite solution ray in case of an infinite optimum.

REPORT 5: Displays coefficients for the constraint equations leading to a contradictory equation in case of infeasibility.

REPORT 6: Displays superfluous equation constraints as linear combinations of relevant constraints in case some constraint equations are superfluous.

REPORT 7: Indicates degeneracies in the optimal solution.

3.7 Thoughts About Units

Output reports should identify the units in which the output quantities are being measured. The approaches to providing the capability for handling information about units range from strictly output-oriented direct specification to integrating units into the database as an essential part of the information. In the latter case, the units for the unknown variables would be automatically determined from the units of the quantities used in problem specifications.

Whatever approach is ultimately selected, the following problem needs to be examined: Given a multidimensional table, what are the units of its elements? At first blush, it appears that each element in a table should be entitled to its own unit of measurement. However, in most practical situations, this is not the case. Indeed, a large portion of tables used in practice will be either "homogeneous", that is, all entries will be in terms of the same unit, or "quasihomogeneous", that is, one of their ranges will have units attached to its keys and these keys will determine the unit of the table element.

For example, the table

SALES (PRODUCTS, YEARS) [K$]

	1979	1980	1981	1982
PRODUCT A	1021	1003	937	420
PRODUCT B	853	916	921	902
PRODUCT C	75	74	82	89

is homogeneous. A corresponding table PRODUCTION (PRODUCTS,YEARS) would be quasihomogenous if there are different production units for different products. A table containing area and population of various countries would be another example of a quasihomogeneous table. In a homogeneous table, aggregations can proceed over every range; in a quasihomogeneous table, some directions of aggregation will be illegal.

Mechanisms for specifying units in a tabular data environment will have to rely on some assumptions about homogeneity properties of tables. Assuming, for instance, that all tables are in fact homogeneous, then a "UNIT Statement" could be invoked, say

UNIT(X(FARES,REGIONS) = CENTS),

to assign units for output purposes. A similar approach was taken in the design of UTPS module UFIT. An alternative way would be to specify units through the UNKNOWN statement:

UNKNOWN(X(FARES,REGIONS)[CENTS]).

For quasihomogeneous tables, the unit specifications process is necessarily more involved, but should present no major problems.

CHAPTER 4: Implementation

Some of the problem specification features described in Chapter 1 have been implemented in a prototype package called ULP. In this chapter, we outline the method of implementation. We will not restrict the discussion to the prototype implementation, but also address implementation techniques for some of the more advanced features, such as index maps and specification instructions. We will, however, not discuss the implementation of the report writer (see Chapter 3), because no implementation experience is available and the proposed form is still very much in the discussion phase.

The implementation of the problem specification portion of ULP involves three stages. Tokenization of the run statement, that is, the identification of symbols and character groups forming words, is followed by a normalization procedure, which involves the introduction of slack variables, the identification of constants, the processing of LET instructions, and finally the identification of index names and subrange maps in CONSTRAIN, LPMIN, LPMAX, and BOUND instructions.

Finally, there is matrix generation, namely the setting up of the actual linear programming matrix in column-sparse form along with the right hand sides and the objective row. Matrix generation techniques are also used for run statements involving APPEND instructions.

The implementation of the database system is a separate subject. The database system as implemented in the prototype ULP package uses a somewhat different format than the one proposed in Chapter 2. Rather than discussing the differences, we will outline a possible method for implementing the database system in the form proposed in this report. There are two issues: the interpretation and disposition of data statements (Sections 2.2-2.5) on the one hand, and the interpretation and execution of run statements intended for expanding and updating the database (Sections 2.6 and 2.7) on the other. The latter issue will be dealt with in the context of matrix generation.

4.1 Overview

We return to the lexical analysis of the run statements. The objective is to replace the alphanumeric names of coefficients, constraints, indexes, variables, etc., as well as the digital representations of constants, arithmetic operators, equal signs, inequality signs, delimiters by integer codes or "tokens". The process of tokenization will be discussed in Section 4.4. At this point we assume that the problem specification is encoded in a form in which integer tokens rather than alphanumeric names, decimal numbers, or special characters represent the items of the problem specification. We call this representation the "tokenized form" of the problem specification. The tokenization is carried out using two UTPS utility routines, ULEX and UNAMEL.

The UNAMEL routine, developed by R. Cody of Peat, Marvick, and Mitchell, organizes the alphanumerical strings of the run statement by keywords. A special subroutine, which resides within UNAMEL and is called "ACCESS", permits the retrieval of the information on a character by characters basis

for specified key words. UNAMEL is essentially upward compatible with the NAMELIST feature of FORTRAN. In the present context, UNAMEL is used to extract from the run statement those strings of characters which need to be tokenized.

The ULEX routine, developed by the first author with the help of K. Hoffman [6] of NBS, serves two purposes. First it sets up and maintains a "dictionary", that is, a repository of names, including special characters and digital strings, with associated "attributes" and tokens. The second purpose is to pick out names from a given string of characters, or rather, of "segmenting" this string into a sequence of names each of which occurs on the dictionary, where it is assigned a token. The sequence of names can thus be replaced by a sequence of tokens.

There are two techniques by which such a segmentation can be achieved. One, which is commonly referred to as "lexical analysis", uses a system of so called "delimiters", that is, special characters used for separating or bracketing, to indicate those substrings which correspond to names or symbols, and which are to be looked up in or entered into a dictionary, respectively.

ULEX is based on a different technique, which we call "lexical synthesis". The idea is to determine a segmentation of the given string such that all segments are names or symbols which are already on the dictionary. There may be many such segmentations of a given string. In order to reduce this ambiguity, each entry in the dictionary is assigned one or more "attributes". This set of possible attributes is specified along with a "legality matrix", which indicates for each pair of attributes whether it is acceptable for one of them to follow the other. It also indicates whether a particular attribute may start or terminate this sequence of words created by segmenting the given string. A segmentation which consists of words whose attributes form an acceptable succession pattern is considered "legal". By accepting only legal segmentations of a given string, ambiguities are for all practical purposes completely avoided.

In the absence of a uniqueness proof, however, the occurence of ambiguities remain a theoretical possibility to be reckoned with. ULEX therefore contains an algorithm which is capable of determining all legal segmentations of a given string, and is therefore also capable of determining whether there are more than one, or none at all. In either case, ULEX will terminate the tokenization process so that the user can adjust the wording of the run statement.

In the course of implementing ULP, both lexical analysis and lexical synthesis techniques are employed. ULEX has been applied to the character strings in CONSTRAIN, LPMAX, LPMIN and BOUND instructions. It can similarly be applied to instructions in the run statement starting with "LET".

The second stage of the implementation consists of a series of normalization steps. In the first such step, one collects digital strings, that is, strings of digital tokens, and converts them to numbers. Each such number is then represented by a single token. Quoted strings of alphanumerical tokens are similarly collected, interpreted as names of

constraints or indexes, and replaced by single tokens. Specification instructions (see Sections 1.7 and 2.5) operator words, and subrange constructs are processed subsequently and substituted into the constraint definitions if necessary. Finally, slack variables are introduced in order to achieve a form of the linear program in which all constraints other than bound constraints are equations. Each inequality constraint requires a slack variable which is added or subtracted depending on whether the inequality involves $\leq$ or $\geq$. New tokens are assigned to these slack variables and the tokenized form of the problem specification is appended, correspondingly. As was mentioned in Chapter 1, the slack variables are assigned names which arise from the names of their respective constraints by adding $ as a prefix.

We are now addressing the third stage of the implementation process. Most linear programming subroutines require that the linear program be represented by a "linear programming matrix" of, say, m rows and n columns, an objective row vector of length n, and a right hand side vector of length m. The question then arises how to generate these matrices and vectors from the problem specification. This is the matrix generation process that has been mentioned above. Software packages accomplishing it are commonly called "matrix generators".

The process of matrix generation will be discussed in detail in Section 4.8. Roughly speaking, it proceeds along the following lines. The numerical entries into the database are conceptually numbered in consecutive fashion. For each table entry, it is then possible to determine this "data number" from the portion of the table in the database, its ranges and their lengths, and finally the particular index combination which characterizes the entry within the table. Each instruction is now treated as follows. All ranges entering the instruction are identified and treated as a k-tuple of ranges. The corresponding k-tuples of indexes are consecutively generated in lexicographic order. For each such k-tuple of indexes, each term of the instruction creates an entry into the linear programming matrix (objective row or right hand sides) whose row and column number can be deduced from the name of the instruction and the associated variable. In this fashion, a list of "position triples"

$$(r, c, d)$$

is created, where r is a row number, c is a column number, and d is a data number.

These triples are sorted by an out-of-core sorting routine by increasing data numbers. Then a sequential pass through the database yields a numerical data entry u, creating a list of "value triples".

$$(r, c, u).$$

This list is again sorted, using the column numbers as the first key, and the row numbers as the second. The result is a column-sparse representation of the linear programming matrix. The creation of tables for the database from specification instructions requires an algorithm which is to some extent analogous to the matrix generation procedure outlined above.

The final issue is the interpretation and execution of data statements. For this purpose, a lexical analysis method with "single character look-ahead" is feasible and may well be preferable over the lexical synthesis approach. In any case, ULEX will be useful for the chore of dictionary maintainence. The actual database is considered as consisting of two sequential files, a "header file" containing the names and definitions of index ranges, index maps, and index range sequences along with reference information tying ranges to tables, index maps, and index range sequences. In addition, a list of references to external files (see Section 2.4) is provided as part of the headers. The second file, which we call the "content file", contains only numerical information. It is partitioned by end-of-file markers into "segments". At the end of each segment, except the last one, the data retrieval process switches to an external file as indicated in the header file.

The observation that matrix generation can be accomplished with two sorts and a single sequential pass through the database is due to an oral communication by M. Knapp-Cordes [9].

4.2 Tokenization of Run Statements: Generic Symbols

As was mentioned in the previous section, the ULEX subroutine package has been used for a restricted prototype version of the problem specification language. We will describe in this and subsequent sections how it might be used for the full version. As far as tokenization is concerned, this is simply a matter of expanding the legality matrix.

The first consideration when planning to apply ULEX is to choose a suitable set of attributes. The following attributes represented by their "attribute number" are suggested for the lexical synthesis of run statements:

2 = unknown	15 = additive
3 = table	16 = multiplicative
4 = range	17 = order relation
5 = map	18 = concatenation
6 = range sequence	19 = set subtraction
7 = open parenthesis	20 = range shift
8 = close parenthesis	21 = inversion
9 = open quote	22 = network
10 = close quote	23 = alphanumeric
11 = comma	24 = digit
12 = equivalence	25 = decimal point
13 = colon	26 = infinity
14 = sign	

Table 4.2.1: Attribute numbers

The first five attributes are assigned to names of problem-specific quantities. These names are collected either from the header-file of the database or from the run statement itself. The process of collecting these names will be discussed in more detail later in this section. The remaining attributes are assigned to words and symbols which are part of the problem specification language and are thus independent of the specific problem formulation. These symbols are entered into the dictionary immediately after the legality matrix has been entered. They are assigned negative token numbers. Table 4.2.2 lists these generic symbols.

The two different words, ">=" and "=>", have the same meaning and, therefore, the same tokens. The same is true for the words "<=" and "=>". Such words are known as "synonyms". Permitting synonyms makes the specification language more "user friendly". There are also examples of the same word being used with different connotations. The symbols 0,...,9 may be digits in a number or part of an alphanumeric name. In each case, they have a different meaning as indicated by their different attributes and tokens. The legality requirement is relied upon to chose the correct meaning when an ambiguous symbol or word is encountered. Such words with several different meanings are called "homonyms". Other instances of homonyms are the single quote, which may represent an open or close quote, the "+" and "-" signs, which may be sign or operation, and the symbol "=". The latter may stand for the equal sign in a constraint equation (token = -26) or indicate an assignment for the purpose of a definition (token = -17). The symbol "=" also occurs as part of a word, say, ">=". In this context, it is, however, not considered as a separate symbol. The lexical synthesis technique is particularly well suited to the handling of synonyms and homonyms.

4.3 Tokenization of Run Statements: Collecting Names

After the generic symbols (Table 4.2.2) have been put on the dictionary together with their attributes and tokens, one needs to enter the names of tables, ranges, index maps, and range sequences. Information from which the constraint names are to be constructed also needs to be processed.

A first source of names of these various kinds is the header file (see Section 4.1) of the database, which lists the names of tables, index ranges, index maps, range sequences and index names residing on the database. Those names are assigned positive integers as tokens on a first come first served basis with tokens from

1	to 99	for tables
101	to 199	for index ranges
201	to 299	for index maps
301	to 399	for range sequences
10001	to 19999	for index names.

Tokens 10001 and 101 are reserved for the nonindex (see Section 1.6) and the empty range (see Section 1.9), respectively.

Table 4.2.2

Generic symbols with attributes

Symbol	Name	Attribute #	Token #
(	open parenthesis	7	-11
)	close parenthesis	8	-12
'	open quote	9	-13
'	close quote	10	-14
,	comma	11	-15
=	equivalence	12	-16
:	colon	13	-17
+	plus sign	14	-21
-	minus sign	14	-22
+	addition operator	15	-23
-	subtraction operator	15	-24
*	multiplication operator	16	-25
=	equal sign	17	-26
>=, =>	greater or equal	17	-27
<=, =<	less or equal	17	-28
_	underline	18	-29
/	slash	19	-30
NEXT	advance operator	20	-31
PREVIOUS	retard operator	20	-32
INVERSE	inversion operator	21	-33
NETWORK	network operator	22	-34
A,...,Z,0,...9	alphanumeric character	23	-41,...,-76
0,..,9	digit	24	-81,...,-90
.	decimal point	25	-91
INF	infinity symbol	26	-99

The token numbers for tables provide also a set of "table numbers" running from 1 to 99 (this convention is readily changed if the limit of 99 tables in the database should be too restrictive). Similarly, by deleting the first digit of the token numbers, "range numbers", "map numbers", "range sequence numbers", and "index numbers" ranging from 1 to 99 or 1 to 9999, respectively, are generated. The index number 1 is the number of the non-index. These numbers are used as pointers in internal arrays containing information pertinent to tables, ranges, indexes, etc. In particular, arrays are set up that translate the running numbers into word numbers on the dictionary.

The names of the tables, index ranges, index maps and range sequences are entered into the ULEX dictionary together with their attribute and token numbers. Attribute numbers are assigned according to Table 4.2.1. The index names are treated differently. They are entered into the ULEX dictionary with no attribute (or dummy attribute).

In addition, data structures are set up which associate with each range, identified by its range number, its subordinate indexes, identified by their index number. The subordinate ranges of each table, the image and domain ranges of maps, and the universal and domain ranges of range sequences are listed for reference in similar data structures. There are various simple ways to set up such data structures. Their discussion, however, would extend beyond the scope of this report. We refer to the collection of these data structures as the "cross-reference list".

The actual map information, that is, the assignment of image indexes as a function of the domain indexes is also transfered from the header file to a special "map list". The actual descriptions of range sequences residing in the database are similarly collected in a "range sequence list".

The next source of names is provided by specification instructions. Each of these instructions is characterized by the key-word "LET". The portion of the string that precedes the first colon contains the name of the quantity to be specified and, in the case of tables, index maps, and range sequences, the name of subordinate ranges. Since its subordinate ranges need to be recognizable before a table, map, or range sequence can be specified, LET RANGE instructions are processed first.

Now is the time to process the UNKNOWN instruction and enter the names of the unknown variables into the dictionary. As token range we propose

401 to 499 for unknown names.

This does not imply that linear programs which are formulated in this way may have only 99 variables, since the unknown names may be indexed. Again dropping the first digit of the token number of an unknown variable will produce the "variable number" by which the unknown variable will be referred to in the cross-reference repository. Also, the range numbers of ranges subordinate to the unknown variable will be listed in the cross-reference repository. An array will be set up which assigns to each variable number a word number on the dictionary.

Variable names correspond to columns of the linear programming matrix. "Constraint names" similarly identify the rows of the latter. As explained in Sections 1.2 and 1.3, contraint names may be tuples of indexes or a specified alphanumeric string -- the constraint preface -- followed by a tuple of indexes in parentheses. The constraint preface is given by an alphanumeric string in quotes, followed by a tuple of range names in parentheses, all contained in the instruction header of the CONSTRAIN instruction in question. Index names are drawn automatically from these ranges and run lexicographically through all combinations. In addition, one of the ranges may be replaced by an alphanumeric string in quotes. This string is treated as a solitary index name, and is therefore called a "pseudoindex". A scanning of the instruction headers of all CONSTRAIN statements, yields a "constraint name list" of constraint prefaces and pseudoindexes: A "constraint list" indicates the fixed ranges associated with each constraint.

4.4 Tokenization of Run Statements: Lexical Synthesis

The first, that is, left-most colon in the CONSTRAIN instructions and the specification instructions, that is, the instructions starting with "LET", divides these instructions into the "instruction header" and the "instruction body". The instruction body starts with the first nonblank character after the colon, except if this character is "=", in which case the instruction body starts after the "=". BOUND, LPMIN and LPMAX instructions contain no colon, and therefore have no header. In these cases, the instruction body is the entire string between the open parenthesis directly following the keyword and its matching close parenthesis.

The lexical synthesis technique is now applied to the instruction body of each LPMIN, LPMAX, CONSTRAIN, BOUND, LET RANGE, LET TABLE, LET MAP and LET RANGE SEQUENCE instruction. This is achieved by purging these strings of unnecessary blank characters and then partition them into legal segmentations using ULEX routines. If all strings have unique segmentations, then this procedure will terminate with a sequence of token strings, one for each processed instruction. This sequence of token strings represents the tokenized form of the run statement.

For the above lexical synthesis procedure, we propose the legality matrix shown in Table 4.3.1. Each entry in this table has the value "true" or 'false" as indicated by "T" and "-", respectively. The first row indicates the attributes which may follow "BOS", that is, the beginning of the string. Similarly, the first column indicates the attributes which may precede "EOS", that is, the end of the string. By this convention, we characterize the attributes which may start or terminate the string of words into which an instruction body is segmented.

For most entries of value "true", we know of examples in which the corresponding succession of attribtues actually occurs. In addition, we have given the value "true" to some entries where we did not feel absolutely certain that the corresponding succession of attributes could be ruled out, or which might occur in natural extensions of the language. We have circled

Table 4.4.1

Legality matrix

	EOS	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
	BOS	–	T	T	T	T	T	T	–	T	–	–	–	–	T	–	–	–	–	–	T	T	T	–	T	T	–
unknown	2	–	–	–	–	–	–	T	–	–	–	–	–	–	–	T	–	T	–	–	–	–	–	–	–	–	–
table	3	T	–	–	–	–	–	T	–	–	–	T	T	T	–	–	T	T	–	–	–	–	–	–	–	–	–
range	4	T	–	–	–	–	–	T	T	–	–	T	–	–	–	–	–	–	T	T	–	–	–	–	–	–	–
map	5	–	–	–	–	–	–	T	T	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
range sequence	6	–	–	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
open parenthesis	7	–	–	–	T	T	T	–	T	T	–	–	–	T	–	T	T	T	–	–	T	–	–	–	–	–	–
close parenthesis	8	T	–	–	T	–	–	T	T	–	–	T	T	T	–	T	T	T	–	T	–	–	–	–	–	–	–
open quote	9	–	–	–	–	–	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	T	–	–	–
close quote	10	T	–	–	–	–	–	T	–	–	–	T	T	–	–	–	–	–	T	–	–	–	–	–	–	–	–
comma	11	–	–	T	T	–	–	–	–	T	–	–	T	–	T	–	–	–	–	–	T	–	T	–	T	T	T
equivalence	12	–	–	T	–	–	–	–	–	T	–	T	–	–	T	–	–	–	–	–	–	T	–	–	T	T	T
colon	13	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
sign	14	–	T	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	T	–	–	T	T	T
additive	15	–	T	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	T	–	–	T	T	–
multiplicative	16	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
order relation	17	–	–	T	–	–	–	–	–	–	–	–	–	T	–	–	–	–	–	–	T	–	–	T	T	T
concatenation	18	–	–	–	T	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
set subtraction	19	–	–	–	T	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	T	–	–	–	–	–
range shift	20	–	–	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
network	21	–	–	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
inverse	22	–	–	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
alphanumeric	23	–	–	–	–	–	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	T	–	–	–
digit	24	T	–	–	–	–	–	–	–	–	–	T	–	–	–	–	T	–	–	–	–	–	–	–	T	T	–
decimal point	25	T	–	–	–	–	–	–	–	–	–	T	–	–	–	–	T	–	–	–	–	–	–	–	T	–	–
infinity	26	T	–	–	–	–	–	–	–	–	–	T	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

these entries in Table 4.4.1. We believe that relying on the principle, when in doubt, choose "true", does not jeopardize the usefulness of the legality matrix for rejecting illegal segmentation.

APPEND and DELETE instructions (see Section 2.7) have such a simple structure that they can be processed directly without the relatively involved lexical synthesis technique.

4.5 Normalizing of the Tokenized Form: Slacks and Substitutions

As indicated in the overview, a series of normalization steps is now applied. In the first of these steps, strings of digits are identified in the sementation and converted to numbers using a decimal to binary conversion routine. The numbers are assigned tokens in the range from

501 to 599.

Similar to the token assignments in Section 4.2, deleting the first digit of the token number will yield the "constant number" under which the constants are filed in a "constant list" .

There are other straightforward substitutions, which can be made in the entire set of token strings. First, set subtraction constructs can be replaced by equivalent generalized subrange constructs (see Section 2.6). Second, in LET RANGE and LET RANGE SEQUENCE instructions, where indexes defining ranges are directly specified by sequences of quoted alphanumeric strings, the separating commas (token = -15) are replaced by the concatenation operator "_" (token = -29).

The next step involves the introduction of slack variables for inequality constraints. These variables are added to the list of unknowns. Their names are generated by letting a "$" sign precede the constraint preface (see Section 4.3). The new token numbers are consecutive starting with the first unused token number. A new term is then entered into the token string just before the order relation symbol. This term consists of the unknown token followed -- unless the CONSTRAIN statement does not specify a fixed range -- by open parenthesis (token = -11), by the tuple of fixed ranges from the constraint list (see Section 4.3) represented by their respective tokens and separated by commas (token = -15), and finally by a close parenthesis (token = -12). The new term is preceded by "+" (token = -23) or "-" (token = -24) depending on whether the order relation is characterizd by "<=" (token = -28) or ">=" (token = -27), respectively. In the case of constraint equations, no slack variable is added. Finally, all order relation symbols in CONSTRAIN statements are changed to "=" (token = -26).

4.6 Normalization of Token Strings: LET Instructions

In this section, we describe what needs to be done concerning LET instructions. As was pointed out before (Section 2.6), these are instructions by which new quantities, that is, tables, index ranges, index maps and range sequences are introduced. The LET instruction may represent a definition "from scratch" or a definition which relies on previously defined quantities. In the latter case, the name of the quantity to be defined appears in the instruction header and the names of the previously defined quantities on which the definition is based appear in the instruction body. We call the latter quantitites the "precursors" of the quantity to be introduced.

Quantities are now being ranked. A quantity has "rank 0" if it is defined in the database. It has "rank 1" if it is defined from scratch, that is, without precursors, or if all its precursors are of rank 0. A quantity has "rank 2" if it is not of rank 1 and all its precursors have a rank which is less than 2, and so on. The LET instructions themselves are thus ranked implicitly: the rank of a LET instruction is the rank of the quantity it defines. If the process of ranking LET instructions terminates before all LET instructions have been ranked, then a circular definition is present, and an error stop ensues.

If all LET instructions are successfully ranked, then they are processed in this order: first all LET instructions of rank 1, then those of rank 2, and so on. In this fashion, each LET instruction will be processed only after all its precursors have been processed already. We now consider individual LET constructions.

After the substitutions of Section 4.5, the LET RANGE instructions are of three kinds: (i) concatenation instructions; (ii) subrange instructions; (iii) range shift instructions. In the first case, a mixed sequence of ranges -- already defined -- and quoted alphanumeric strings representing new indexes is encountered. The string of indexes for the concatenated range is readily extracted and processed in the same way as a range definition that had been transferred from the database. In the second case, an index sequence is generated starting with the sequencing range, the resulting indexes are transformed using either a predefined index map or a subrange construct with the universal range as screening range. In addition, index range sequences may be defined by applying the INVERSE operator to a precursor quantity which is either an index range sequence or an index map. There are straightforward algorithms available for the above inversion.

LET MAP instructions are similar to the LET RANGE and LET RANGE SEQUENCE in that they relate indexes to one another. No LET MAP instructions other than definitions from scratch have been included so far into the specification language. We propose to handle these definitions just as if they originated from the database.

This is the general philosophy for this implementation proposal: process each index range, index map and index range sequence as if the information originated index by index from the database. The LET RANGE, LET MAP, LET RANGE SEQUENCE instructions simply supplement the database.

LET TABLE instructions are a different story, because precursor tables may be stored in the database proper and not in the header file of the database. Not only is such table information potentially too large to copy into core storage, but retreiving this information would require an extra pass through the database. Tables which are defined numerically from scratch, including using the NETWORK operator, do not pose this problem. The procedure is to store modification information in a "table modification list" from which it can be retrieved at matrix generation time. The vehicle used for such modification are the "conditional position triples" discussed in Section 4.8. Aggregation utilizes "multiple position triples" also discussed in Section 4.8. For "position triples" see Section 4.1.

4.7 Normalizing Token Strings: Indexes and Subranges

The final normalization steps concerning token strings representing CONSTRAIN, LPMIN, LPMAX and BOUND instructions can now be described. Note that all token strings representing LET instructions have disappeared.

Essentially two tasks remain. The first concerns the identification of indexes in "index position". These index positions indicate arguments after table or range sequence names and are part of a string separated by commas and enclosed in parentheses. For each index position, there exists a unique range such that indexes found in this particular index position must belong to that range. Whenever a quoted alphanumeric string is found in an index position, it is compared with the index names of the respective range and, if the matching index name is found, is replaced by the corresponding index token.

Two kinds of subrange constructs remain to be processed. First, there are the direct subrange constructs and those with explicitly specified index correspondences. Second there are generalized subrange constructs referring to an index map. For both kinds of subrange construct, we generate an index map with the sequencing range as domain and the screening range as image range with the nonindex admitted as value. This map assigns to every index in the sequencing range either an index in the screening range or the nonindex, depending on whether a skipping of the term is intended. This index map is entered into the map list (see Section 4.3) and assigned a token. The entire subrange construct is then replaced by the map token, followed by open parenthesis (token = -11), the token of the sequencing range, and close parenthesis (token = -12). The fully normalized form of the token strings has been reached.

4.8 Matrix Generation

We proceed to describe the matrix generation process in more detail than in Section 4.1. The matrix to be generated consists of the constraint coefficients together with the coefficients of the objective function and the constants on the right hand side. The first task is to define the rows and the columns of the matrix.

To this end, we order the variable names in a linear fashion. This defines a linear order for the single unknowns if we assume a lexicographic ordering of the single unknowns associated with each table name, respectively.

The single unknowns are then numbered consecutively in this order, and this number becomes the "column number" of the linear programming matrix to be generated. A column number of -1 is associated with the "right-hand-side column". For each variable name, the lowest of the column numbers of single unknowns associated with this variable name is of interest. The column number just prior to this number, that is, the lowest column number minus one, is stored for each variable name. This permits straightforward calculation of the column number from the variable name and the particular index setting.

The CONSTRAIN instructions are similarly ordered, and the single constraints generated by each CONSTRAIN instruction are again lexicographically ordered, with respect to the sequence of fixed ranges in the CONSTRAIN instruction. This leads to a consecutive numbering of all single constraints and thereby to a definition of "row numbers" for the linear programming matrix. Again a row number "just prior" to the row numbers associated with a particular CONSTRAIN instruction is stored, permitting simple calculation of row numbers from the constraint name and the index setting for the fixed ranges. This calculation is analogous to the calculation of column numbers.

Finally, a data number (see Section 4.1) is derived by conceptually numbering the numerical entries in the database. Since table entries are stored lexicographically (see Section 2.3), knowledge of the sequence of the tables in the database permits the calculation of the data number from table name and index setting in fashion similar to the calculation of row and column numbers.

The first phase of the matrix generation process row considers the CONSTRAIN instructions in sequence. Each CONSTRAIN instruction is divided by the additive operators "+" (token = -23), "-" (token = -24), and the equation symbol "=" (token = -26) into "terms". .The last of these terms, the one following "=", is called the "right-hand-side term". All terms except the right-hand-side term consist either of a "table clause" and a "variable clause" separated by "*" (token = -25) or simply of a variable clause. The right-hand-side term consists of a table clause only.

A table clause is either a constant, or a table name, or a table name followed by a mixed tuple of range, mapped range, or index specifications separated by commas. The table name may be a "primary table name", that is, refer to a table on the database, or a "secondary table name" introduced via a LET instruction. A variable clause consists of a variable name also followed by a tuple of range, mapped range, or index specification, separated by commas. Terms consisting of a variable clause only are equivalent to terms with a table clause which consists of the constant 1.0, and are treated correspondingly.

For each term, we consider the sequence of ranges which consists first of the fixed ranges of the CONSTRAIN instruction to be followed by the ranges which occur in the variable clause of the term, but are not one of the above mentioned fixed ranges. These ranges are followed in turn by ranges which are not fixed and occur only in the table clause. We call these ranges "aggregation ranges". The entire sequence of ranges is the "driving sequence" of the term. The matrix generation algorithm now runs lexicographically through all index combinations in the driving sequence.

Each such index combination is called an "index signature". Together with the constraint name and the variable name, it determines a row number r and a column number c. Consider now a range in the mixed sequence of ranges, mapped ranges, and indexes following the table that is part of the table clause. This sequence may, of course, be empty. In this case, the table index signature is also empty. If not, then a range in the mixed sequence is also represented in the driving sequence and a particular index in the index signature corresponds to it. In the case of a mapped range, the domain of the index map is represented in the driving sequence. The corresponding index is transformed by the index map into either a non-index, in which case one moves on to the next index signature without taking further action, or a valid index in the corresponding range of the table is produced. Finally, the index for some range of the table may have been specified at the outset. In any case, we wind up with a tuple consisting of a table name, and a tuple of indexes, each from one of the ranges of the table. Thus all the information for a table look-up is available. This information is therefore called a "table look-up signature". If the table is a primary table, then a data number d can be calculated from this table look-up signature, and we can form the position triple (Section 4.1)

$$(r, c, d),$$

and add it to the list of previously determined position triples. If the table clause refers to a constant, then the negative constant number is entered as the data number. If the term was a right-hand-side term, then $c = -1$.

The effect of aggregation ranges is that several position triples with identical row and column numbers but different data numbers are generated. We call such position triples "multiple position triples". Such triples specify several numbers for the same matrix location. By considering the sum of these numbers as the definitive entry, we effect aggregation.

The case of secondary table names remains. In this case, the table name refers to information on the table modification list. This information is such that depending on the indexes in the look-up signature, a new table clause is substituted. This new table clause may introduce new aggregation ranges. The indexes in the look-up signature together with the indexes from aggregation ranges determine a new table look-up signature with respect to the new table clause. If the latter is primary or constant, then again a position triple can be determined. Otherwise, the procedure is repeated until a primary or constant table clause is reached. We call position triples that are derived in this fashion "conditional position triples".

As mentioned in Section 4.1, the second phase of the matrix generation process starts with sorting of the position triples by data number. A sequential pass through the database will replace the data number by an actual numerical value, generating a list of value triples (Section 4.1)

$$(r, c, u).$$

This list is sorted again, primarily by the column numbers c and secondarily by the row numbers r. Finally, the value triples for which both row and column numbers agree -- these value triples are now found in consecutive positions on the list -- are replaced by a single value triple whose value is the sum of their values. After this aggregation step, a column-sparse representation of the linear programming matrix has been achieved.

4.9 Processing of Data Statements

For the processing of data statements we recommend several lexical analysis procedures applied in succession. These procedures are using typically a single character "look-ahead" and are readily represented by "finite state machines". This process will produce an intermediary data structure which consists of "header information" and "digital information". The digital information is converted into binary numbers and stored on the content file in segments separated by end-of-file markers (see Section 4.1). The header information will be transferred to the header file (see Section 4.1) of the database.

The intermediate digital information is divided into 10 segments anticipating the subsequent partition of the database. More precisely, the intermediate digital information consists of a sequence of signs, digits, decimal points, commas, and at-signs (= "@"). Commas separate digital number designations, using signs, digits and decimal points. The at-signs replace commas as separators at larger intervals. A list indicates for each at-sign either the number of zero entries which are to precede it in the content file or it indicates the number of entries to be read from a particular external file. The at-signs of the latter kind divide the string into segments which correspond to the segments on the content file.

We will very briefly sketch one of many possible approaches to the problem of extracting the intermediate digital information from the data statement. A very first step would be to find the end of the data statements, that is, a line of all blank characters or an end-of-file matrix. This is a straightforward procedure and, in the context of UTPS, it is performed automatically by the UTPS utility routine SIGNON. The next step is to remove all comment lines. Again this is straightforward. A simple approach involving four "finite states" can be used to remove all comment clauses and to produce a character string without ends-of-lines. Superfluous blank characters can be squeezed out concurrently or in yet another separate procedure.

The result is an edited data statement which is now truly a string of characters rather than a sequence of cards. Its superfluous blanks have been removed. The string is partitioned by semicolons into blocks. Text clauses remain as terminating portions of their respective blocks.

The type of each block is readily identified. Indeed, each block starts with an alphaname (see Section 2.1). If the alphaname is followed by a colon, it is either a range block, a file block or a singleton table block. Range

blocks are recognized by a quote following the colon. File blocks are characterized by the number sign "#" following the colon. If the alphaname is not followed by a colon, then it must be followed by an open parenthesis. If the matching close parenthesis is followed by a colon or a slash, then the block is a table block. If the first open parenthesis is followed immediately by a close parenthesis, then the block is a range sequence block. Finally if the first pair of parentheses enclose an alphaname and are followed by an alphaname, then the block is a map block.

Once the type of a block is determined, its constituents are readily identified. Some of these, particularly range names and file designations must have been already defined and entered into the dictionary. The new name, that is, the name introduced by the block in question, is then itself entered into the dictionary. Every type of block contributes to header information; but only table blocks provide the basis for the intermediate digital information, which is an edited sequence of table body information.

APPENDIX A: Comparison with Other Problem Specification Methods

Using two sample problems, we will compare the problem specification method of ULP with those of XML (a proposed modeling language) and three linear programming packages -- OMNI, DATAFORM, and LPMODEL. XML is designed to formulate problems in a systematic way, but does not represent a form of computer input. We will utilize it to restate the sample problems. No data entry has been included with the XML example; the problem is specified in a general form. OMNI and DATAFORM are matrix generator languages in which the user must specify the linear programming matrix a column at a time. LPMODEL has a philosophy similar to that of ULP, where the user does not need to think in terms of a matrix at all, but formulates the problem in a more natural way. It also separates the problem specification from the data input, as ULP does. (The following problem formulations have not actually been tested with the different packages, but were developed by reading user manuals.)

EXAMPLE 1

This problem is the same as EXAMPLE 1 from Section 1.1.

(1) EXAMPLE 1 - XML (Fourer [4])

SETS

metals		
alloys		

PARAMETERS

supply	INDEXING:	OVER metals
	ATTRIBUTES:	nonnegative
	COMMENTS:	supply[i] is supply of metal i
price	INDEXING:	OVER alloys
	ATTRIBUTES:	nonnegative
	COMMENT:	price[j] is price of alloy j
comp	INDEXING:	OVER metals, OVER alloys
	ATTRIBUTES:	nonnegative
	COMMENT:	comp[i,j] is units of metal i per unit of alloy j

VARIABLES

x	INDEXING:	OVER alloys
	ATTRIBUTES:	nonnegative
	COMMENT:	x[j] is units of alloy j to be produced

OBJECTIVE

profit	ATTRIBUTES:	maximize
	SPECIFICATION:	SIGMA j OVER alloys (price[j]*x[j])

CONSTRAINTS

suplim	INDEXING:	i OVER metals
	SPECIFICATION:	SIGMA j OVER alloys (comp[i,j]*x[j]) = supply[i]
	COMMENT:	Amount of metals used must not exceed supply limit.

(2) EXAMPLE 1 - OMNI (Boudrye [3])

```
DICTIONARY

    CLASS METALS

          METAL1

          METAL2

    CLASS ALLOYS

          ALLOY1

          ALLOY2

          ALLOY3

          ALLOY4

DATA

    TABLE SUPPLY

                    TONS

          METAL1    6

          METAL2    5

    TABLE PRICE

                    DOLLARS

          ALLOY1    1000

          ALLOY2    1500

          ALLOY3    1800

          ALLOY4    4000

    TABLE COMP

                    ALLOY1   ALLOY2   ALLOY3   ALLOY4

          METAL1      .5       .6       .3       .1

          METAL2      .5       .4       .7       .9
```

```
      FORM ROW ID

        PRICE = OBJ

        (METALS) = FIX

COLUMNS

      FORM VECTOR (ALLOYS)

        PRICE = TABLE PRICE((ALLOYS),DOLLARS)

        (METALS) = TABLE COMP((ALLOYS),(METALS))

RHS

      FORM VECTOR RHSIDE

         (METALS) = TABLE SUPPLY((METALS),TONS)

ENDATA
```

In OMNI, all tables must be two-dimensional, and names must be provided for the rows and columns.

In the ROWS section, each row of the matrix is named and given a type, i.e. objective function (OBJ), equality constraint (FIX), less-than-or-equal-to constraint (MAX), or greater-than-or-equal-to constraint (MIN).

The matrix is generated a column (i.e. vector) at a time. Each column is named. The class name ALLOYS in parentheses means that the statement must be repeated for each member of the class. For a fixed column name, table values are then placed in rows named PRICE, METAL1, and METAL2. The right-hand side vector is generated in a similar way.

(3) EXAMPLE 1 - DATAFORM (Ketron [8])

```
*DATA DEFINITION SECTION
        TABLE   SUPPLY = TONS
                METAL 1 = 6
                METAL 2 = 5
        TABLE   PRICE = DOLLARS
                ALLOY 1 = 1000
                ALLOY 2 = 1500
                ALLOY 3 = 1800
                ALLOY 4 = 4000
        TABLE   COMP = ALLOY 1, ALLOY 2, ALLOY 3, ALLOY 4
                METAL 1 = .5, .6, .3, .1
                METAL 2 = .5, .4, .7, .9
*PROBLEM GENERATION SECTION
*GENERATE THE MATRIX
        COL     T:COMP(0,$1), PRICE = T:PRICE($1,1),
                T:COMP($2,0) = T:COMP($2,$1)
*GENERATE THE RIGHT-HAND SIDE
        RHS     SUPPLY, T:SUPPLY($1,0) = SUPPLY($1,1)
*IDENTIFY THE OBJECTIVE FUNCTION
        ROW     PRICE <N>
```

DATAFORM is very similar to OMNI in its method of problem specification; the two languages differ mainly in details of syntax.

As in OMNI, tables in DATAFORM must be two-dimensional with names for each row and column.

The matrix is generated a column at a time using the COL statement. This statement has the following form:

COL column name, row name = value, row name = value,...

In this example, the column names are specified by T:COMP(0,$1), which symbolizes the element in row 0 and column $1 of table COMP. Row 0 contains the column headings of the table and $1 is a loop control which causes the COL statement to be repeated for each column heading. In this example, the columns will be named ALLOY 1, ALLOY 2, ALLOY 3, and ALLOY 4. Next, for a given column name, a value from the PRICE table is placed in a row named PRICE and values from the COMP table are placed in rows named METAL 1 and METAL 2. (The $2 loops over the row names of the COMP table.)

The right-hand side is generated separately using the RHS statement:

RHS rhs name, row name = value

In this case, the right-hand side is named SUPPLY, and values from the SUPPLY table are put in rows METAL 1 and METAL 2.

All rows of the matrix are assumed to represent equality constraints, unless stated otherwise. In this example, the row named PRICE is identified as the objective function (N means "no restriction").

(4) EXAMPLE 1 - LPMODEL (Katz [7])

```
ALLOYS ← ALLOY_1, ALLOY_2, ALLOY_3, ALLOY_4

METALS ← METAL_1, METAL_2

SUPPLY.METALS ← 6  5

PRICE.ALLOYS ← 1000  1500  1800  4000

COMPOSITION.ALLOYS.METALS ← .5 .5 .6 .4 .3 .7 .1 .9
```

```
SUM [ALLOYS: COMPOSITION.ALLOYS.METALS x ALLOYS? = SUPPLY.METALS]

MAXIMIZE [ALLOYS: PRICE.ALLOYS x ALLOYS?]
```

LPMODEL is similar to ULP in that the user does not specify the matrix column by column, but uses a concise and natural algebraic notation to formulate the constraints and objective function of the problem. Also, the data input is independent of the problem formulation, and the data can be thought of as coming from some already-existing database.

In line 6, "SUM [ALLOYS:" symbolizes that a summation is to be done over all elements of the class ALLOYS. The statement is repeated for each element of the fixed class METALS, leading to two constraints. Variables are represented by names ending with a question mark.

Line 7 specifies the objective function in a similar way, where again the summation is over the elements of the ALLOYS class.

A key difference between LPMODEL and ULP is in the representation of summations. LPMODEL writes an index range followed by a colon to indicate that a summation is to be done over the elements of that range. Any other range appearing in the statement is fixed, that is, the statement is repeated for each element of the range. *P uses the opposite convention -- a range followed by a colon indicates a fixed range and all other ranges are to be summed over. This has the advantage of providing a name for each constraint, which in turn supplies a name for each slack variable.

ULP anticipates two different languages for the database specification and for the problem specification. This emphasizes that the database may exist for other purposes than a particular linear programming application. However, it also provides the capability to input the data directly with the problem formulation. This can be done using the LET instructions of Section 2.6. We will illustrate both options.

(5a) EXAMPLE 1 - ULP

Database specification:

```
*RANGES

 METALS: 'METAL '(1-2);

 ALLOYS: 'ALLOY '(1-4);

*TABLES

 SUPPLY(METALS): 6 5;

 PRICE(ALLOYS): 1000 1500 1800 4000;

 COMPOSITION(METALS, ALLOYS): .5 .6 .3 .1
                              .5 .4 .7 .9
```

Problem Specification:

```
UNKNOWN(X(ALLOYS))

  COMMENT(X(ALLOYS)=AMOUNT PRODUCED OF EACH ALLOY)

LPMAX(PRICE(ALLOYS)*(ALLOYS))

  COMMENT(MAXIMIZE REVENUES)

CONSTRAIN(METALS: COMPOSITION(ALLOYS,METALS)*X(ALLOYS)=SUPPLY(METALS))
```

(5b) EXAMPLE 1 - ULP

Combined Problem and Data Specification:

```
UNKNOWN(X(ALLOYS))
  COMMENT(X(ALLOYS)=AMOUNT PRODUCED OF EACH INDIVIDUAL ALLOY)
LPMAX(PRICE(ALLOYS)*X(ALLOYS))
  COMMENT(MAXIMIZE REVENUE)
CONSTRAIN(METALS: COMPOSITION(ALLOYS,METALS)*X(ALLOYS)=SUPPLY(METALS))
COMMENT(DATA)
LET RANGE(METALS: 'METAL '(1- 2)
LET RANGE(ALLOYS: 'ALLOY '(1-4)
LET TABLE(SUPPLY(METALS): 6., 5.)
LET TABLE(PRICE(ALLOYS): 1000, 1500, 1800, 4000)
LET TABLE(COMPOSITION(METALS,ALLOYS): .5, .6, .3, .1,
                                      .5, .4, .7, .9)
```

EXAMPLE 2

This example gives a more complex problem (presented in Fourer[4]) to compare XML, OMNI, LPMODEL, and ULP. DATAFORM has been omitted since it is long and basically similar to OMNI.

Problem: A factory manufactures three different products (high quality, medium quality, and low quality) in each of three different production periods. Two raw materials (scrap and new) are required for each product. The following data are given:

(1) Each product returns a certain profit, which may vary for the different production periods.

Profit (per unit)

Product	Period 1	Period 2	Period 3
Low	25	20	10
Medium	50	50	50
High	75	80	100

(2) At most 40 total units can be produced in any period.

(3) Each unit of a product requires a certain number of units of the two raw materials.

Composition

Raw Material	Low	Medium	High
Scrap	5	3	1
New	1	2	3

(4) Each raw material has a fixed storage cost per unit per period.

Raw Material	Storage Cost
Scrap	0.5
New	2.0

(5) Any raw material left unused after the last production period has an estimated remaining value per unit.

Raw Material	Remaining Value
Scrap	15
New	25

(6) A certain initial stock of the raw materials is available to be used over the three periods.

Raw Material	Initial Stock
Scrap	400
New	275

How much of each product should be produced in each period in order to maximize the net profit, adjusted for storage costs and the remaining value of the raw materials?

(1) EXAMPLE 2 - XML (Fourer [4] - p. 176)

SETS

prod	COMMENT:	set of products
raw	COMMENT:	set of raw materials

PARAMETERS

time	ATTRIBUTES:	positive integer
	COMMENT:	number of production periods
max	ATTRIBUTES:	positive
	COMMENT:	maximum total unit production per period
a	INDEXING:	OVER raw, OVER prod
	ATTRIBUTES:	nonnegative
	COMMENT:	a[i,j] is units of raw material i needed to manufacture one unit of product j
b	INDEXING:	OVER raw
	ATTRIBUTES:	nonnegative
	COMMENT:	b[i] is max initial stock of raw material i
c	INDEXING:	OVER prod, FROM 1 TO time
	COMMENT:	c[j,t] is estimated profit per unit of product j in period t
d	INDEXING:	OVER raw
	COMMENT:	d[i] is storage cost per period per unit of raw material i
f	INDEXING:	OVER raw
	COMMENT:	f[i] is estimated remaining value per unit of raw material i after last period

VARIABLES

x	INDEXING:	OVER prod, FROM 1 TO time
	ATTRIBUTES:	nonnegative
	COMMENT:	x[j,t] is units of product j manufactured in period t
s	INDEXING:	OVER raw, FROM 1 TO time+1
	ATTRIBUTES:	nonnegative
	COMMENT:	s[i,t] is stock of raw material i at beginning of period t

OBJECTIVE

profit	ATTRIBUTES:	maximize
	SPECIFICATION:	SIGMA t FROM 1 TO time (SIGMA j OVER prod (c[j,t]*x[j,t]) -SIGMA i OVER raw (d[i]*s[i,t]))
	COMMENT:	total over all periods of estimated profit less storage cost, plus value of remaining raw materials after last period

CONSTRAINTS

limit	INDEXING:	t from 1 TO time
	SPECIFICATION:	SIGMA j OVER prod (x[j,t]) <= max
	COMMENT:	total unit production per period must not exceed maximum
init	INDEXING:	i OVER raw
	SPECIFICATION:	s[i,1] <= b[i]
	COMMENT:	stock for period 1 must not exceed maximum
bal	INDEXING:	i OVER raw, t FROM 1 TO time
	SPECIFICATION:	s[i,t+1] = s[i,t] - SIGMA j OVER prod (a[i,j]*x[j,t])
	COMMENT:	stock for next period equals stock for present period less raw materials used in present period

(2) EXAMPLE 2 - OMNI (Fourer [4] - p. 177)

```
DICTIONARY

    CLASS T              Set of production periods
      1,2,3

    CLASS U              Set of periods preceding production periods
      0,1,2

    CLASS V              Extra period after last production period
      4

    CLASS W              Period preceding extra period
      3

    CLASS PRD            Set of products:
      LOW                  low quality product
      MED                  medium quality product
      HIH                  high quality product

    CLASS RAW            Set of raw materials:
      SCR                  scrap raw material
      NEW                  new raw material

DATA

    TABLE M              Maximum total unit production per period
        MAX
    MAX 40

    TABLE A              Units of each raw material needed to manufacture one
*                        unit of each product
          LOW  MED  HIH
    SCR    5    3    1
    NEW    1    2    3


    TABLE B              Maximum initial stock of each raw material
          MAX
    SCR   400
    NEW   275


    TABLE C              Estimated profit per unit for each product in each
*                        period
           1     2     3
    LOW   25    20    10
    MED   50    50    50
    HIH   75    80   100
```

```
    TABLE D              Storage cost per unit for each raw material
           STOR
    SCR     0.5
    NEW     2.0


    TABLE F              Remaining value for each raw material after last
*                        period
           VALUE
    SCR      15
    NEW      25


FORM ROW ID

*Maximize total over all periods of estimated profit less storage cost, plus
*value of remaining materials after last period.
        OBJ = OBJ

*Balance production and stock of each raw material in each period.
        BAL(RAW)(T) = FIX

*Limit total production in each period.
        CAP(T) = MAX


COLUMNS

*Manufacturing activity, for each product in each period:
         FORM  VECTOR  X(PRD)(T)

*        Consumption of each raw material, by each product in each period:
            BAL(RAW)(T) = TABLE A((PRD),(RAW))

*        Consumption of capacity, in each period:
            CAP(T) = 1

*        Estimated profit, from each product in each period:
            OBJ = TABLE C((T),(PRD))

*Stockpiling activity, for each raw material in each period:
         FORM VECTOR S(RAW)(T)

*        Production of each raw material from stocks, start of each period:
            BAL(RAW)(T) = -1

*        Consumption of each raw material into stocks after after each period:
            BAL(RAW)(U/T) = 1 EXCEPT T=1

*        Storage costs, for each raw material in each period:
            OBJ = - TABLE D(STOR,(RAW))
```

```
*Stockpiling activity, for each raw material after last period:
         FORM VECTOR S(RAW)(V)

*        Consumption of each raw material into stocks after last period:
            BAL(RAW)(W/V) = 1

*        Remaining value, for each raw material after last period:
            OBJ = TABLE F(VALUE,(RAW))

RHS

    FORM VECTOR RHSIDE

*   Production capacity in each period:
       CAP(T) = TABLE M(MAX,MAX)
*   (Note:right-hand sides for balance rows are all zero.)


BOUNDS

    FORM BOUNDS MAXSTOCK
*   Limit on stock of each raw material, in first period:
       S(RAW)1,UP = TABLE B(MAX,(RAW))


ENDATA
```

(3) EXAMPLE 2 - LPMODEL (Katz et.al [7])

```
PRODUCTS ← LOW,MEDIUM,HIGH

MATERIALS ← SCRAP,NEW

PERIODS ← PERIOD_1,PERIOD_2,PERIOD_3

PROFIT.PRODUCTS.PERIODS ← 25 20 10 50 50 50 75 80 100

MAXIMUM_PRODUCTION ← 40

COMPOSITION.MATERIALS.PRODUCTS ← 5 3 1 1 2 3

STORAGE_COST.MATERIALS ← 0.5 2.0

INITIAL_STOCK.MATERIALS ← 400 275

REMAINING_VALUE.MATERIALS ← 15 25
```

```
SUM [PRODUCTS: PRODUCTS.PERIODS?] ≤ MAXIMUM_PRODUCTION

MATERIALS.PERIOD_1? ≤ INITIAL_STOCK.MATERIALS

MATERIALS.PERIOD_2? = MATERIALS.PERIOD_1? -
   SUM[PRODUCTS: COMPOSITION.MATERIALS.PRODUCTS x
   PRODUCTS.PERIOD_1?]

MATERIALS.PERIOD_3? = MATERIALS.PERIOD_2? -
   SUM[PRODUCTS: COMPOSITION.MATERIALS.PRODUCTS x
   PRODUCTS.PERIOD_2?]

MATERIALS? = MATERIALS.PERIOD_3? -
   SUM[PRODUCTS: COMPOSITION.MATERIALS.PRODUCTS x
   PRODUCTS.PERIOD_3?]

MAXIMIZE [PRODUCTS: PERIODS: PROFIT.PRODUCTS.PERIODS x
   PRODUCTS.PERIODS?] - [MATERIALS: PERIODS:
   STORAGE_COST.MATERIALS x MATERIALS.PERIODS?]
   + SUM[MATERIALS: REMAINING_VALUE.MATERIALS x
   MATERIALS?]
```

This problem formulation is not fully independent of the data since a separate constraint instruction is written for each of the production periods, rather than using a single general constraint instruction to represent them all, as in the following ULP example. There the NEXT operator is used to permit a formulation in terms of index ranges rather than individual indexes, which are part of the data specifications. It was not indicated in the description [7] whether LPMODEL has a similar operator or the capability to relate indexes of different ranges (classes) to each other.

(4) EXAMPLE 2 - ULP

```
*RANGES

PRODUCTS: LOW,MEDIUM,HIGH;

MATEARIALS: SCRAP,NEW;

PERIODS: 'PERIOD '(1-3);

*TABLES

PROFIT(PRODUCTS,PERIODS):  25   20   10
                           50   50   50
                           75   80  100;

MAXIMUM PRODUCTION( ): 40;

COMPOSITION(MATERIALS,PRODUCTS):  5   3   1
                                  1   2   3;

STORAGE COST(MATERIALS):  0.5   2.0;

INITIAL STOCK(MATERIALS):  400   275;

REMAINING VALUE(MATERIALS):  15   25
```

```
UNKNOWN(X(PRODUCTS,PERIODS),S(MATERIALS,PERIODS),R(MATERIALS))

COMMENT(X(PRODUCTS,PERIODS)=AMOUNT OF PRODUCT TO MANUFACTURE IN EACH PERIOD)

COMMENT(S(MATERIALS,PERIODS)=STOCK OF RAW MATERIAL AT START OF EACH PERIOD)

COMMENT(R(MATERIALS)=STOCK OF EACH RAW MATERIAL REMAINING AFTER LAST PERIOD)

LPMAX(PROFIT(PRODUCTS,PERIODS)*X(PRODUCTS,PERIODS)
      -STORAGE COST(MATERIALS)*S(MATERIALS,PERIODS)
      +REMAINING VALUE(MATERIALS)*R(MATERIALS))

CONSTRAIN(PERIODS:X(PRODUCTS,PERIODS) ≤ MAXIMUM PRODUCTION)

CONSTRAIN(MATERIALS,NEXT(PERIODS):
          S(MATERIALS,NEXT(PERIODS))-S(MATERIALS,PERIODS)
          +COMPOSITION(MATERIALS,PRODUCTS)*X(PRODUCTS,PERIODS) = 0)

CONSTRAIN(MATERIALS:
          R(MATERIALS)-S(MATERIALS,'PERIOD 3')
          +COMPOSITION(MATERIALS,PRODUCTS)*X(PRODUCTS,'PERIOD 3') = 0)

BOUND(S(MATERIALS,'PERIOD 1') ≤ INITIAL STOCK(MATERIALS))
```

Appendix B

Backus-Naur Form (BNF) Representation of LET TABLE Instructions

1. <LET TABLE INSTRUCTION> ≡ LET TABLE(<HEADER><SEPARATOR><BODY>)
2. <HEADER> ≡ <TABLE NAME> | <TABLE NAME>(<RANGE TUPLE>)
3. <TABLE NAME> ≡ <ALPHANAME>
4. <ALPHANAME> ≡ <LETTER> | <LETTER><NAME>
5. <LETTER> ≡ A | B | C | D | etc.
6. <NAME> ≡ <ALPHANAME> | <DIGIT> | <DIGIT><NAME>
7. <DIGIT> ≡ 0 | 1 | 2 | 3 | etc.
8. <RANGE TUPLE> ≡ <RANGE NAME> | <RANGE NAME>,<RANGE TUPLE>
9. <RANGE NAME> ≡ <ALPHANAME>
10. <SEPARATOR> ≡ : | :=
11. <BODY> ≡ <TABLE DESIGNATION> | <TABLE DESIGNATION>:<MODIFICATION CLAUSE>
12. <TABLE DESIGNATION> ≡ <NUMBER LIST> | <DEFINITION LIST>
13. <NUMBER LIST> ≡ <NUMBER> | <NUMBER>,<NUMBER LIST>
14. <NUMBER> ≡ <DECIMAL> | +<DECIMAL> | -<DECIMAL>
15. <DECIMAL> ≡ <INTEGER> | .<INTEGER> | <INTEGER>. | <INTEGER>.<INTEGER>
16. <INTEGER> ≡ <DIGIT> | <DIGIT><INTEGER>
17. <DEFINITION LIST> ≡ <TABLE EXPRESSION> | <TABLE EQUAL LIST>
18. <TABLE EXPRESSION> ≡ <TABLE NAME> | <TABLE NAME>(<RANGE CLAUSE>)
19. <RANGE CLAUSE> ≡ <RANGE TERM> | <RANGE TERM>,<RANGE CLAUSE>
20. <RANGE TERM> ≡ <PLAIN RANGE TERM> | <SUBRANGE CONSTRUCT>
21. <PLAIN RANGE TERM> ≡ <RANGE EXPRESSION> | '<INDEX NAME>'
22. <RANGE EXPRESSION> ≡ <RANGE NAME> | <RANGE OPERATOR>(<RANGE TUPLE>)
23. <RANGE OPERATOR> ≡ NEXT | PREVIOUS
24. <INDEX NAME> ≡ <NAME>
25. <SUBRANGE CONSTRUCT> ≡ <PLAIN RANGE TERM><CONNECTOR><RANGE EXRESSION>

26. <CONNECTOR> ≡ () | (<MAP NAME>) | (<EQUIVALENCE LIST>)

27. <MAP NAME> ≡ <ALPHANAME>

28. <EQUIVALENCE LIST> ≡ <EQUIVALANCE> | <EQUIVALENCE>,<EQUIVALANCE LIST>

29. <EQUIVALANCE> ≡ '<INDEX NAME>'='<INDEX NAME>' | ''='<INDEX NAME>'

30. <TABLE EQUAL LIST> ≡ <TABLE EQUAL> | <TABLE EQUAL>,<TABLE EQUAL LIST>

31. <TABLE EQUAL> ≡ <DIRECT TABLE EQUAL> | <INDIRECT TABLE EQUAL>

32. <DIRECT TABLE EQUAL> ≡ <TABLE EXPRESSION>=<NUMBER LIST>

33. <INDIRECT TABLE EQUAL> ≡ <TABLE EXPRESSION>=<TABLE EXPRESSION>

34. <MODIFICATION CLAUSE> ≡ <TABLE EQUAL LIST>

References

[1] Bayer, R., and Witzgall, C., Some Complete Calculi for Matrices, Comm. of the ACM, 13, 223-237, 1970.

[2] Bayer, R., and Witzgall, C., Index Ranges for Matrix Calculi, Comm. of the ACM, 15, 1033-1039, 1972.

[3] Boudrye, C., and Greenberg, R., OMNI User Guide for the Energy Information Administration. Linear Programming, Inc., Silver Spring, MD, 1980.

[4] Fourer, R., Modeling languages versus matrix generators for linear programming. ACM Trans. Math. Softw. 9,2 (June 1983) 143-183.

[5] Greenberg, H.J., and Kalan, J.E., Enhancing Fortran to Aid Manipulation of Large Structured Matrices, Journal of Research of the NBS, 84, 21-25, 1977.

[6] Hoffman, K. and Witzgall, C., A Lexical Synthesis Approach to User-oriented Input Specification, in: Tools for Improved Computing in the 80's, Seventeenth Annual Technical Symposium of the ACM, National Bureau of Standards (June 1978).

[7] Katz, S., Risman, L.J., and Rodeh, M., A system for construction linear programming models. IBM Systems Journal 19,4 (1980) 505-520.

[8] Ketron, Inc., The DATAFORM Problem Description Language: A Tutorial. Arlington, VA, 1978.

[9] Knapp-Cordes, M., (oral communication), 1979.

[10] Murty, K.G., Linear and Combinatorial Programming, John Wiley and Sons, Inc., 1976.

[11] O'Neill, R.P., An Interactive Query System for MPS Solution Information.

[12] Palmer, K.H. et al., A Model-Management Framework for Mathematical Programming, An Exxon Monograph, John Wiley, New York, 1984.

[13] Phillips, J.R., and Adams, H.C., Comm of the ACM, 15, 1023-1031, 1972.

[14] Schrage, L., Linear Programming Models with LINDO, The Scientific Press, Palo Alto, CA, 1981.

[15] Schrage, L., User's Manual for LINDO, The Scientific Press, Palo Alto, CA, 1981.

[16] Shen, S.N.T., and Krulee, G.K., Solving Linear Programming Problems Stated in English by Computer, Proceedings ACM 73, 299-303, 1973.

[17] UMTA, Urban Transportation Planning System (UTPS), Introduction for Management, Department of Transportation, Washington, D.C., DOT-I-8049, June 1980.

[18] Ellison, E.F.D. and Mitra, G., UIMP: User Interface for Mathematical Programming. ACM Trans. Math. Softw. 8,3 (September 1982) 229-255.

[19] Lucas, C., Mitra, G., and Darby-Dowman, K., Modelling of Mathematical Programs: An Analysis of Strategy and an Outline Description of a Computer Assisted System. Technical Report TR/09/83, Department of Mathematics, Brunel University, Uxbridge, England, 1983.

[20] Darby-Dowman, K., Lucas, C., and Mitra, G., Computer Assisted Modelling of Linear, Integer and Separable Programming Problems. Technical Report TR/08/84, Department of Mathematics, Brunel University, Uxbridge, England, 1984.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBSIR 85-3125	2. Performing Organ. Report No.	3. Publication Date April 1985

4. TITLE AND SUBTITLE

Problem and Data Specification for Linear Programs

5. AUTHOR(S)

Christoph Witzgall and Marjorie McClain

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.
Interagency Agreement DOT -AT-20023

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

Methods Division, Office of Methods and Support
U.S. Department of Transportation
Urban Mass Transportation Administration
400 7th Street S.W. Washington, D.C. 20509

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

A language for specifying linear programs is proposed. The specification language is designed so as to enable the user to define the input for a particular linear program in terms of a given database of multi-dimensional tables. The specification language is formulated within the general framework of the UTPS system developed by the U.S. Department of Transportation. The structure of the underlying database system is described, and instructions for the writing of reports, again within the framework of the UTPS system, are discussed. Generation of the matrix for the specified linear program can be achieved during a single sequential pass through the database.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

database; input generation; lexical analysis; lexical synthesis; linear programming; modeling language; matrix generator; modeling language; multidimensional table; optimization; problem specification; report generator; software engineering

13. AVAILABILITY

☒ Unlimited
☐ For Official Distribution. Do Not Release to NTIS
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES
119

15. Price
$13.00