Crowder, Harlan Pinkney

CONTRIBUTIONS TO A THEORY FOR LINEAR PROGRAM PROBLEM MODELING

City University of New York PH.D. 1983

CONTRIBUTIONS TO A THEORY FOR LINEAR PROGRAM PROBLEM MODELING

by

HARLAN CROWDER

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1982

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

10/8/82
date

Michael Anshel
Professor Michael Anshel,
Chairman of Examining Committee

10/8/82
date

Paul R Karmel
Dr. Paul Karmel,
Executive Officer

Professor Donald Goldfarb
Professor Charles Haspel
Professor John M. Mulvey
Professor Jacob Rootenberg
Professor George Ross
Professor Patrick Sterbenz

Supervisory Committee

The City University of New York

Abstract

CONTRIBUTIONS TO A THEORY FOR
LINEAR PROGRAM PROBLEM MODELING

by

Harlan Crowder

Advisor: Professor Michael Anshel

This Thesis addresses a critical problem in the practical application of operations research: how to describe linear programming (LP) models to a computer. The data for such models occur naturally as sets of arrays, which are rectangularly arranged data aggregates. These input data quantify the operational and structural characteristics of processes modeled by LP. The representation of LP models in a computer is also an array, typically a sparse matrix representing the coefficient table for sets of linear equations and inequalities. The nonzero coefficients of this target data structure are comprised, either directly or by intermediate computation, of elements from input data arrays.

The main results of this Thesis are i) showing the relationship between the structure of input data arrays and the resulting target data arrays for LP models, and ii) presentation of a set of elementary array transformations, called distribution functions, for mapping input data arrays into a target array.

Acknowledgements

For encouragement, understanding, and, above all, patience, I am indebted to Michael Anshel and Philip Wolfe.

For helpful discussions and suggestions, thanks to Adin Falkoff, Alan Hoffman, Donald Orth, David Rabenhorst, and Arnold Wolf.

For never stopping believing, sincere appreciation to Ellis Johnson and Manfred Padberg.

And special thanks to Kacy Keene.

CONTENTS

INTRODUCTION

Linear programming (LP) is used extensively in operations research to analyze and plan real-world processes and systems. For example, it is used by farmers to plan crop rotation schedules, dietitians to ensure nutritional requirements in institutional meal planning, meat packers to prepare minimal-cost sausage recipes, economic analysts to explore tactical and strategic capital investment alternatives, utility companies for placement of facilities and analysis of peak-load requirements, integrated circuit designers to produce microcircuit specifications, and oil companies to plan exploration for new resources, schedule refinery operations, and distribute products.

The use of LP requires a mathematical description (a model) of the real-world process under study, a method for transforming the model into an LP problem, and a computational procedure for finding the solution. Solution methods have been studied extensively during the past 35 years, both in the area of algorithmic design and verification, and development of robust and efficient computer codes. Most state-of-the-art LP solution procedures today use some variant of Dantzig's simplex method [Dan63].

Investigation of methods for representing LP models and procedures for transforming models into LP problems has not kept pace with the investigation of solution procedures, primarily for two reasons: lack of definition and lack of technology.

Finding the optimal solution to a linear program is a well defined mathematical problem. The development and refinement of solution algorithms have traditionally drawn from established mathematical disciplines such as numerical analysis, linear algebra, functional analysis, and, recently, complexity theory. The development of LP solution computer software has benefitted from parallel developments in other areas of numerical and mathematical software.

LP model representation and transformation is not a well defined problem. While it is related to such established computer science disciplines as data base design and utilization, artificial intelligence (particularly symbolic algebra), and analysis of data structures, it has not had concepts in these areas brought to bear on the problem in a serious way. The concepts involved in building LP models are as much art as science. This is reflected in the scarcity of published material on building general LP models; see, for example, [Dan63, Wil78]. Commercially available software for model specification and problem building, so-called "matrix generators," have generally been problem-domain specific and have not used techniques from other computer science disciplines. (For more details about existing LP modeling systems and procedures, both commercial and experimental, see "Modeling Systems" on page 13.)

LP modeling systems have suffered from a lack of technology. The problem of solving an LP problem is computationally intensive; it generally requires no human interaction and is best performed in a "batch," job-oriented computational environment. The opposite is true for the problem of building an LP model; it requires extensive human-machine interaction and is best carried out in an interactive computing environment. Until only recently, however, sophisticated, versatile, widely available interactive computing systems did not exist.

Because of these gaps, definitional and technological, we know much more today about solving LP problems than we know about constructing them. This has led to an expensive discrepancy in the practical application of LP: If we consider the use of LP to analyze and solve a particular problem (discussed in "The LP Process" on page 10), it involves the collection and analysis of relevant and timely data, building and verification of the model, solving the problem, and analysis and verification of the solution. In practically all cases, only a small fraction of resources (measured in human and machine costs) are expended in the solution phase; the bulk of available resources for performing an LP application are expended in the data analysis and modeling phase. (Some estimates have put the figure at 90%; see [For79].)

A case in point, admittedly somewhat extreme, was the Project Independence Energy System (PIES), developed by the U.S. Department of Energy during 1974 in response to the OPEC oil embargo, to model U.S. energy production and consumption. The PIES project consumed the full-time effort of 60 people for six months to produce a model,

which then required five minutes of computer time to solve [Pie74]. Of course, the model was re-solved many times with various scenarios by varying the input parameters and data, but the total solution resources expended were negligible compared to the resources required for modeling the problem.

OVERVIEW

This Thesis is concerned with developing a systematic functional approach for describing LP models, and an automatic procedure for building LP problems. By its nature, the LP modeling process is not an exact science; it depends as much on common sense and intuition as it does mathematical knowledge. The concepts presented here do not alter this balance. The aim is to develop a useful tool that both extends the model builder's mathematical repertoire and opens new avenues for intuitive approaches to the problem.

The following gives a brief description of the major Sections of this Thesis.

Preliminaries

This Section introduces basic definitions, terminology and notation in two main areas: linear programming and APL. The mathematical LP problem is stated, and several previously ill-defined LP concepts are established for our purposes by definition (for example, the difference between an LP model and an LP problem). We outline the

process by which LP is applied to real applications, and survey the existing methods and procedures used to model LP applications. APL is used extensively in the development of concepts in this Thesis. Here we introduce APL arrays, functions, and operators, and define and demonstrate the APL primitive functions and operators used in this Thesis.

Generalized Reduction

We give a detailed description of the APL operators reduction and inner product, and show how these operations are related to linear programming. We then develop a set of criteria for extending the reduction operator to a larger domain of application, and formally define the generalized reduction function. The structural implications and effects on the array arguments of the generalized reduction function are explained and formally defined.

The Linear Program Modeling Problem

We present an efficient and versatile array-based data structure for storing LP problems in a computer. The kinds of data objects used to build an LP problem are discussed, and a method is introduced for partitioning LP problems into subproblems, based on the structure of its component data. The use of this partitioning scheme is demonstrated using the classical LP transportation problem. Finally, the important concept of distribution functions are introduced in terms of the transportation problem example.

Distribution Functions

This Section presents the formal definition of a set of distribution functions for specifying LP models and building LP problems. Each of the distribution functions is described and illustrated with examples. How these functions relate to the way people think about LP models is discussed.

Implementation: The Model Description Language (MDL)

This Section gives the specification of a high-level LP modeling language that uses the distribution function concept. The interactive MDL environment is described, MDL statements are defined and demonstrated by example, and the MDL interpreter is described from a user's viewpoint. A set of utility functions resident in the MDL environment for the analysis of LP problems is presented and demonstrated.

MDL Application Examples

This Section describes a set of real LP problems, and shows how these applications are modeled in MDL.

Appendix 1: BNF Description of the MDL Syntax

This short appendix augments "Implementation: The Model Description Language (MDL)" on page 128. It gives a formal description of the MDL syntax in Backus-Naur Form.

Appendix 2: Distribution Function Algorithms

This Appendix defines the actions of the functions described in "Distribution Functions" on page 88. In particular, the results produced by distribution functions are derived in terms of the function's arguments.

PRELIMINARIES

The purpose of this Section is to introduce basic definitions, terminology and notation used in this Thesis, in two main areas: linear programming (LP) and APL.

LINEAR PROGRAMMING

Here we define the classical mathematical LP problem, and give working definitions with respect to this Thesis for several previously ill-defined concepts related to LP. We present the process by which LP is applied to most real applications. And we briefly review existing methods and systems for specifying LP models and building LP problems.

DEFINITIONS

The Linear Programming (LP) Problem is:

> Given a real m-by-n matrix A, a real m-vector B, a real n-vector C, a real n-vector D, and a real scalar C0, determine the real n-vector X that will

$$\text{minimize } C0 + C \cdot X \tag{1.1}$$

subject to

$$A \cdot X = B \tag{1.2}$$

and

$$0 \leq X \leq D \tag{1.3}$$

A is the constraint matrix, B is the requirement vector or right-hand-side, C is the objective function vector, C0 is the objective function constant, and X is the solution or activity level vector. The symbol • denotes matrix-vector and vector-vector multiplication.

A Linear Programming Optimization (LPO) Problem is:

> The data items A, B, C, D and C0 for a specific instance of an LP problem; or an isomorphic representation of these data.

A Linear Programming Model is:

> A mathematical description of a process to be analyzed by linear programming.

An LP Model Language is:

> A notation for describing an LP model that, when combined with data from a specified domain, can be processed automatically by a computer to produce an LPO problem.

THE LP PROCESS

The use of linear programming to analyze and plan real-world systems and processes is a common operations research (OR) application. LP applications can be very straightforward and inexpensive, requiring one person's time and few resources for a one-shot analysis of a simple system, or such applications can be very complex and expensive, requiring the full-time support of legions of OR analysts and massive computer resources. Regardless of the complexity and size, the following steps are generally required to use LP in real OR applications.

1. Building a model

   Building a mathematical model of a real-world process requires two talents: an understanding of the physical system or process for which the model is being built, and an understanding of how to translate various physical relationships of the system into mathematical relationships. In most real-world applications, few people involved have both talents. One of the major operational problems in any OR application is how to get the system operators and system modelers communicating in a coherent manner.

   Using the terminology of [Dan63], Chapter 3, an LP model consists of items and activities. The items of a model represent resources or physical materials and are the inputs and outputs of the model. Activities are relations that describe an idealized interactivity of items in the system. In a model, relationships between items and activities are

expressed as sets of linear equations and inequalities. The coefficients of these linear relations are generally known only generically during the modeling phase; exact values are not known, but data structures and value ranges are assumed.

2. Data collection and analysis

This step involves the identification, collection, analysis, storage, and maintenance of data for an LP problem. The sources of data for small applications may be simple tables of numbers. For a complex application, the data may come from many sources and may include results provided by other computational systems. Some data values may be known exactly; for example, the cardinality of a set of existing facilities. Other data values may be only a guess; for example, the estimated future demand for a new consumer product. Still other data values may be timely, much like fresh vegetables: good today, acceptable tomorrow, rotten next week. Regardless of the source, accuracy, or value of data, its collection and analysis is time consuming and expensive except for the most trivial applications.

3. Building an LPO problem

This step involves using the mathematical model and the associated data to produce an LPO problem for presentation to a solver system. The actual tools used for this process are varied; see "Modeling Systems" below. If the model is expressed in a modeling language, then building the problem requires executing the model description. Otherwise, the model must be translated into an executable form.

4. Problem solution

This step involves finding a solution to the LPO problem using a solving procedure. As previously stated, state-of-the-art systems are capable of solving most real problems today. (There are exceptions, of course, especially in the area of integer linear programming; there are, however, few such applications relative to continuous LP.)

5. Solution analysis/model verification

This step involves analyzing the solution and ensuring that it makes sense in terms of the process or system being modeled. If it does not, then the model and data must be inspected to determine the source of error. In some cases, a correction involves a minor change to the model and/or data; in other cases, major modifications may be required. In any case, this step usually requires several re-applications of the problem building and solution steps.

This step can also involve sensitivity analysis, determining the effect on the solution of perturbations of the problem data.

6. Solution reporting

This step involves formatting the solution so that it can be used by interested people. The form of a solution report can be a written document or, with more frequency these days, an electronic report stored in a computer. This use of on-line reporting has blurred the distinction between the solution

analysis and solution report steps. Often, reports presented in this manner allow for the analysis of solution variability based on perturbations of model parameters.

This step can also involve formatting the solution for input to another computational procedure.

This Thesis is primarily concerned with describing LP models and building LPO problems (steps 1 and 3). Also, as appropriate, we will comment on step 2, data collection and analysis.

MODELING SYSTEMS

A modeling system, for our purposes, is an automatic procedure for building LPO problems. We can divide modeling systems, as they have traditionally been used, into three broad categories: problem-specific systems, application-specific systems, and general modeling systems.

Problem-specific modeling systems

A problem-specific system generates LPO problems for one model. Normally, input parameters and data can be varied, but not the general structural aspects of the model. Such systems are usually written in a general purpose high-level computer language such as Fortran, PL/I, or Cobol. The output produced by such a system is a representation of the LPO problem suitable for processing by an LP solver; for example, see [Ibm79], Chapter 4.

Problem-specific modeling systems are the most frequently used method for building LPO problems for real-world applications. While

efficient in terms of computer processing time, such systems are inefficient in terms of programmer and analyst time, especially when modifications to the structure of the model are required.

Application-specific modeling systems

Modeling systems of this type build LPO problems in a specific domain of application. Examples include NETGEN-II for generating network related problems [Ela81] and LPSYSTEM for building agricultural application models [Kat80]. Such systems generally have a specification language for describing structural characteristics and interrelationships of the process being modeled in the terminology of the application area. These systems are more flexible than problem-specific systems, especially when changes to model structure must be made.

General modeling systems

Many systems have been proposed, and a few implemented, for the general specification of LP models and the building of LPO problems. Commercially successful systems include OMNI [Hav76] and MGRW [Ibm74]. Examples of other systems and proposals can be found in [Bis77, Fou78, Hed75, Jar78, Mil77a, Mil77b, Woo76].

All general modeling systems have the following characteristics and features in some form:

- A model specification language for describing the components of an LPO problem (constraints, activities, requirements, bounds) and the mathematical relationship among components. Most specification languages try, in some form, to mimic traditional mathematical notation for expressing linear systems of equations and inequalities.

- A data base subsystem for collecting and organizing data required for the problem, and a data definition method for referencing data aggregates in the model specification.
- A model-data binding mechanism for actually assigning coefficient values when the LPO problem is built.
- A compiler program for processing the model specification, and an executor program for building the LPO problem.

The general modeling system that we propose in this Thesis, described in "Implementation: The Model Description Language (MDL)" on page 128, has the following characteristics and features:

- LP models are specified in a high-level, array-based modeling language.
- LP model specification programs and LP problem data, both in the form of arrays, share the same interactive computing environment. Information about the structure of problem data and how it relates to the structure of the model is used by the model specification program.
- LPO problems are represented as sparse arrays that can be manipulated in the interactive computing environment.
- LPO problems can be constructed row-wise (by constraint), column-wise (by activity), or a combination of the two.
- Model specification programs are interpreted by the MDL processor. There is no compilation phase; data binding and LPO problem production are accomplished in one step.

- A model specification program can be segmented into single model description statements that can be executed individually. The resulting LPO problem can thus be built as a series of subproblems, simplifying model validation.
- The Model Description Language is a superset of APL. Thus the data manipulation and processing power of APL is available in model specification programs.

APL

This Thesis uses the rectangular nested array as the model for representing data aggregates. Much of the development here has been strongly influenced by Iverson's A Programming Language (APL) [Fal73, Ibm78, Ive62, Ive76, Pol75], by proposed and existing extensions to APL [Bro71, Bro79, Gha73, Gul79, Ive78, Ibm82], and by the area of generalized arrays developed by More [Mor73, Mor75, Mor76a, Mor76b, Mor79a, Mor79b, Mor81]. We use these concepts as required to encompass our specific requirements. Because APL is a powerful tool for describing algorithms, much of the development in this Thesis uses the APL vocabulary.

This Section introduces and defines the concepts of arrays, which are aggregates of data organized in a systematic way, functions, which operate on arrays to produce new arrays, and

operators, which operate on functions to produce new functions. Much of the material in this Section was developed from early draft versions of [Ibm82], and from [Rab82].

We use the following notational conventions:

- The result of evaluating an APL expression is shown using the following convention: The expression to be evaluated will be indented six spaces from the left margin; the result will be flush left. For example,

```
      2+3
5
```

- The symbol ↔ between two expressions denotes equivalence:

```
      3 - 1  ↔  2
```

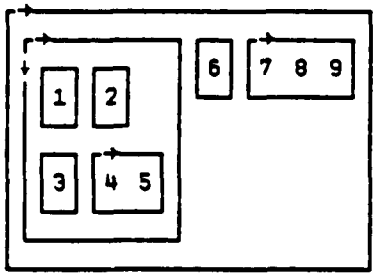
ARRAYS

An *array* is an ordered rectangular collection of *items*. Arrays are characterized by their values (numeric or character), there *valence* or *rank* (number of dimensions), and their *shape* (the extent of each dimension). Arrays can be *scalars*, which are nilvalent and dimensionless, *lists* or *vectors*, which are monovalent and have one dimension, *matrices* or *tables*, which are divalent and have two dimensions, or arrays of higher valence and number of dimensions. The items of an array can be single numbers or characters, or other arrays. An array whose items are all single numbers or characters is called a *simple array*.

If *A* is 5, then the array *A* is a scalar whose only item is the number 5. The list 2 4 6 is a simple array with three items. The array 2 4 (6 8) is a list with three items; the first two items are the numbers 2 and 3; the third item is the list 6 8.

Consider the nested array represented pictorially as follows:

A



Array *A* is a list of three items. The first item is a matrix of four items; the items of the matrix are the numbers 1, 2, 3, and the list 4 5. The second item of *A* is the scalar number 6. The third item of *A* is the list 7 8 9.

The shape of an array is a simple list giving the length of each axis of the array. The list 3 5 7 has one axis of length 3, as does the list 2 4 (6 8). If *M* has shape 3 4, then *M* is a matrix with three rows and four columns.

An empty array has at least one axis length of zero; an empty array has no items. If *E* has shape 0 5 then *E* is an empty matrix with no rows and five columns. Since scalars are dimensionless, their shape is an empty array (that is, a list of length zero).

The rank or valence is the number of dimensions of the array and is a measure of the length of the shape vector of the array. Thus,

the rank is the shape of the shape of an array and is, therefore, a list of length 1. Scalars have rank 0, lists rank 1, matrices rank 2, etc.

A singular array has exactly one item [Gha73]. The character scalar 'A' and the length of the list 2 4 6 are both singular arrays.

The main order of an array is a list of the items of the array in lexicographical order. For a matrix, this corresponds to row-major order; in general, main order is obtained by allowing the last axis to increment most rapidly. The main order for a scalar is a list containing one item.

The leaves of an array are simple scalars that comprise the elements of the array at all nesting levels. The array (2 3 (2 4)) has four leaves: 2, 3, 2, and 4.

FUNCTIONS

Functions operate on one or two arrays, called its argument(s), to produce an array as its result. The result can serve as an argument to another function. Thus, 3×4 is 12 and 2+3×4 is 14. A function that takes one argument is called monadic; its argument always appears on the right. A function that takes two arguments is called dyadic; its arguments appear one on each side of the function and are called the left and right argument.

The order of execution of a statement containing more than one function is right to left, except for parenthesized subexpressions, in which case the expression within matching parentheses pairs is evaluated before applying to the result any function outside the

matching pair. Thus, (15÷3)÷2 is 7, and 15÷3÷2 is 3, as is 15÷(3÷2). Stated another way, functions have long scope on the right and short scope on the left. The right argument of a function is the value of the complete expression to the right of the function; the only required use of parentheses is to form the left argument.

Functions are either primitive or defined. The primitive functions are denoted by special symbols. The set of primitive functions used in this Thesis is defined and demonstrated below. Defined functions are composed of primitive functions and other defined functions; a method for defining functions will be developed later in this Section.

Primitive Functions

Primitive scalar functions have the following characteristics:

- A primitive monadic scalar function applies to each item of its argument. For example,

```
      - 3 5
¯3 ¯5
```

- A scalar argument of a primitive scalar dyadic function will we extended to conform to the shape of the other argument. For example,

```
      2 + 3 4 5
5 6 7
```

- A primitive scalar dyadic function is applied to corresponding items in its arguments, as in

```
      1 2 3 + 4 5 6
5 7 9
```

Primitive pervasive functions are scalar functions that have these additional characteristics:

- A monadic pervasive function is applied to each leaf of its array argument; the structure of the argument and result are identical. For example,

  ```
  - (1 ¯2 3) 4 ¯5  ↔  (¯1 2 ¯3) ¯4 5
  ```

- A dyadic pervasive function is applied to corresponding leaves of its arguments (after scalar extensions) and produce results identical in structure to its arguments. If a simple scalar corresponds to a nonsimple scalar in its arguments, then the function is applied between the simple scalar and the items of the nonsimple scalar. For example,

  ```
  1 (2 3) 4 + 3 (4 5) (1 2)  ↔  4 (6 8) (5 6)
  ```

Pervasive Functions

The following APL primitive pervasive functions are used in this Thesis.

Negative: *Z* ← - *R*

R is a number. *Z* is the arithmetic negation of *R*.

Example:

```
      - 1 ¯2 0 3
¯1 2 0 ¯3
```

Not: *Z*← ~ *R*

R is logical (either 0 or 1). If *R* is 0, *Z* is 1. If *R* is 1, *Z* is 0.

Example:

```
      ~ 1 0 0 1
0 1 1 0
```

Reciprocal: *Z* ← ÷ *R*

R is a nonzero number. *Z* is defined in terms of the Divide function:

Z ↔ 1 ÷ *R*

Example:

```
      ÷ .25 .5 1 2 4
4 2 1 .5 .25
```

Add: *Z* ← *L* + *R*

L and *R* are numbers. *Z* is the arithmetic sum of *L* and *R*.

Examples:

```
      ¯1 + 1
0

      1 3 5 + 2
3 5 7
```

And: *Z* ← *L* ∧ *R*

L and *R* are logical (either 0 or 1). *Z* is the logical and of *L* and *R*.

Example:

```
      1 0 1 0 ∧ 1 1 0 0
1 0 0 0
```

Divide: *Z* ← *L* ÷ *R*

L and *R* are numbers. *Z* is the numeric quotient of *L* divided by *R*.

Examples:

```
      4 8 12 ÷ 4
1 2 3

      5 ÷ 2
2.5
```

Equal: *Z* ← *L* = *R*

L and *R* are both characters, or *L* and *R* are both numbers. *Z* is 1 if *L* and *R* are the same character (or number). Otherwise, *Z* is 0.

Examples:

```
      1 3 5 = 1 2 3
1 0 0
```

```
      'CAT' = 'FAT'
0 1 1
```

Greater: $Z \leftarrow L > R$

L and R are numbers. Z is 1 if L is greater than R. Otherwise, Z is 0.

Example:

```
      2 3 4 > 1 3 5
1 0 0
```

Less: $Z \leftarrow L < R$

L and R are numbers. Z is 1 if L is less than R. Otherwise, Z is 0.

Example:

```
      2 3 4 < 1 3 5
0 0 1
```

Maximum: $Z \leftarrow L \lceil R$

L and R are numbers. Z is the larger of L and R.

Example:

```
      1 1 ¯1 ¯1 ⌈ 2 ¯2 2 ¯2
2 1 2 ¯1
```

Minimum: $Z \leftarrow L \lfloor R$

L and R are numbers. Z is the smaller of L and R.

Example:

```
      1 1 ¯1 ¯1 ⌈ 2 ¯2 2 ¯2
1 ¯2 ¯1 ¯2
```

Multiply: $Z \leftarrow L \times R$

L and R are numbers. Z is the arithmetic product of L and R.

Example:

```
      0 2 ¯3 × 1 2 3
0 4 ¯9
```

Not Equal: $Z \leftarrow L \neq R$

L and R are both characters, or L and R are both numbers. Z is 0 if L and R are the same character (or number). Otherwise, Z is 1.

Examples:

```
      1 3 5 ≠ 1 2 3
0 1 1
```

```
      'CAT' ≠ 'FAT'
1 0 0
```

Not Greater: $Z \leftarrow L \leq R$

L and R are numbers. If L is less than R, or L is equal to R, then Z is 1. Otherwise, Z is 0.

Example:

```
      2 3 4 ≤ 1 3 5
0 1 1
```

Not Less: $Z \leftarrow L \geq R$

L and R are numbers. If L is greater than R, or L is equal to R, then Z is 1. Otherwise, Z is 0.

Example:

```
      2 3 4 ≥ 1 3 5
1 1 0
```

Or: $Z \leftarrow L \vee R$

L and R are logical (either 0 or 1). Z is the logical or of L and R.

Example:

```
      1 0 1 0 ∨ 1 1 0 0
1 1 1 0
```

Power: $Z \leftarrow L \star R$

L and R are numbers. If L is 0, R must be nonnegative. If R is 0, then Z is L. If L is a nonnegative integer, then Z is the product over L repetitions of R. For all other cases, the following relation is persevered:

$$R \star A+B \quad \leftrightarrow \quad (R \star A) \times R \star B$$

Thus, $X \star -N$ is the reciprocal of $X \star N$, and $X \star +N$ is the Nth root of X.

Examples:

```
      2 * ¯2 ¯1 0 1 2
 .25 .5 1 2 4

      64 * ÷1 2 3
64 8 4
```

Residue: $Z \leftarrow L \mid R$

L and R are numbers. If L is 0, then Z is R. If L is not 0, then Z is $R-L \times \lfloor R \div L$, the remainder of dividing R by L.

Examples:

```
     3 5 7 | 5
2 0 5

     5 | 3 5 7
3 0 2
```

Subtract: $Z \leftarrow L - R$

L and R are numbers. Z is the arithmetic difference L minus R.

Example:

```
     5 ¯5 5 ¯5 - 3 3 ¯3 ¯3
2 ¯8 8 ¯2
```

Pervasive Functions with Axis

Any primitive pervasive dyadic function can be applied with an axis specification to modify the function's behavior. The form of axis specification is

```
      Z ← L F[A] R
```

Examples:

```
      M
1 2 3
4 5 6

      10 20 +[1] M
11 12 13
24 25 26

      10 20 30 +[2] M
11 22 33
14 25 36
```

Nonpervasive Functions

The following APL primitive nonpervasive functions are used in this Thesis.

Enclose: `Z ← ⊂ R`

R is any array. *Z* is a scalar array whose only item is the array *R*. If *R* is a simple scalar, then *Z* is *R*.

Examples:

```
      ⊂ 2
2

      ⊂ 'CAT'
 CAT

      ⍴⍴ ⊂ 'CAT'
0

      ⍴⊃ ⊂ 'CAT'
3
```

Enclose with axis: `Z ← ⊂[A] R`

R is any array. *A* is a simple scalar or vector of integer indices of *R*. The set of axes specified in *A* are enclosed, forming an array Z of rank ((⍴⍴*R*)-⍴,*A*), with items of rank ⍴,*A*.

Examples:

```
      M
1 2 3
4 5 6

      ⊂[1] M
 1 4  2 5  3 6

      ⍴⊂[1] M
3

      ⊃⊂[1] M
1 4

      ⊂[2] M
 1 2 3  4 5 6

      ⍴⊂[2] M
2

      ⊃⊂[2] M
1 2 3
```

First: $Z \leftarrow \supset R$

R **is any array.** *Z* **is the first item of** *R*, **taken in major order.**

Examples:

```
      ⊃ ⍳5
1

      ⊃ (1 2) (3 4 5)
1 2

      ⊃ 2 2⍴(1 2) (3 4) (5 6) (7 8)
1 2
```

Grade Down: $Z \leftarrow ⍒ R$

R **is a numeric vector.** *Z* **is a permutation of** $⍳⍴R$ **that puts** *R* **in nonascending order.**

Examples:

```
      V←13 12 10 11 14

      ⍒ V
5 1 2 4 3
```

```
      V[⍒V]
14 13 12 11 10
```

Grade Up: $Z \leftarrow \grave{A}\ R$

R is a numeric vector. *Z* is a permutation of $\iota\rho R$ that puts *R* in nondescending order.

Examples:

```
      V←13 12 10 11 14

      ⍋ V
3 4 2 1 5

      V[⍋V]
10 11 12 13 14
```

Interval: $Z \leftarrow \iota\ R$

R is a simple nonnegative integer scalar or one element vector. *Z* is a simple vector of consecutive ascending integers starting with 1.

Examples:

```
      ⍳1
1

      ⍴⍳1
1

      ⍳5
1 2 3 4 5

      ⍴⍳5
5
```

Ravel: $Z \leftarrow ,\ R$

R is any array. *Z* is a list whose items are the items of *R*, taken in main order.

Examples:

```
      M
1 2 3
4 5 6

      , M
1 2 3 4 5 6
```

```
      , 5
5

      ⍴,5
1
```

Reverse: $Z \leftarrow \phi\ R$

R is any array. *Z* is an array with the same shape as *R*, and with the items of *R* reversed along the last axis.

Examples:

```
      ⌽ 2 4 6 8
8 6 4 2

      M
1  2  3  4
5  6  7  8
9 10 11 12

      ⌽ M
 4  3  2  1
 8  7  6  5
12 11 10  9
```

Reverse with axis: $Z \leftarrow \phi[A]\ R$

R is any array. *A* is a simple integer scalar or one element vector specifying an axis of *R*. *Z* is an array with the same shape as *R*, and with the items of *R* reversed along the axis specified in *A*.

Example:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      ⌽[1] M
9 10 11 12
5  6  7  8
1  2  3  4
```

Shape: $Z \leftarrow \rho R$

R is any array. Z is a nonnegative integer vector whose elements are the length of the coordinates of R. The length of Z (that is, ρZ) is the same as the rank or valence of R. In particular, if R is a scalar, then Z is the empty vector.

Examples:

```
      ⍴ 1 2 3
3
      ⍴ 5

      ⍴⍴ 5
0
      ⍴ ,5
1
```

Simple: $Z \leftarrow \equiv R$

R is any array. Z is 1 if every item of R is a scalar character or number. Otherwise, Z is 0.

Examples:

```
      ≡ 5
1
      ≡ 1 2 3
1
      ≡ 1 (2 3)
0
      ≡ ⊂5
1
      ≡ ⊂5 6
0
```

Transpose (monadic): $Z \leftarrow \lozenge R$

R is any array. Z is an array with the same rank as R, and with the order of the axes of R reversed (that is, $\rho Z \leftrightarrow \phi \rho R$).

Example:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      ⍉ M
1  5  9
2  6 10
3  7 11
4  8 12
```

Unite: $Z \leftarrow \cup R$

R is any array. Z is a simple vector whose elements are the leaves of R, taken in nested main order.

Example:

```
      ∪ (2 2⍴1 2 3 (4 5)) 6 (7 8 9)
1 2 3 4 5 6 7 8 9
```

Catenate: $Z \leftarrow L , R$

L and R are any arrays.

If L and R are scalars or vectors, Z is the vector formed by chaining L and R together.

Examples:

```
      1 , 2
1 2

      1 , 2 3
1 2 3

      1 2 , 3 4
1 2 3 4
```

If L and R are arrays of higher rank, then they are conformable for catenate if either i) they have the same shape, except possibly for the last coordinate, ii) their ranks differ by 1, and requirement i) is met after augmenting the array with the smaller rank with a unit-length last coordinate, or iii) L or R is a scalar.

Examples:

```
      M
1 2 3
4 5 6

      N
10 20
30 40

      M , N
1  2  3 10 20
4  5  6 30 40

      M , 10 20
1  2  3 10
4  5  6 20

      100 , M , 200
100  1  2  3 200
100  4  5  6 200
```

Catenate with axis: $Z \leftarrow L\ ,[A]\ R$

L and R are any arrays. A is a simple scalar or one element vector axis specification. Z is the array formed by applying the catenate function between L and R along the axis specified in A, rather than the last.

Examples:

```
      M
1 2
3 4
5 6

      N
10 20
30 40

      M ,[1] N
 1  2
 3  4
 5  6
10 20
30 40
```

```
      M ,[1] 10 20
 1  2
 3  4
 5  6
10 20

      100 ,[1] M ,[1] 200
100 100
  1   2
  3   4
  5   6
200 200

      '*',[1],('*',(2 3⍴'A'),'*'),[1]'*'
*****
*AAA*
*AAA*
*****
```

Compress: *Z ← L / R*

L is a simple logical scalar or vector. *R* is any array. *Z* is the subarray of *R* selected along the last coordinate of *R* by the 1's in *L*.

Example:

```
      1 0 1 1 0 / 1 2 3 4 5
1 3 4

      M
1  2  3  4
5  6  7  8
9 10 11 12

      1 0 1 0 / M
1  3
5  7
9 11
```

Compress with axis: *Z ← L /[A] R*

L is a simple logical scalar or vector. *A* is a simple scalar or one element vector axis specification. *R* is any array. *Z* is the subarray of *R* selected along the coordinate of *R* specified in *A* by the 1's in *L*.

Example:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      1 0 1 /[1] M
1  2  3  4
9 10 11 12
```

Drop: $Z \leftarrow L \downarrow R$

R is any array. *L* is a simple integer scalar or vector. If *L* is a scalar, then it will be treated as a one element vector. If *R* is a scalar, it will be treated as a singular array with rank ρ,*L*. *Z* is an array with the same rank as *R*, formed by removing planes from the coordinates of *R*. If *L*[*I*] is positive, then *L*[*I*] planes are removed from the beginning of the *I*th axis of *R*. If *L*[*I*] is negative, then -*L*[*I*] planes are removed from the end of the *I*th axis of R.

Examples:

```
      3 ↓ 1 2 3 4 5
4 5

      ¯3 ↓ 1 2 3 4 5
1 2

      M
1  2  3  4
5  6  7  8
9 10 11 12

      1 2 ↓ M
 7  8
11 12

      ¯1 ¯2 ↓ M
1 2
5 6
```

Drop with axis: $Z \leftarrow L \downarrow[A] R$

R is any array. L is a simple integer scalar or vector. A is a simple integer scalar or vector selecting axes in R. If L is a scalar, then it will be extended to a vector of length ρ,A. Z is an array with the same rank as R, formed by removing planes from the coordinates of R specified in A.

Examples:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      1 ↓[1] M
5  6  7  8
9 10 11 12

      2 ↓[2] M
 3  4
 7  8
11 12
```

Equivalent: $Z \leftarrow L \equiv R$

L and R are any arrays. Z is 1 if L and R are identical in value and structure. Otherwise, Z is 0.

Examples:

```
      2 ≡ 3
0

      3 ≡ 2 + 1
1

      1 ≡ ,1
0
```

Index: $Z \leftarrow L \; [] \; R$

R is any array. L is an array of valid integer indices for elements of R. Z is the subarray of R indexed by L.

If *R* is a vector, *L* may be a scalar or one element vector for selecting scalar elements of *R*.

Example:

```
      2 ⌷ 10 20 30 40 50
20
```

More than one element of a vector may be selected by indexing with an enclosed array.

Examples:

```
      (⊂2 4) ⌷ 10 20 30 40 50
 20 40

      (⊂2 2⍴⍳4) ⌷ 10 20 30 40 50
 10 20
 30 40
```

If *R* is an array of higher rank, *L* is a vector the same length as the rank of *R* indexing a single element.

Example:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      2 3 ⌷ M
7
```

More than one element may be selected by indexing with a vector of enclosed arrays.

Example:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      ((1 2) (1 2 3)) ⌷ M
 1 2 3
 5 6 7
```

Indexing can also be done using the bracket index form.

Examples:

```
      10 20 30 40 50 [2]
20

      10 20 30 40 50 [2 4]
20 40

      10 20 30 40 50 [2 2⍴⍳4]
10 20
30 40

      M
1  2  3  4
5  6  7  8
9 10 11 12

      M [2;3]
7

      M [1 2;1 2 3]
1 2 3
5 6 7
```

Index with axis: *Z* ← *L* []\[*A*] *R*

R is any array. *L* is an array of valid integer indices for elements of *R*. *A* is a simple integer scalar or vector specifying axes of *R*. *Z* is the subarray of *R* indexed by *L* along the axes specified in *A*.

Examples:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      (≤1 2) [] [1] M
 1 2 3 4
 5 6 7 8

      (≤2 3) [] [2] M
  2  3
  6  7
 10 11
```

Indexing can also be done using the bracket index form.

Examples:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      M [1 2;]
1 2 3 4
5 6 7 8

      M [;2 3]
 2  3
 6  7
10 11
```

Member: $Z \leftarrow L \in R$

L and *R* are any arrays. *Z* is a simple logical array with shape ρL. An element of *Z* is 1 if the corresponding item in *L* can be found anywhere in *R*. Otherwise, the element in *Z* is 0.

Examples:

```
      1 2 3 4 5 ∈ 2 4 6 8 10
0 1 0 1 0

      (⊂2 3) ∈ (1 2) (2 3) (3 4)
1

      ((2 3) (1 3)) ∈ (1 2) (2 3) (3 4)
1 0
```

Pick: $Z \leftarrow L \supset R$

R is any array. *L* is a scalar or vector of valid indices for *R*. *Z* is an item of *R* specified by the path *L*.

If *L* is a scalar or one element vector, then *Z* is $\supset L \lbrack\rbrack R$.

Examples:

```
      Z ← 1 ⊃ 1 (2 3) (4 5 6)

      Z
1

      ⍴⍴Z
0
```

```
      Z ← 2 ⊃ 1 (2 3) (4 5 6)
      Z
2 3
      ⍴Z
2
      M←2 2⍴1 (2 3) (4 5 6) (7 8 9 10)
      (⊂1 1) ⊃ M
1
      (⊂1 2) ⊃ M
2 3
```

If *L* has more than one item, then the function is applied recursively:

$$L \supset R \quad \leftrightarrow \quad (1\downarrow L) \supset (1\uparrow L) \supset R$$

If *L* is a two element vector, then *L*⊃*R* picks an item of an item of *R*.

Examples:

```
      2 1 ⊃ 1 (2 3) (4 5 6)
2
      2 2 ⊃ 1 (2 3) (4 5 6)
3
      M←2 2⍴1 (2 3) (4 5 6) (7 8 9 10)
      (2 2) 1 ⊃ M
7
```

Reshape: $Z \leftarrow L \rho R$

R is any array. *L* is a simple nonnegative integer scalar or vector. *Z* is an array with shape *L* whose items are the items of *R* taken in major order, and repeated cyclically if required.

Examples:

```
     5 ⍴ 1
1 1 1 1 1
```

```
      3 4 ⍴ ⍳12
1  2  3  4
5  6  7  8
9 10 11 12
```

Rotate: $Z \leftarrow L \phi R$

R is any array. L is a simple integer array. Z is an array with the same shape as R.

If L is a positive scalar, then L items are removed from the beginning of each rank-1 subarray along the last axis of R, and appended to the end of the same subarray.

If L is a negative scalar, then $-L$ items are removed from the end of each rank-1 subarray along the last axis of R, and prefixed to the beginning of the same subarray.

If L is scalar 0, then Z is R.

If L is nonscalar, then ρL must be $^{-}1 \downarrow \rho R$, and the subarrays of R along the last axis are treated independently according to the corresponding elements of L.

Examples:

```
      2 ⌽ 1 2 3 4 5 6 7
3 4 5 6 7 1 2

      ¯2 ⌽ 1 2 3 4 5 6 7
6 7 1 2 3 4 5

      M
1  2  3  4
5  6  7  8
9 10 11 12

      1 ⌽ M
 2  3  4  1
 6  7  8  5
10 11 12  9

      ¯1 ⌽ M
 4 1  2  3
 8 5  6  7
12 9 10 11
```

```
      0 1 ¯2 ⌽ M
 1  2  3  4
 6  7  8  5
11 12  9 10
```

Take: $Z \leftarrow L \uparrow R$

R is any array. *L* is a simple integer scalar or vector. If *L* is a scalar, then it will be treated as a one element vector. If *R* is a scalar, it will be treated as a singular array with rank ρ,*L*. *Z* is an array with the same rank as *R*, formed by taking planes from the coordinates of *R*. If *L*[*I*] is positive, then *L*[*I*] planes are taken from the beginning of the *I*th axis of *R*. If *L*[*I*] is negative, then -*L*[*I*] planes are taken from the end of the *I*th axis of R.

If the magnitude of *L*[*I*] exceeds the length of the *I*th axis of *R*, then the extra positions in *Z* are filled with ' ' (blank) if $\supset R$ is character, or with 0 if $\supset R$ is numeric.

Examples:

```
      3 ↑ 1 2 3 4 5
1 2 3

      ¯3 ↑ 1 2 3 4 5
3 4 5

      M
 1  2  3  4
 5  6  7  8
 9 10 11 12

      1 2 ↑ M
1 2

      ¯1 ¯2 ↑ M
11 12
```

Take with axis: $Z \leftarrow L \uparrow[A] R$

R is any array. L is a simple integer scalar or vector. A is a simple integer scalar or vector selecting axes in R. If L is a scalar, then it will be extended to a vector of length ρ,A. Z is an array with the same rank as R, formed by taking planes from the coordinates of R specified in A.

Examples:

```
      N
1  2  3  4
5  6  7  8
9 10 11 12

      2 ↑[1] N
1  2  3  4
5  6  7  8

      3 ↑[2] N
1  2  3
5  6  7
9 10 11
```

Transpose (dyadic): $Z \leftarrow L \lozenge R$

R is any array. L is a simple integer scalar or vector of the axes of R. The number of elements in L must be the rank of R, and L must be a permutation of $\iota\rho\rho R$. Z is an array similar to R, with the order of the axes of R permuted such that

$$\rho Z \leftrightarrow (\rho R)[L]$$

Examples:

```
      A
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      ρA
2 3 4
```

```
      3 2 1 ⍉ A
 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23

 4 16
 8 20
12 24

      ⍴3 2 1 ⍉ A
4 3 2

      2 1 3 ⍉ A
 1  2  3  4
13 14 15 16

 5  6  7  8
17 18 19 20

 9 10 11 12
21 22 23 24

      ⍴2 1 3 ⍉ A
3 2 4

      1 3 2 ⍉ A
 1  5  9
 2  6 10
 3  7 11
 4  8 12

13 17 21
14 18 22
15 19 23
16 20 24

      ⍴1 3 2 ⍉ A
2 4 3
```

Defined Functions

Defined functions are expressed in Iverson's *direct function definition* notation [Ive76]. A function definition will be represented in general by

FNAME : *EXP*

where *FNAME* is the function name and *EXP* is an expression composed of combinations of arrays, primitive functions and other defined functions. The value of the result produced by *FNAME* is obtained by evaluating *EXP*.

The special symbols ⍺ and ⍵ will denote the left and right arguments of the function, respectively. Thus, if *EXP* contains ⍵ but not ⍺, *FNAME* is a monadic function. If *EXP* contains both ⍺ and ⍵, *FNAME* is a dyadic function.

For example, an expression for computing the square root of the elements of a numeric array *A* is *A*⋆0.5. Thus, 4⋆0.5 is 2 and 4 9 16⋆0.5 is 2 3 4. This expression can be used to define the monadic function *SQRT* as

SQRT:⍵⋆0.5

Thus, *SQRT* 4 is 2 and *SQRT* 4 9 16 is 2 3 4.

The dyadic function *F* can be defined informally as follows: the result of applying *F* is the sum of the left argument and the square of the right argument. For example, 3 *F* 2 is 7 and 3 5 4 *F* 2 2 3 is 7 9 13. The definition of *F* is

F:⍺+⍵⋆2

A more general form of direct function definition is represented by

FNAME : 0-*EXP* : *B-EXP* : 1-*EXP*

where *FNAME* is the function name, 0-*EXP* and 1-*EXP* are expressions as before, and *B-EXP* is an expression that evaluates to the value 0 or 1. If *B-EXP* evaluates to the value 0, the value of the result

produced by *FNAME* will be the value obtained by evaluating 0-*EXP*. If *B-EXP* evaluates to 1, the result of *FNAME* is obtained from the evaluation of 1-*EXP*.

This form of function representation allows functions to be defined recursively, since 0-*EXP* and 1-*EXP* can invoke defined functions, including *FNAME*. For example, a monadic function *FACT* to compute the factorial of a scalar integer can be defined by

```
      FACT: ω×FACT ω-1 : ω=0 : 1
```

Thus, *FACT* 5 is 120, and *FACT* 1 is 1, as is *FACT* 0.

As another example of a recursively defined function, a simple and elegant statement of Euclid's algorithm for the greatest common divisor of two numbers, assuming that α≤ω, is given by the function *GCD*:

```
      GCD:Z GCD α:0=Z←α|ω:α
```

For example,

```
      18 GCD 24
6
      18 GCD 36
18
      15 GCD 18
3
```

OPERATORS

Operators operate on one or two functions to produce a function as a result. The result of an operator is a derived function, which can serve as an argument to another operator. A monadic operator always takes its function argument on the left. The arguments of a dyadic operator appear one on each side of the operator.

Operators have higher precedence than functions, and they have long scope on the left. Dyadic operators have short scope on the right. (The scope of operators is exactly opposite that of functions.)

The following APL operators are used in this Thesis.

Each: $Z \leftarrow F\ddot{}\ R$

F is a monadic function. R is an array whose items are appropriate for the function F. Z is an array with shape ρR formed by applying F to each item of R.

Examples:

```
      ⍴¨ (1 2) (1 2 3) (1 2 3 4)
2 3 4

      ⍳¨ 2 3 4 5
 1 2  1 2 3  1 2 3 4  1 2 3 4 5

      ⍴ ⍳¨ 2 3 4 5
4

      ⍴¨ ⍳¨ 2 3 4 5
2 3 4 5
```

Reduce: $Z \leftarrow F/\ R$

F is a dyadic function. R is an array whose subarrays along the last coordinate are appropriate for the function F. Z is the the array formed by applying F between subarrays along the last coordinate of R. For more details, see "Reduction" on page 49.

Examples:

```
      +/ 5 2 3 1 4
15

      ×/ 5 2 3 1 4
120

      ⌈/ 5 2 3 1 4
5
```

```
      M
6 2 3
5 4 1

      +/M
11 10

      ×/ M
36 20

      ⌈/ M
6 5
```

Inner Product: *Z ← L F.G R*

F and *G* are dyadic functions. *L* and *R* are any arrays whose elements are appropriate for the function *G*. The array *Z* is obtained by applying the function *G* between the subarrays taken along the last coordinate of *L* and the subarrays taken along the first coordinate of *R* in all combinations; the derived function *F/* is applied to each of these results. For more details, see "Inner Product" on page 51.

Examples:

```
      10 20 30 +.× 3 2 1
100

      M
10 20 30
40 50 60

      1 2 +.× M
90 120 150

      M +.× 1 2 3
140 320
```

Outer Product: *Z ← L ∘.G R*

G is a dyadic function. *L* and *R* are any arrays whose elements are appropriate for the function *G*. The array *Z* is obtained by applying the function *G* between each item of *L* and each item of *R*, in all combinations.

Examples:

```
      0 1 2 3 4 5 ∘.× 1 2 3 4 5
0   0   0   0   0
1   2   3   4   5
2   4   6   8  10
3   6   9  12  15
4   8  12  16  20
5  10  15  20  25

      1 2 3 4 ∘.≤ 1 2 3 4
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
```

GENERALIZED REDUCTION

Vector-vector, matrix-vector, and matrix-matrix multiplications are operations that arise in many contexts in linear programming. For example, the formulation of the linear programming problem in "Linear Programming" on page 8 was cast in terms of a vector-vector multiplication (expression (1.1), the objective function) and a matrix-vector multiplication (expression (1.2), the constraint set). Here we want to review the APL facilities available for these kinds of array products, and introduce the new concept of generalized reduction that is crucial to subsequent discussion of linear program problem modeling.

REDUCTION

The monadic operator reduction is denoted by F/R for a dyadic function F and an array R. For a numeric list V, the expression $+/V$ is the arithmetic sum of the items of V and $\times/V$ is the product of the items of V. In general, F/V is

$$V[1]\ F\ V[2]\ F\ \ldots\ F\ V[\rho V] \qquad (2.1)$$

For example, if V is 4 2 3 5 1, then $+/V$ is 15, $\times/V$ is 120, and $\lceil/V$ is 5.

If *G* is the function

```
      G:⍺+⍵*0.5
```

then *G*/2 3 1 is 4.

If *S* is any scalar, then by definition, *F*/*S* is *S* for all functions *F*. If *E* is an empty list, then *F*/*E* is the identity element for the function *F*. For example, +/*E* is 0 and ×/*E* is 1.

For arrays *R* with valence greater than one, the result *F*/*R* is obtained by applying *F* over the last coordinate of *R*. For example,

```
      R
 1  2  3
 4  5  6

      +/R
6 15
```

In general, the valence of the array result of reduction is one less than the valence of the array argument. Thus lists are reduced to scalars, matrices are reduced to lists, etc.

The axis operator, denoted by *F*[*A*], can be used to modify the action of several functions including functions derived by reduction. In particular, *F*/[*A*]*R* can be used to specify which coordinate of the array *R* that the function *F* is to be applied along. For example,

```
      A
 1   2   3   4
 5   6   7   8
 9  10  11  12

      +/[1]A
15  18  21  24

      +/[2]A
10  26  42

      +/A
10  26  42
```

INNER PRODUCT

The dyadic operator inner product is denoted by *L F.G R* for dyadic functions *F* and *G* and arrays *L* and *R*. For *L* and *R* to be conformable for inner product, the last coordinate of *L* must be the same length as the first coordinate of *R*. That is,

$$-1\uparrow\rho L \leftrightarrow 1\uparrow\rho R \qquad (2.2)$$

The result of inner product is obtained by applying the function *G* between the subarrays taken along the last dimension of *L* and the subarrays taken along the first dimension of *R* in all combinations; the derived function *F/* is applied to each of these results.

Note that the derived function +.× is the classical multiplication function of linear algebra.

Examples:

```
      1 2 3 +.× 6 4 2
20

      A
10 20
30 40

      B
1 2
3 4

      A+.×B
 70 100
150 220

      A+.×1 2
50 110
```

INNER PRODUCT AND LINEAR PROGRAMMING

We can now restate the definition of the linear programming problem presented in "Linear Programming" on page 8 (expressions (1.1)-(1.3)) in terms of arrays and inner product:

Given arrays A, B, C, D and $C0$ with the following structure --

```
2 ↔ ⍴⍴A
1 ↔ ⍴⍴B ↔ ⍴⍴C ↔ ⍴⍴D
1↑⍴A ↔ ⍴B
¯1↑⍴A ↔ ⍴C ↔ ⍴D
0 ↔ ⍴⍴C0
```

-- find a vector X of length $\bar{}1\uparrow\rho A$ that will minimize

$$C0 + C+.\times X \tag{2.3}$$

subject to

$$B\equiv A+.\times X \tag{2.4}$$

$$(\wedge/X\geq 0)\wedge\wedge/X\leq D \tag{2.5}$$

Despite its notational simplicity and computational power, inner product is limited in two respects. First, it is limited in its structural flexibility; the function derived by reduction only applies over a predetermined set of coordinates (the last of the left argument and the first of the right argument). And second, inner product only allows a single reduction operation; you can not, for example, sum-reduce all coordinates to get the scalar grand total after multiplying two matrices of equal shape together. To overcome these restrictions, we want to generalize reduction in order to solve the following type of problem:

Suppose we have two numeric arrays *X* and *Y*, and assume for now that *X* and *Y* have the same rank and shape. For example,

```
      X
1 2 3
4 5 6

      Y
10 20 30
40 50 60
```

We could first multiply *X* and *Y* together item-wise, and then sum-reduce the resulting array along the first coordinate to obtain 170 290 450. In traditional mathematical notation, this operation is denoted by the expression

$$\sum_{i} x_{ij}y_{ij}, \quad \text{all } j$$

Or we could sum-reduce the result along the second coordinate to obtain 140 770. This is normally expressed as

$$\sum_{j} x_{ij}y_{ij}, \quad \text{all } i$$

Or we could sum-reduce the result along the first and second coordinate to get 910, normally written as

$$\sum_{ij} x_{ij}y_{ij}$$

The generalized reduction function *GRF* is an APL defined function for performing these calculations:

```
      1 GRF X Y
170 290 450

      2 GRF X Y
140 770

      1 2 GRF X Y
910
```

The function *GRF* has the following syntax:

Z ← *A GRF* (*L R*)

where *L* and *R* are numeric arrays (assume for now they have the same rank and shape), and *A* is a list of axes for *L* and *R*. *GRF* has the following definition:

```
      GRF: ((,⍺)[⍋,⍺]) SPR (⊃⍵)×2⊃⍵
```

The right argument to the successive plus-reduction function *SPR* is the scalar product of *L* and *R*. The left argument to *SPR* is the list of axis indices, sorted into ascending order. *SPR* plus-reduces its right argument over all coordinates mentioned in its left argument:

```
      SPR: (¯1↓⍺) SPR +/[¯1↓⍺]⍵ : 0=⍴⍴,⍺ : ⍵
```

We now want to consider extending the domain of generalized reduction to arrays of unequal rank. For example, suppose we have two arrays *X* and *Y* --

```
      X
2 0 1
0 3 1

      ⍴X
2 3

      Y
 1  3  5  7
 9 11 13 15
17 19 21 23

 2  4  6  8
10 12 14 16
18 20 22 24

      ⍴Y
2 3 4
```

We want the result *Z* of

```
      Z←1 2 GRF X Y                                    (2.6)
```

to be the list 67 81 95 109 where

```
      Z[I] ↔ +/,X×Y[;;I]
```

In traditional mathematical notation, this calculation is represented by the expression

$$\sum_{ij} x_{ij} y_{ijk}, \quad \text{all } k$$

Similarly, given the arrays *X* and *Y* --

```
      X
1 0 0 2
0 3 0 1

      ⍴X
2 4

      Y
10 20 30

40 50 60

      ⍴Y
2 1 3
```

we want the result *Z* of

```
      Z←1 2 GRF X Y                                    (2.7)
```

to be the list 190 260 330 where

```
      Z[I] ←→ +/,X×⍉4 2⍴Y[ ;I]
```

As we will see in "The Linear Program Modeling Problem" on page 67, generalized reduction calculations of the form (2.6) and (2.7) play an important role in the mathematical formulation of LP problems. Therefore, we want to establish a rigorous definition for an extension of the generalized reduction function *GRF* to these and similar calculations.

COVER AND MATCH

The simple array *A* covers the simple array *B* if the corresponding elements in the initial overlap of their shape vectors ⍴*A* and ⍴*B* are identical or, for the elements that differ, the elements of *B* are unity. Empty arrays are covered by any nonempty array and scalars cover any array. The function *CV* checks the cover relationship for two nonempty arrays, producing 1 if ⍺ covers ⍵, else 0.

```
      CV:∧/(1=T↑⍴⍵)∨(T↑⍴⍵)=(T←(⍴⍴⍺)⌊⍴⍴⍵)↑⍴⍺:0=⍴⍴⍵:1
```

For example, if

```
      ⍴A ↔ 1 4
      ⍴B ↔ 3 4
      ⍴C ↔ 3
      ⍴D ↔ 1 5
```

then

```
      (A B C D) ∘.CV A B C D
1 0 0 0
1 1 1 0
1 1 1 1
0 0 0 1
```

Two arrays match if the initial overlap of their shape vectors is identical. That is, *A* and *B* match if and only if *A CV B* and *B CV A*. For example, if ⍴*A* is 2 3 and ⍴*B* is 2 3 4, then *A* and *B* match. Scalars match any array.

If *A* covers *B*, but not conversely, *A* and *B* can be made to match by extending unit-length axes in the initial overlap of the shape of *B* to the corresponding axis length in the shape of *A*. This process, called axis extension, fills the new subarrays along the extended

axes with replications of the original array *B*. (A rigorous explanation of extending unit-length axes is given later in "Axis Extension and Replication" on page 61.)

For example, if ⍴*A* is 2 3 and *B* is 1 3⍴⍳3, *A* covers *B* but *B* does not cover *A*. Carrying out axis extension on the first axis of *B* to match *A* yields the array with shape 2 3:

```
1 2 3
1 2 3
```

The shape excess of two arrays *A* and *B* is defined informally as follows: suppose (⍴⍴*A*)<⍴⍴*B*. Then the shape excess of *A* and *B* is (⍴⍴*A*)↓⍴*B*. The shape excess of two arrays is formed by deleting from the shape vector of the array with the higher valence the number of elements given by the valence of the other array. The shape excess of a scalar and any array *A* is ⍴*A*.

The function *SX* gives a formal definition of shape excess:

```
SX:((⍴⍴⍺)↓⍴⍵),(⍴⍴⍵)↓⍴⍺
```

One (or both) arguments to catenate will be the empty list.

THE GENERALIZED REDUCTION FUNCTION

The generalized reduction function extends the derived function +.× to operate over multiple axes.

Syntax: *Z ← A GRF (L R)*

The right argument is a list containing two items, *L* and *R*, which are both simple numeric arrays. The left argument *A* is a simple numeric scalar or list of indices that index the coordinates of *L*. The result *Z* is a simple numeric array.

Conformability requirement:

L must cover *R*. If *N* is the list of coordinate indices of *L* not in *A* -- that is, *N* ← (~(⍳⍴⍴*L*)∈*A*)/⍳⍴⍴*L* -- then

$$(\rho Z) \equiv (\rho L)[N],L \; \underline{SX} \; R \qquad (2.8)$$

where *SX* is the shape excess function.

Definition:

From the conformability requirement, *L* covers *R*. If *L* and *R* do not match, axis extension is performed on *R*, extending its unit-length axes to the corresponding axes in *L*. (A rigorous treatment of axis extension is given in "Axis Extension and Replication" on page 61.) After axis extension, (*K*↑⍴*L*)≡*K*↑⍴*R*, where *K* is (⍴⍴*L*)⌊⍴⍴*R*.

After axis extension on *R*, if *L* and *R* differ in rank, the array with the smaller rank is replicated to the same shape as the array with the larger rank. For example, suppose (⍴⍴*L*)>⍴⍴*R*. Then, if *T* ↔ (⍴⍴*R*)-⍴⍴*L*, the replication of *R* to the shape of *L* is

$$R\leftarrow(T\phi\iota\rho\rho R)\backslash R\leftarrow(((\rho\rho R)\uparrow\rho L),\rho R)\rho R \qquad (2.9)$$

After axis extension on *R* (if required) and replication (if required), (⍴*L*)≡⍴*R*. Then

$$Z\leftarrow(,A)[\lfloor\downarrow,A] \; SPR \; L\times R \qquad (2.10)$$

where, as before, *SPR* is the successive plus-reduction function.

Examples:

```
      U
1 0 2
0 1 0
```

```
      ρU
2 3

      V
10 20 30
40 50 60

      ρV
2 3

      1 GRF U V
10 50 60

      2 GRF U V
70 50

      1 2 GRF U V
120

      U
1 0 2
0 1 0

      ρU
2 3

      W
 10  20  30  40
 50  60  70  80
 90 100 110 120

130 140 150 160
170 180 190 200
210 220 230 240

      ρW
2 3 4

      1 GRF U W
 10  20  30  40
170 180 190 200
180 200 220 240

      2 GRF U W
190 220 250 280
170 180 190 200

      1 2 GRF U W
360 400 440 480

      U
1 0 2
0 1 0
```

```
      ρU
2 3

      X
10 20 30 40

50 60 70 80

      ρX
2 1 4

      1 GRF U X
10 20 30 40
50 60 70 80
20 40 60 80

      2 GRF U X
30 60 90 120
50 60 70  80

      1 2 GRF U X
80 120 160 200

      Y
1 0 2
0 1 0

2 0 1
0 1 0

      ρY
2 2 3

      Z
10 20 30
40 50 60

      ρZ
1 2 3

      1 GRF Y Z
30   0 90
 0 100  0

      2 GRF Y Z
10 50 60
20 50 30

      3 GRF Y Z
70 50
50 50
```

```
      1 2 GRF Y Z
30 100 90

      1 3 GRF Y Z
120 100

      2 3 GRF Y Z
120 100

      1 2 3 GRF Y Z
220
```

AXIS EXTENSION AND REPLICATION

Axis extension and axis replication are key concepts in generalized reduction. This Section gives intuitive and rigorous definitions of these operations.

Axis Extension

In a generalized reduction expression of the form

Z ← A GRF (L R)

a unit-length axis in the array *R* acts as a place holder. It means that a unit-length coordinate in *R* is to be treated as having the same length as the corresponding coordinate in *L*. The definition and conformability requirements for generalized reduction implicitly assume that unit-length axes in *R* will extend to the corresponding axes lengths in *L*. Two questions arise in this context; first, what are the structural implications of introducing unit-length axes into an array? And what is the formal definition of axis extension of unit-length axes?

To answer the first question, consider the unit axis function $\underline{UA}$ defined as

```
      ⍺UA⍵:(⍺⍋⍺,⍳⍴⍴⍵)⍉(((⍴,⍺)⍴1),⍴⍵)⍴⍵
```

The right argument ω is a nonscalar array. The left argument α is a scalar or list of fractional indices indicating were, in relation to existing coordinates in ω, unit-length axes are to be introduced. The right argument of transpose Φ is an array of the desired valency with all the new unit-length axes occurring in the initial segment of its shape vector. The left argument of transpose places the new axes in their proper location based on α.

For example, if

```
      3 4≡⍴A
```

then

```
      3 1 4≡⍴1.5 UA A
      1 3 1 4≡⍴0.5 1.3 2.2 UA A
```

Let A be a nonempty, nonscalar array, $A1$ be A augmented with unit-length axes, and U a list containing the locations in $\rho A1$ of the value 1. That is,

```
      U≡(1=⍴A1)/⍳⍴⍴A1
```

For example, if ρA is 2 3 and $\rho A1$ is 2 1 3 1 1, then U is 2 4 5. We can explore the relationship between A and $A1$ using the following identities:

```
      ⍴,A ↔ ⍴,A1                                    (2.11)
```

That is, A and $A1$ contain the same number of items.

```
      A ↔ ((⍴,U)⍴1)⌷[U]A1                            (2.12)
```

Indexing a subarray of $A1$ using all unit-length axes is A.

$$,A \leftrightarrow ,A1 \qquad (2.13)$$

The main order of A and $A1$ is the same.

$$A \leftrightarrow (\rho A)\rho A1 \leftrightarrow (\rho A)\rho,A1 \leftrightarrow (\rho A)\rho,A \qquad (2.14)$$

Identity (2.13) and the definition of reshape.

$$A \leftrightarrow U\ SPR\ A1 \qquad (2.15)$$

SPR is the successive plus-reduction function defined in "Inner Product and Linear Programming" on page 52. The reduction of an array along any unit-length coordinate serves only to lower the valency by one, and in particular, the reduction of $A1$ along <u>all</u> unit-length axes is A.

<u>Definition:</u>

Axis extension involves extending unit-length axes of an array, replicating copies of the array along the extended axes. The function <u>⍙⍗</u> offers a formal definition of axis extension. The right argument ⍵ is an array containing unit-length axes; the left argument ⍺ is a simple scalar or list of positive integers giving the length to which each unit-length axis of ⍵ is to be extended. For conformability, ⍴,⍺ ↔ +/1=⍴⍵.

```
⍙⍗:T⍉(⍺,(⍴,⍺)+⍴Z)⍴Z←(⍋T←⍳⍴⍵)⍉⍵
```

The array Z is ⍵ with all unit-length axes moved to the initial segment of ⍴Z by the appropriate transpose. The left argument of reshape discards the unit-length axes from ⍴Z, replacing them with ⍺; the result of reshape is then transposed back to the original order.

Examples:

```
A←1 3 4⍴⍳12
```

```
      A
 1  2  3  4
 5  6  7  8
 9 10 11 12

      2 ⍪X A
 1  2  3  4
 5  6  7  8
 9 10 11 12

 1  2  3  4
 5  6  7  8
 9 10 11 12

      ⍴2 ⍪X A
2 3 4

      B←3 1 4⍴⍳12

      B
 1  2  3  4

 5  6  7  8

 9 10 11 12

      2 ⍪X B
 1  2  3  4
 1  2  3  4

 5  6  7  8
 5  6  7  8

 9 10 11 12
 9 10 11 12

      ⍴2 ⍪X B
3 2 4
```

Let *A* be a nonscalar array with no unit-length axes, *A*1 be *A* augmented with unit-length axes in locations *U* of ⍴*A*1, and *A*2 be *L* ⍪X *A*1. Then

```
      A≡I⌷[U]A2
```

for all *I* for which *I*⌷[*U*]*A*2 is defined (that is, for ∧/*I*>0 and ∧/*I*≤*L*). The number of "copies" of *A* in *A*2 is ×/*L*.

Axis Replication

Given two simple arrays α and ω that match (that is, α ⍷ ω and ω ⍷ α), but that differ in rank (for example, (⍴⍴α)>⍴⍴ω), the array ω can be replicated to the same shape as α by the axis replication function ⍀:

```
⍀: (((⍴⍴ω)-⍴⍴α)⊖⍳⍴⍴Z)⍉Z←(((⍴⍴ω)↓⍴α),⍴ω)⍴ω
```

The first (⍴⍴α)-⍴⍴ω coordinates of Z index subarrays which are ω. In Z, ω has been replicated ×/(⍴⍴ω)↓⍴α times. The left argument of transpose ⍉ places the new coordinates in the proper location.

Assuming that ω contains no unit-length axes, axis replication of ω to the shape of α can be defined in terms of functions ⌿ and ⍀:

```
((⍴⍴ω)↓⍴α) ⍀ ((⍴⍴ω)+⌈.5+⍳(⍴⍴α)-⍴⍴ω) ⌿ ω
```

Examples:

```
      A
1 2

      ⍴A
2

      B
1 2 3
4 5 6

      ⍴B
2 3

      C
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      ⍴C
2 3 4
```

```
        B  AR  A
1  1  1
2  2  2

        C  AR  B
1  1  1  1
2  2  2  2
3  3  3  3

4  4  4  4
5  5  5  5
6  6  6  6

        C  AR  A
1  1  1  1
1  1  1  1
1  1  1  1

2  2  2  2
2  2  2  2
2  2  2  2
```

THE LINEAR PROGRAM MODELING PROBLEM

Given the description of a process to be analyzed by linear programming (that is, a model) and the data for a specific instance of that model, the linear program modeling problem is to translate the model into an LP model language that can be executed to produce an LPO problem. Several questions arise in this context: First, what is a good representation for an LPO problem? Second, what exactly is the data for an LP problem? What is its form, what are its sources, and what relationship, if any, does it bear to the model of the problem? And finally, given the model of an LP problem and a structural specification of data for a specific instance of that problem, can we devise an automatic procedure for the systematic construction of an LPO problem for any set of data meeting the specification? This Section address these questions.

Most LP problems resulting from modeling real processes have several characteristics that we want to exploit.

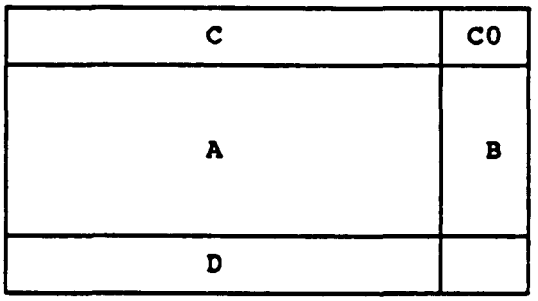
- LPO problems are sparse; that is, on the average, most activities appear in only a few constraints.
- The nonzero coefficients of an LPO problem arise from a set of irreducible data, either directly or by intermediate calculation, that is generally small in relation to the LPO problem.

- The structure of most LPO problems, which is a mathematical abstraction of a process, contain substructures that are replicated many times, either exactly or with slight modification, and that are abstractions of the same or similar subprocesses.
- Structural relationships about sets of activities and constraints, generally explicit in an LP model, are reflected in the structure of the resulting LPO problem.

In this Section, we show an efficient array structure for representing LPO problems. It takes advantage of sparse structure, but more important, it is a good target for the procedure we develop for building LPO problems. We then identify the data that comprise a sparse LPO problem, and introduce a partitioning scheme that decomposes an LPO problem into a series of subproblems. The partitioning scheme and its effect are demonstrated on a classical LP problem. We then introduce the concept of distribution, a process for the systematic selective specification of LPO problems.

PROBLEM REPRESENTATIONS

The classical tableau representation of an LPO problem is a bivalent array *M* constructed from the arrays A, B, C, C0 and D in relations (1.1), (1.2) and (1.3) defined in "Linear Programming" on page 8:



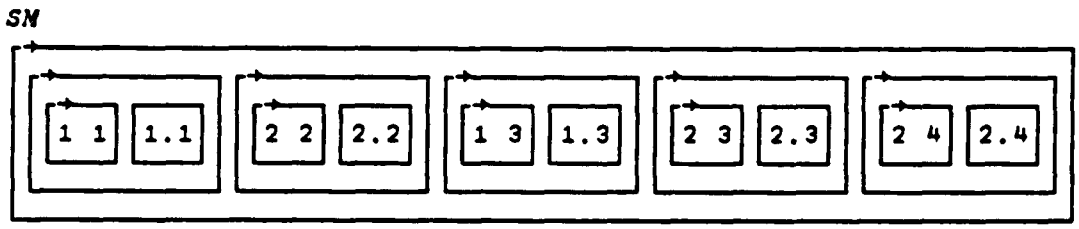
The rows of *M* represent the objective function, the constraint set, and the activity bounds. The columns of *M* represent the activities and requirements.

While convenient for pedagogical purposes, this tableau representation is inefficient for computational procedures primarily because most LP models are sparse. A better representation is obtained by packing the tableau *M* in sparse representation. One such sparse representation stores *M* as a list *SM*, each item of which is a list of length 2. A typical item of *SM*, say ⊃*SM*, has the form ((*I J*) *D*). The first item of a typical item of *SM* is a list of length 2 giving an address in *M* (that is, a row index *I* and column index *J* in *M*). The second item of a typical item of *SM* is *D*, the coefficient *M*[*I*;*J*]. Only the nonzero coefficients of *M* are stored in *SM*. An item ((*I J*) *D*) of *SM* is a problem element.

The following example shows the relationship between *M* and *SM*. If *M* is the matrix

```
1.1 0   1.3 0
0   2.2 2.3 2.4
```

then *SM* is the list of problem elements, shown here in pictorial format:



To construct an LPO problem, it is enough to build the list *SM*. But from what do we build *SM*? The leaves of *SM* consist of two types of data: identifier data that are ordinal numbers and form the address portion of the items of *SM*, and problem data that are arbitrary real numbers and form the coefficient portion of the items of *SM*. We are interested in the values, structure and source of identifier and problem data.

DATA

A sparse LPO problem, represented by the list *SM*, whose typical item is the problem element ((*I J*) *D*), consists of identifier data (the *I* and *J*) and problem data (the *D*).

Problem data

The sources of problem data are varied and depend on the underlying process being modeled. Such data may originate as simple arrays (for example, lists and tables of numbers), fields of records in a file, results of computation by a subroutine or function, or some combination of these sources. Problem data may be used "as is" from a data source, or it may be obtained from a calculation using

other problem data. Regardless of the origin, we make the assumption that, on demand, these data can be presented to us as rectangular, possibly nested arrays.

Identifier data

In the simplest terms, identifier data consists of two lists: a list of row identifiers *IM* and a list of column identifiers *JM*. In terms of the tableau *M* representation of an LPO problem, row identifiers consist of the indices of the rows of *M* (that is, *IM* ↔ ⍳1↑⍴*M*) and column identifiers consist of the indices of the columns of *M* (*JM* ↔ ⍳1↓⍴*M*). One item from *IM* (say *I*) and one item from *JM* (say *J*) addresses a unique item in *M* (that is, *M*[*I*;*J*]). In terms of the underlying model of an LPO problem, the array *JM* indexes the activities and requirements; the array *IM* indexes the constraints and bounds.

Given the list structure of *IM* and *JM*, a procedure for constructing *SM* is: take an item from *IM* and an item from *JM* (this gives an address in *M*), and a nonzero item from a problem data array; these three datum constitute a problem element, an item of *SM*. Unfortunately, with *IM* and *JM* represented as simple lists, this procedure amounts to building *SM* one item at a time. There is no useful structural information in *IM* and *JM* that allow the systematic construction of aggregates of items of *SM*. On the other hand, problem data, regardless of their source, are structured arrays; they occur as simple lists and tables and possibly more richly structured data aggregates, and their structure is usually dictated by the underlying mathematical model of the problem.

We want to find a way of giving this kind of model-related structure to identifier data as well, treating the index arrays *IM* and *JM* not as simple lists but as structured data aggregates.

We now define a way of structuring identifier data by partitioning and show how this structure in turn partitions the tableau representation of an LPO problem. In "Constructing Problems: An Example" on page 74, we see the role this partitioning scheme plays in modeling problems.

PROBLEM PARTITIONING

Consider row and column identifier data represented by arrays *IA* and *JA*, respectively. *IA* and *JA* are lists; the items of *IA* and *JA* are simple arrays of arbitrary valence; and *IA* and *JA* are related to *IM* and *JM*:

```
IM ≡ ∪ IA
JM ≡ ∪ JA
```

That is, the main order of the leaves of *IA* and *JA* are *IM* and *JM*, respectively. We say that *IA* and *JA* are array partitions of *IM* and *JM*, respectively. For example, for an LPO problem with 10 rows,

```
IM ←→ 1 2 3 4 5 6 7 8 9 10 .
```

One possible *IA* is

```
IA ←→ (1 2) 3 (2 2⍴4 5 6 7) 8 (9 10)
```

If simple lists *IM* and *JM* are the row and column indices, respectively, of matrix *M*, then *IA* and *JA*, the array partitions of *IM* and *JM*, induce a partition on *M* into submatrices to produce *MA*. The

array *MA* is a matrix; the items of *MA* are simple arrays with valence not greater than 2. The structure of *MA* is related to the structure of *IA* and *JA*:

$$\rho MA \leftrightarrow (\rho IA),\rho JA \qquad (3.1)$$

That is, *MA* has ρIA rows and ρJA columns.

$$,M \leftrightarrow \cup,"MA$$

The main order of *M* is the main order of *MA* after raveling each item. The items of *MA* are related to subarrays of *M* as indexed by items of *IA* and *JA*:

$$(\subset K\ L)\supset MA \leftrightarrow M[,K\supset IA;,L\supset JA]$$

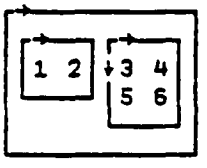
for all *K* and *L* for which $K\supset IA$ and $L\supset JA$ are defined. Indexing the rows of *M* with a listing of an item of *IA* and the columns of *M* with a listing of an item of *JA* is an item of *MA* with path $\subset K\ L$.

For example, let *M*, *IM* and *JM* have the following values:

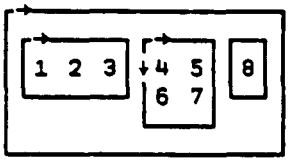
```
      M
17  6  9 11 10  6  4  4
12 17  1 11 10 20 15 12
18 13 17  4  5 15  3  2
 6  1  9  1 15 19  5  4
 7 18 14  4 14  8  8 10
 3 12 17 12 20 12  3 20

      IM
1 2 3 4 5 6

      JM
1 2 3 4 5 6 7 8
```

Now let *IM* and *JM* be partitioned, forming *IA* and *JA*, shown in pictorial format:

IA

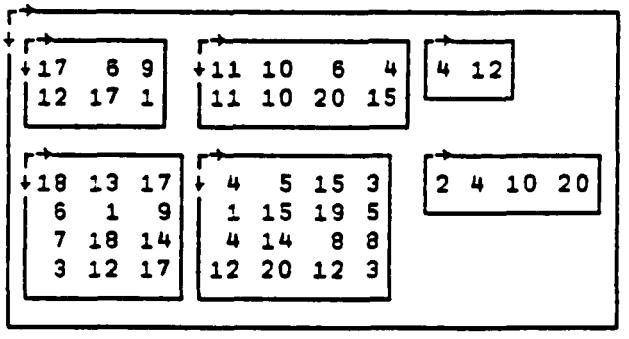


JA



Partitioning *IN* and *JN* to form *IA* and *JA* partitions *N* to form *MA*, shown here in pictorial format:

MA



How the row and column identifier data *IN* and *JN* of an LPO problem are partitioned to form *IA* and *JA* depends on the underlying mathematical model for the problem being analyzed. We introduce this concept using a familiar LP problem.

CONSTRUCTING PROBLEMS: AN EXAMPLE

Consider a single commodity transportation problem with three sources and four destinations. The sources have an availability of commodity given by a list *AVL* where 3 ↔ ⍴*AVL*. The destinations have a requirement of commodity given by a list *REQ* where 4 ↔ ⍴*REQ*. The cost of moving one unit of commodity from sources to destinations is given by a matrix *CST* where 3 4 ↔ ⍴*CST*. The objective is to find a

minimum-cost transportation schedule *X* where 3 4 ↔ ⍴*X* that meets all destination requirements without exceeding source availabilities.

Using the function *GRF* defined in "The Generalized Reduction Function" on page 57, an LP model for this problem has the following form: find the array *X* with shape 3 4 that will minimize

```
1 2 GRF X CST                                          (3.2)
```

subject to

```
∧/AVL≥2 GRF X 1                                        (3.3)
∧/(-REQ)≥1 GRF X ¯1                                    (3.4)
```

The nonnegativity of *X* is implicitly assumed.

If the arrays *AVL*, *REQ* and *CST* have the following values --

```
      AVL
7 3 6

      REQ
4 3 7 2

      CST
 2 12  7  8
 4  1 11 11
15  6  8 13
```

-- then the tableau representation of this problem is shown pictorially as

2	12	7	8	4	1	11	11	15	6	8	13		
1	1	1	1									≤	7
				1	1	1	1					≤	3
¯1				¯1				¯1	1	1	1	≤	6
	¯1				¯1			¯1		¯1		≤	¯4
		¯1				¯1					¯1	≤	¯3
			¯1				¯1					≤	¯7
											¯1	≤	¯2

Relation (3.2) gives rise to one constraint (the objective function constraint), relation (3.3) gives rise to 3 constraints (the inequality constraints on availabilities) and relation (3.4) gives rise to 4 constraints (the inequalities on requirements).

In simple list form, the row and column identifier data *IM* and *JM* for this problem are ⍳8 and ⍳13, respectively. To induce a partitioning on *IM* and *JM* to form *IA* and *JA*, we note that this LP problem consists of three classes of constraints: the objective function constraint (with index 1); three constraints on availabilities (with indices 2 3 4); and four constraints on requirements (with indices 5 6 7 8). Hence, a partition on *IM* is

IA ←→ *OBJI AVLI REQI*

where

OBJI
1

AVLI
2 3 4

REQI
5 6 7 8

Likewise, the tableau representation of this problem consists of two classes of columns: 12 corresponding to activities (with indices ⍳12) and one corresponding to the right-hand-side (with index 13). Hence, a partition of *JM* is

JA ←→ *XJ RHSJ*

where

```
      XJ
1  2  3  4
5  6  7  8
9 10 11 12

      RHSJ
13
```

Note that *XJ* has the same shape as the activity array *X*.

This partitioning of identifier data induces a partitioning on *M*, the tableau representation of the problem, to form *MA*. By (3.1),

$$\rho MA \leftrightarrow (\rho IA),\rho JA \leftrightarrow 3\ 2 \tag{3.5}$$

If we identify the items of *MA* as

$$MA \leftrightarrow 3\ 2\rho M1\ M2\ M3\ M4\ M5\ M6 \tag{3.6}$$

then *MA* can be represented schematically as

MA

M1	*M2*
M3	*M4*
M5	*M6*

where

```
      M1
2 12 7 8 4 1 11 11 15 6 8 13

      M2
0

      M3
1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1

      M4
7 3 6
```

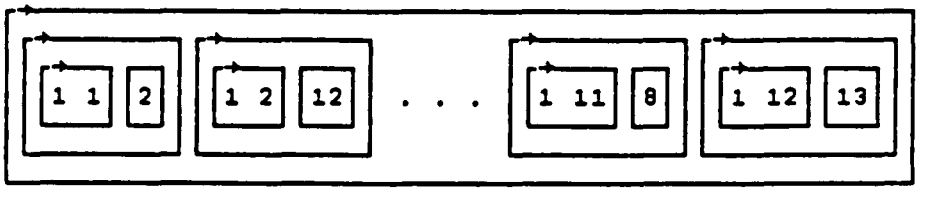
```
      M5
¯1  0  0  0 ¯1  0  0  0 ¯1  0  0  0
 0 ¯1  0  0  0 ¯1  0  0  0 ¯1  0  0
 0  0 ¯1  0  0  0 ¯1  0  0  0 ¯1  0
 0  0  0 ¯1  0  0  0 ¯1  0  0  0 ¯1

      M6
¯4 ¯3 ¯7 ¯2
```

Given the set of problem data *AVL*, *REQ* and *CST*, row identifier data *OBJI*, *AVLI* and *REQI*, and column identifier data *XJ* and *RHSJ*, how do we build a sparse representation *SM* consisting of problem elements for this problem? Observe that, for example, problem elements of the form ((*I J*) *D*) for submatrix *M*1 involve selections from three arrays: a row identifier array (*OBJI*), a column identifier array (*XJ*), and a data array (*CST*). Hence, a procedure *P*1 to construct problem elements in *M*1 is a trivalent function

*Z*1 ← *P*1 (*OBJI XJ CST*)

where *Z*1 is the list of problem elements representing *M*1:

*Z*1



In a similar manner, we can identify procedures *P*3, *P*4, *P*5 and *P*6 for producing the problem elements of *M*3, *M*4, *M*5 and *M*6, respectively. (*M*2 is all zero and requires no problem elements.)

*P*3-*P*6, their arguments and results, would be:

```
Z3 ← P3 (AVLI XJ 1)
Z4 ← P4 (REQI XJ ¯1)
Z5 ← P5 (AVLI RHSJ AVL)
Z6 ← P6 (REQI RHSJ REQ)
```

These functions would produce results Z1, Z3, Z4, Z5 and Z6, and the sparse representation of the problem *SM* is

$$SM \leftrightarrow Z1,Z3,Z4,Z5,Z6 \qquad (3.7)$$

The functions *P1*, *P3*, *P4*, *P5* and *P6* are examples of distribution functions that systematically produce lists of problem elements of a sparse matrix representation. "Distribution Functions" on page 88 gives rigorous definitions of a family of functions for carrying out several types of distribution operations. Here we want to give an intuitive treatment of the two most common distribution functions, generalized reduction distribution (of which *P1*, *P3* and *P5* are examples), and scalar distribution (of which *P4* and *P6* are examples). Our aim is to show how these functions can be used to build a sparse representation of an LPO problem, and in particular, the sparse representation *SM* in expression (3.7) of the transportation problem model given by (3.2), (3.3) and (3.4).

DISTRIBUTION

Most LP model formulations consist of multiple instances of generalized reduction as, for example, expressions (3.2), (3.3) and (3.4) for the transportation problem. LP problem solving routines, however, expect problems to be represented in a form compatible with vector-vector and matrix-vector +.× inner product, either explicitly as a tableau *M* or implicitly as a sparsely represented

tableau *SM*. The generalized reduction distribution operation transforms a computation in generalized reduction form into a computation in terms of +.× inner product.

In particular, consider the following linear system: given the arrays *A*, *B* and *C*, find the array *Y* such that

$$B \equiv A\ GRF\ (Y\ C) \qquad (3.8)$$

where *Y* and *C* are conformable for the generalized reduction function *GRF* (see "The Generalized Reduction Function" on page 57), and *A* is a list of coordinate indices of *Y*. To transform (3.8) into a matrix-vector +.× inner product, we require the matrix *Q* such that

$$(,B) \equiv Q +.\times ,Y \qquad (3.9)$$

That is, the +.× inner product of *Q* and the ravel of *Y* gives the ravel of *B*.

This transformation is accomplished by the generalized reduction distribution function *GRD* that has the following syntax:

$$Q \leftarrow A\ GRD\ (YI\ C) \qquad (3.10)$$

where *A* and *C* are the same as in (3.8) and *YI* is an array of the indices of the ravel of *Y* in (3.8):

$$YI \leftarrow (\rho Y)\rho \iota \times /\rho Y \qquad (3.11)$$

Thus *YI* and *Y* have the same rank and shape. The conformability requirements of *GRD* are exactly the same as those of *GRF*, and the result *Q* of (3.10) is the required matrix *Q* in (3.9).

Let $\underline{N}$ be the index producing function

$$\underline{N}:(\rho\omega)\rho \iota \times /\rho\omega$$

Example:

```
      Y
2 0 1 0
0 2 0 1
1 1 0 0

      N Y
1  2  3  4
5  6  7  8
9 10 11 12
```

The relationship between *GRF* and *GRD*, in terms of (3.8), (3.9) and (3.10) is:

If

$$B \equiv A\ GRF\ Y\ C \tag{3.12}$$

then

$$(,B) \equiv (A\ GRD\ (\underline{N}\ Y)\ C) +.\times ,Y \tag{3.13}$$

Examples:

```
      Y
2 0 1 0
0 2 0 1
1 1 0 0

      C
10  20  30  40
50  60  70  80
90 100 110 120

      1 GRF Y C
110 220 30 80

      Q←1 GRD (N Y) C

      Q
10  0  0  0 50  0  0  0 90   0   0   0
 0 20  0  0  0 60  0  0  0 100   0   0
 0  0 30  0  0  0 70  0  0   0 110   0
 0  0  0 40  0  0  0 80  0   0   0 120

      Q+.×,Y
110 220 30 80
```

```
      2 GRF Y C
50 200 190

      Q←2 GRD (N Y) C
      Q
10 20 30 40  0  0  0  0  0   0   0   0
 0  0  0  0 50 60 70 80  0   0   0   0
 0  0  0  0  0  0  0  0 90 100 110 120

      Q+.×,Y
50 200 190

      1 2 GRF Y C
440

      Q←1 2 GRD (N Y) C
      Q
10 20 30 40 50 60 70 80 90 100 110 120

      Q+.×,Y
440
```

We can now extend *GRD* to produce a sparse matrix result. Consider the function *GRDS* with syntax

SM ← A GRDS (R C B)

(A function similar to *GRDS* is defined formally in "Generalized Reduction Distribution" on page 90; here we are less rigorous.) The purpose of *GRDS* is to produce a list of problem elements *SM*. The right argument is a list containing three items. The first item *R* is an array of row indices for the matrix to be produced. The second item *C* is an array of column indices for the matrix to be produced. The third item *B* is an array of coefficient data. A typical item of *SM* is ((*I J*) *D*) where *I∈R*, *J∈C* and *D∈B*. The left argument *A* is a list of coordinate indices of *C*.

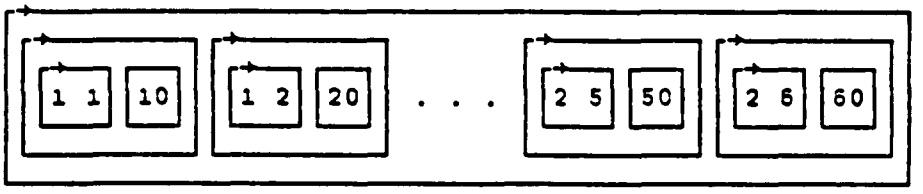
The result *SM* is a list of problem elements for a sparse representation of a matrix derived from a generalized reduction computation by generalized reduction distribution.

Examples:

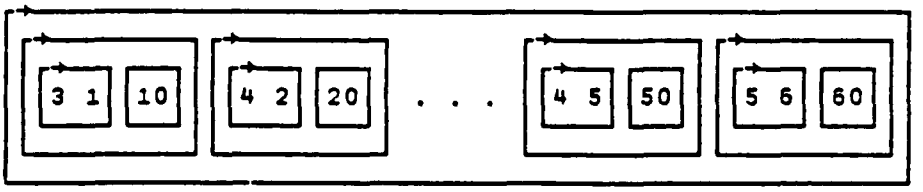
```
      K
1 2 3
4 5 6

      C
10 20 30
40 50 60

      2 GRD K C
10 20 30  0  0  0
 0  0  0 40 50 60

      1 GRD K C
10  0  0 40  0  0
 0 20  0  0 50  0
 0  0 30  0  0 60

      1 2 GRD K C
10 20 30 40 50 60

      S1← 2 GRDS (1 2) K C
      S2← 1 GRDS (3 4 5) K C
      S3← 1 2 GRDS 6 K C
```

The arrays *S*1, *S*2 and *S*3 are shown in pictorial format:

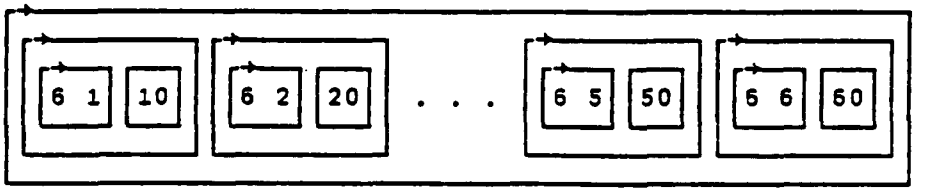
*S*1



*S*2



*S*3



The matrix represented by the array (*S1*,*S2*,*S3*) is

```
10 20 30  0  0  0
 0  0  0 40 50 60
10  0  0 40  0  0
 0 20  0  0 50  0
 0  0 30  0  0 60
10 20 30 40 50 60
```

Scalar distribution

The scalar distribution function has syntax

Z ← *SDS* (*R C D*)

(A scalar distribution function similar to *SDS* is defined formally in "Scalar Distribution" on page 105.) The right argument is a list containing three items. The first item *R* is an array of row indices for a matrix *M*. The second item *C* is an array of column indices for *M*. The third item *D* is any numeric array. *R*, *C* and *D* must all have the same shape; if any is a singular array, it is extended to the shape of the other array(s). *Z* is a list of problem elements for *M*, formed by the selection of corresponding items of *R*, *C* and *D*.

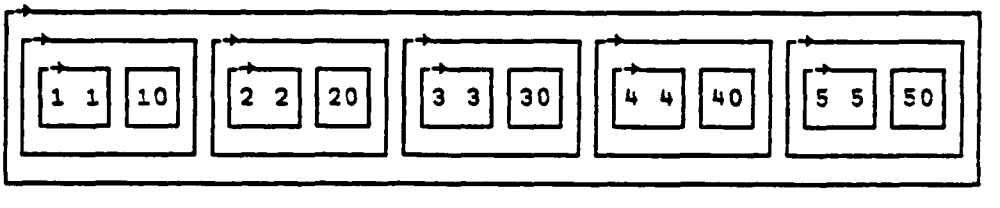
Example:

The expression

S ← *SDS* (⍳5) (⍳5) (10×⍳5)

creates the array *S*, shown in pictorially as

S



which is a sparse representation of the matrix

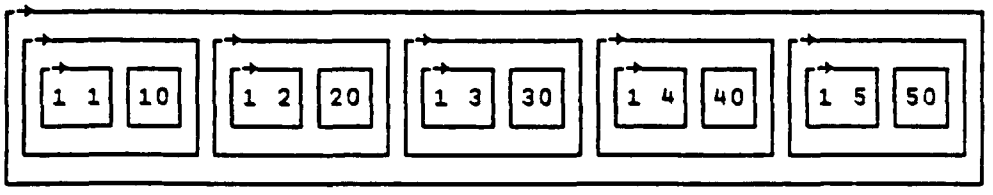
```
10  0  0  0  0
 0 20  0  0  0
 0  0 30  0  0
 0  0  0 40  0
 0  0  0  0 50
```

The expression

```
V ← SDS 1 (⍳5) (10×⍳5)
```

produces the array

V



which is the sparse representation of the list

```
10 20 30 40 50
```

We are now ready to build the transportation problem using the generalized reduction distribution and scalar distribution functions *GRDS* and *SDS*.

As before, we seek a sparse representation *SM* of the transportation problem tableau, composed as

```
SM ← Z1,Z3,Z4,Z5,Z6
```

using row identifier data *OBJI*, *AVLI* and *REQI*, column identifier data *XJ* and *RHSJ*, and problem data *AVL*, *REQ* and *CST*:

```
      AVL
7 3 6

      REQ
4 3 7 2

      CST
 2 12  7  8
 4  1 11 11
15  6  8 13
```

```
      OBJI
1

      AVLI
2 3 4

      REQI
5 6 7 8

      XJ
1   2   3   4
5   6   7   8
9  10  11  12

      RHSJ
13
```

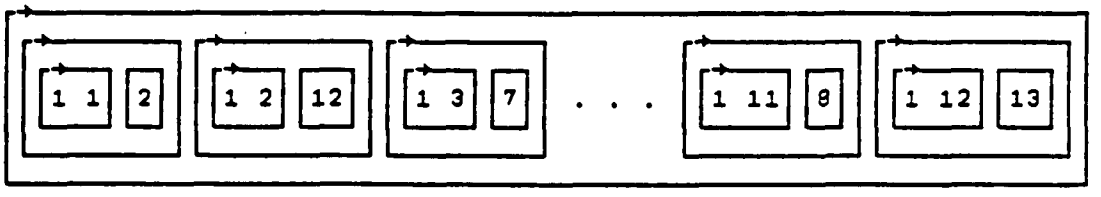
The following sequence produces the desired result:

```
Z1 ← 1 2 GRDS (OBJI XJ CST)
Z3 ← 2 GRDS (AVLI XJ 1)
Z4 ← 1 GRDS (REQI XJ ¯1)
Z5 ← SDS (AVLI RHSJ AVL)
Z6 ← SDS (REQI RHSJ REQ)
```

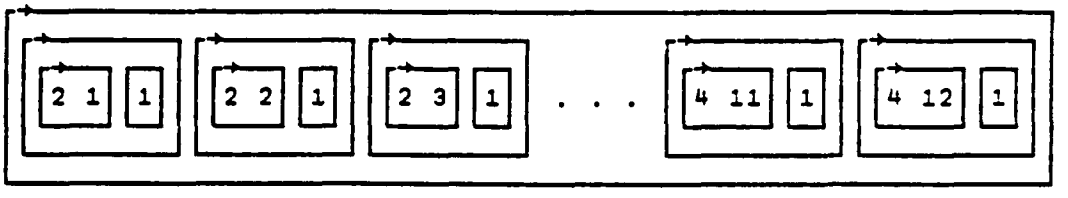
For example, Z3 is a sparse representation of *M*3, that submatrix of *M* with row indices *AVLI* and column indices *XJ*, and containing nonzero coefficients 1.

The arrays Z1, Z3, Z4, Z5 and Z6 are shown in pictorial format:

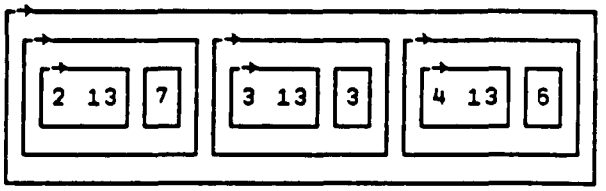
Z1



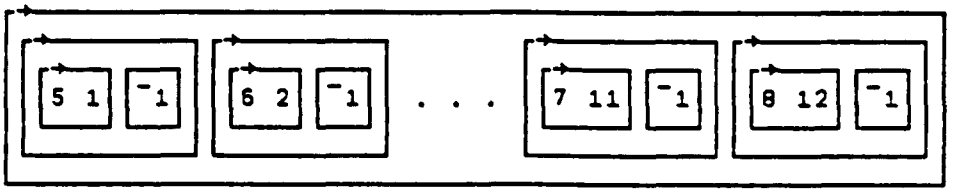
Z3



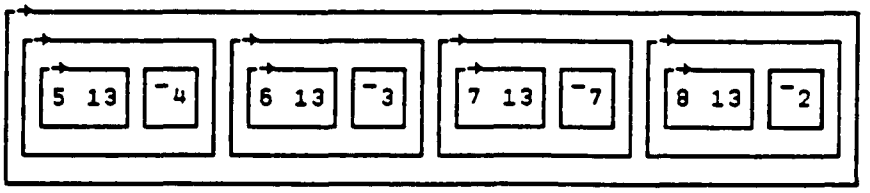
Z4



Z5



Z6



In the next Section, we define a set of distribution functions, including generalized reduction and scalar distribution, for describing LP models and building LPO problems.

DISTRIBUTION FUNCTIONS

This Section describes the family of distribution functions for building sparse matrix representations of LPO problems. With one exception, distribution functions all take three arguments; generalized reduction distribution, in addition, takes a list of axis indices. The results of distribution functions are arrays of problem elements.

The following table summarizes the names, syntax and argument descriptions of the distribution functions:

Name	Syntax
Generalized Reduction distribution	*Z←A GR (R C D)*
Scalar Distribution	*Z←SD (R C D)*
Assign Distribution	*Z←AD (R C D)*
Scalar Network distribution	*Z←SN (C RF RT)*
Cartesian Network distribution	*Z←CN (C RF RT)*
where: *R, RF, RT* - Row identifier arrays *C* - Column identifier array *D* - Problem data array *A* - Axis index list *Z* - Resulting problem elements	

Results of distribution functions

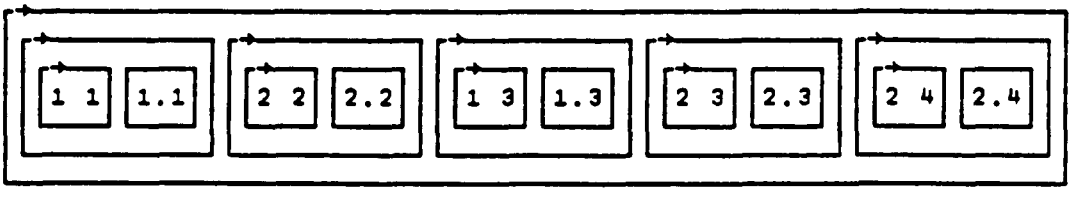
Only explicit results are produced by distribution functions; there are no side effects. The form of problem elements produced by distribution functions described in this Section are a

rearrangement of the sparse matrix representation introduced in "Problem Representations" on page 68. Previously, a sparse matrix was represented by a list whose typical item is ((*I J*) *D*), where *I* is a row index, *J* a column index, and *D* a data item. The explicit result of distribution functions described here is a table-list sparse matrix representation. Let *STL* be such a representation. *STL* is a list of two items. The first item of *STL* (say *T*) is a numeric table of integers with two columns; the second item of *STL* (say *L*) is a numeric list. The number of rows of *T* is equal to the length of *L*. A row of *T* gives a tableau address (that is, a row and column index) of the corresponding element in *L*.

For example, in the sparse matrix representation introduced in "Problem Representations" on page 68, the matrix

```
1.1 0   1.3 0
0   2.2 2.3 2.4
```

was represented by the following array, shown in pictorial form:



The corresponding table-list representation is

```
1 1   1.1 2.2 1.3 2.3 2.4
2 2
1 3
2 3
2 4
```

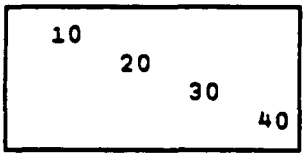
Notes on examples

The definition and explanation of each distribution function is accompanied by several examples. As an aid to interpreting the

action of a function, we will generally use the array result as argument to the *TABLE* function, which builds a tableau representation of the result. *TABLE* elides elements with value zero in the resulting matrix, and also frames the result.

Example:

```
      STL
1 1    10 20 30 40
2 2
3 3
4 4
      TABLE STL
```



GENERALIZED REDUCTION DISTRIBUTION

Generalized reduction distribution transforms a generalized reduction computation involving application of the derived function +.× over selected axes to a matrix-vector inner product computation.

Syntax: *Z ← A GR (R C D)*

The right argument is a list containing three items. The first item *R* is either i) a simple array of row identifiers, or ii) an array whose items are simple arrays of row identifiers. The second item *C* is either i) a simple array of column identifiers, or ii) an array whose items are simple arrays of column identifiers. The third item *D* is either i) a simple array of problem data, or ii) an array whose items are simple arrays of problem data. The left argument *A* is a simple integer scalar or list that indexes either i)

the coordinates of *C* if *C* is a simple array, or ii) the coordinates of the items of *C* if *C* is a nonsimple array. The result *Z* is an array of problem elements, formed from the leaves of *R*, *C* and *D*, in table-list sparse representation.

Conformability Requirements:

If *R*, *C*, and *D* are simple arrays, then *C* must cover *D*. (See "Cover and Match" on page 56.) If *N* is the list of coordinate indices of *C* not in *A* (that is, *N*≡(~(⍳⍴⍴*C*)∈*A*)/⍳⍴⍴*C*), then

$$(\rho R)\equiv(\rho C)[N],C \; \underline{SX} \; D \qquad (4.1)$$

where $\underline{SX}$ is the shape excess function introduced in "Cover and Match" on page 56:

$\underline{SX}$:((⍴⍴α)↑⍴ω),(⍴⍴ω)↑⍴α

If *R*, *C*, and *D* are nonsimple arrays, then they must have the same shape, subject to replication of singular arrays, and the conformability requirement stated above must hold for corresponding items of *R*, *C*, and *D*.

Definition:

If *R*, *C*, and *D* are simple arrays, then *CR* creates problem elements from items of *R* and subarrays of items of *C* and *D*. After the proper extension of *D* described below, the selection of subarrays in *C* and *D* is determined by the coordinate indices mentioned in *A*. If either *C* or *D* is a scalar, it is replicated to the shape of the other array.

From the conformability requirement, *C* covers *D*. If *C* and *D* do not match, axis extension is carried out on *D*, extending its unit-length axes to the the corresponding axes in *C*. After axis extension, (*K*↑⍴*C*)≡*K*↑⍴*D*, where *K* is (⍴⍴*C*)⌊⍴⍴*D*.

After axis extension on *D*, if *C* and *D* differ in rank, the array with the smaller rank is replicated to the same shape as the array with the larger rank. For example, suppose (⍴⍴*C*)>⍴⍴*D*. Then, if *T* ↔ (⍴⍴*D*)-⍴⍴*C*, the replication of *D* to the shape of *C* is

```
D←(T⌽⍳⍴⍴D)⍉D←(((⍴⍴D)↑⍴C),⍴D)⍴D
```

If *R* is a scalar, then *C* and *D* have the same shape (after appropriate axis extension and replication); *GR* creates problem elements from *R* and corresponding items of *C* and *D*.

If *R* is nonscalar, let *P* be a scalar path defined on *R*. After appropriate axis extension and replication, (⍴*C*)≡⍴*D*. Let *N* be the list of coordinate indices of *C* (or *D*) <u>not</u> in *A*. Problem elements are formed from the row identifier

```
P⊃R ,                                                    (4.2)
```

each of the elements of the subarray of column identifiers

```
(⊃P)⌷[N]C ,                                              (4.3)
```

and each of the elements of the subarray of problem data

```
(⊃P)⌷[N]D                                                (4.4)
```

for all *P* for which *P*⊃*R* is defined.

Expressions (4.3) and (4.4) index subarrays of *C* and *D* along the axes listed in *A*.

If *R*, *C* and *D* are nonsimple arrays, then generalized reduction distribution is applied itemwise to the items of *R*, *C* and *D* after appropriate replication of singular items.

The result *Z* is an array of problem elements with two items. The first item of *Z* is a two-column table of tableau addresses,

formed from the leaves of *R* and *C*. The second item of *Z* is a list of tableau coefficients formed from the leaves of *D*. *K* is the number of problem elements produced, where

```
K←(⍴,R)××/(⍴C)[A]                                    (4.5)
```

if *R*, *C*, and *D* are simple arrays, or

```
K←(⍴¨,¨R)+.××/¨⊃¨(⊂⊂A)[]¨C                            (4.6)
```

if *R*, *C*, and *D* are nonsimple arrays. Thus,

```
K←1↑⍴⊃Z
```

and

```
K←⍴2⊃Z.
```

The result *Z* represents a tableau with shape `∪⍴¨∪¨R C`.

"Appendix 2: Distribution Function Algorithms" on page 170 gives the generalized reduction distribution algorithm.

Examples:

The following examples of generalized reduction distribution are array analogues of typical mathematical expressions involving summation along coordinates of array products. In each case, the example includes the array product in traditional mathematical notation.

Example set 1 - Column identifier array and data array of equal rank and shape.

Example 1:

```
      R
1 2 3 4

      C
1  2  3  4
5  6  7  8
9 10 11 12
```

```
       D
10 11 12 13
14 15 16 17
18 19 20 21

       1 GR R C D
 1  1   10 14 18 11 15 19 12 16 20 13 17 21
 1  5
 1  9
 2  2
 2  6
 2 10
 3  3
 3  7
 3 11
 4  4
 4  8
 4 12
```

TABLE 1 GR R C D

10				14				18			
	11				15				19		
		12				16				20	
			13				17				21

Traditional notation:

$$\sum_i d_{ij}x_{ij}, \quad \text{all } j$$

Example 2:

```
       R
1 2 3

       C
1  2  3  4
5  6  7  8
9 10 11 12

       D
10 11 12 13
14 15 16 17
18 19 20 21
```

```
TABLE 2 GR R C D
```

```
10 11 12 13
            14 15 16 17
                        18 19 20 21
```

Traditional notation:

$$\sum_{j} d_{ij}x_{ij}, \quad \text{all } i$$

Example 3:

```
TABLE 1 2 GR 1 C D
```

```
10 11 12 13 14 15 16 17 18 19 20 21
```

Traditional notation:

$$\sum_{ij} d_{ij}x_{ij}$$

Example set 2 - Replication of column identifier array.

Example 4:

```
       R
1  2  3  4
5  6  7  8
9 10 11 12

       C
1 2 3
4 5 6

       D
10 11 12 13
14 15 16 17
18 19 20 21

22 23 24 25
26 27 28 29
30 31 32 33
```

TABLE 1 GR R C D

10				22			
11				23			
12				24			
13				25			
	14				26		
	15				27		
	16				28		
	17				29		
		18				30	
		19				31	
		20				32	
		21				33	

Traditional notation:

$$\sum_{1} d_{ijk} x_{ij}, \quad \text{all } j, k$$

Example 5:

```
       R
1  2  3  4
5  6  7  8

       C
1  2  3
4  5  6

       D
10  11  12  13
14  15  16  17
18  19  20  21

22  23  24  25
26  27  28  29
30  31  32  33
```

TABLE 2 GR R C D

10	14	18					
11	15	19					
12	16	20					
13	17	21					
			22	26	30		
			23	27	31		
			24	28	32		
			25	29	33		

Traditional notation:

$$\sum_{j} d_{ijk} x_{ij}, \quad \text{all } i, k$$

Example 6:

```
        R
1  2  3  4

        C
1  2  3
4  5  6

        D
10  11  12  13
14  15  15  17
18  19  20  21

22  23  24  25
26  27  28  29
30  31  32  33
```

TABLE 1 2 *GR R C D*

10	14	18	22	26	30
11	15	19	23	27	31
12	16	20	24	28	32
13	17	21	25	29	33

Traditional notation:

$$\sum_{ij} d_{ijk} x_{ij}, \quad \text{all } k$$

Example set 3 - Axis extension of data array, replication of column identifier array.

Example 7:

```
        R
1  2  3  4
5  6  7  8

        C
1  2  3
4  5  6
```

```
      D
10 11 12 13

14 15 16 17

      ⍴D
2 1 4
```

```
      TABLE 2 GR R C D
```

10	10	10			
11	11	11			
12	12	12			
13	13	13			
			14	14	14
			15	15	15
			16	16	16
			17	17	17

Traditional notation:

$$\sum_{j} d_{ik} x_{ij}, \quad \text{all } i, k$$

Example 8:

```
      R
1  2  3  4
5  6  7  8
9 10 11 12

      C
1 2 3
4 5 6

      D
10 11 12 13

14 15 16 17

      ⍴D
2 1 4
```

TABLE 1 GR R C D

```
10          14
11          15
12          16
13          17
    10          14
    11          15
    12          16
    13          17
        10          14
        11          15
        12          16
        13          17
```

Traditional notation:

$$\sum_{i} d_{ik} x_{ij}, \quad \text{all } j, k$$

Example 9:

```
        R
1 2 3 4

        C
1 2 3
4 5 6

        D
10 11 12 13

14 15 16 17

        ρD
2 1 4
```

TABLE 1 2 GR R C D

```
10 10 10 14 14 14
11 11 11 15 15 15
12 12 12 16 16 16
13 13 13 17 17 17
```

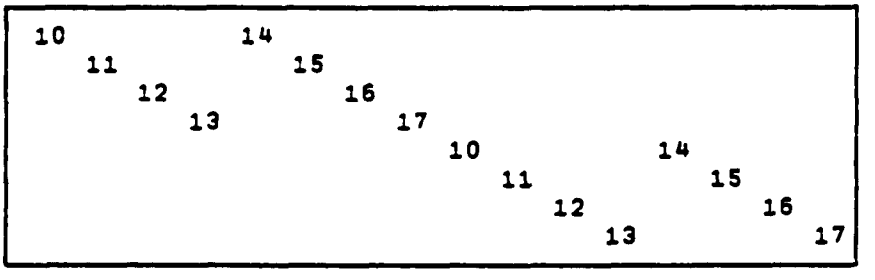
Traditional notation:

$$\sum_{ij} d_{ik} x_{ij}, \quad \text{all } k$$

Example set 4 - Axis extension of data array:

Example 10:

```
      R
1 2 3 4
5 6 7 8

      C
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

      ρC
2 2 4

      D
10 11 12 13
14 15 16 17

      ρD
1 2 4

      TABLE 2 GR R C D
```



Traditional notation:

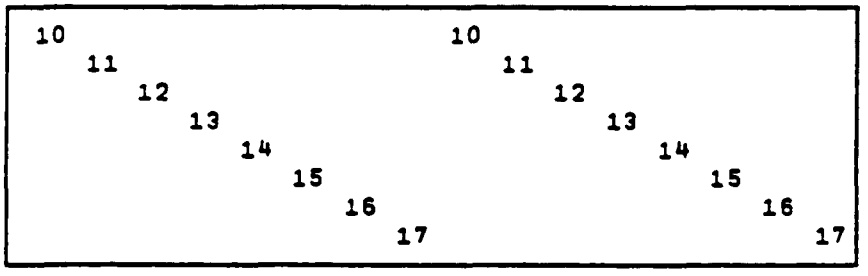
$$\sum_j d_{jk} x_{ijk}, \quad \text{all } i, k$$

Example 11:

```
      R
1 2 3 4
5 6 7 8
```

```
      C
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

      D
10 11 12 13
14 15 16 17

      ρD
1 2 4
```

```
      TABLE 1 GR R C D
```



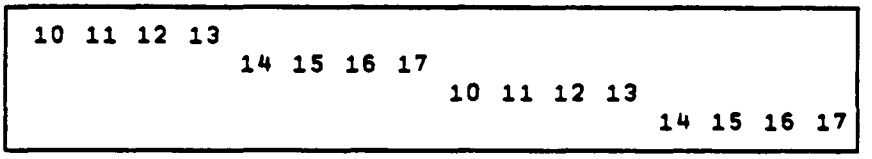
Traditional notation:

$$\sum_{i} d_{jk} x_{ijk}, \quad \text{all } j, k$$

Example 12:

```
      R
1 2
3 4

      C
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

      D
10 11 12 13
14 15 16 17

      ρD
1 2 4
```

```
      TABLE 3 GR R C D
```



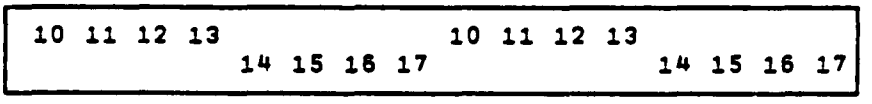
Traditional notation:

$$\sum_{k} d_{jk} x_{ijk}, \quad \text{all } i, j$$

Example 13:

```
      R
1 2

      C
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

      D
10 11 12 13
14 15 16 17

      ρD
1 2 4
```

```
      TABLE 1 3 GR R C D
```



Traditional notation:

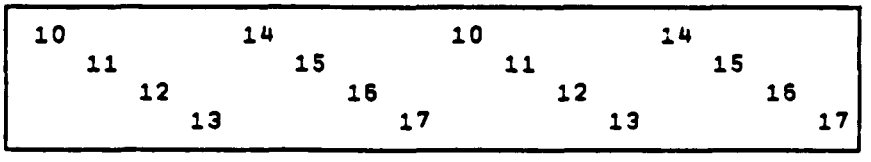
$$\sum_{ik} d_{jk} x_{ijk}, \quad \text{all } j$$

Example 14:

```
      R
1 2 3 4
```

```
      C
 1   2   3   4
 5   6   7   8

 9  10  11  12
13  14  15  16

      D
10  11  12  13
14  15  16  17

      ρD
1  2  4
```

```
      TABLE 1 2 GR R C D
```



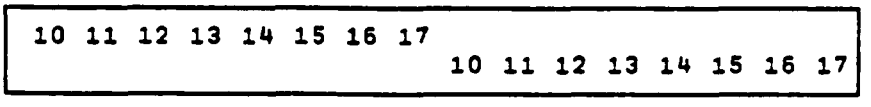
Traditional notation:

$$\sum_{ij} d_{jk} x_{ijk}, \quad \text{all } k$$

Example 15:

```
      R
1  2

      C
 1   2   3   4
 5   6   7   8

 9  10  11  12
13  14  15  16

      D
10  11  12  13
14  15  16  17

      ρD
1  2  4
```

```
      TABLE 2 3 GR R C D
```



Traditional notation:

$$\sum_{jk} d_{jk} x_{ijk}, \quad \text{all } i$$

Example 16:

```
      R
1

      C
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

      D
10 11 12 13
14 15 16 17

      ρD
1 2 4

      TABLE 1 2 3 GR R C D
```

10	11	12	13	14	15	16	17	10	11	12	13	14	15	16	17

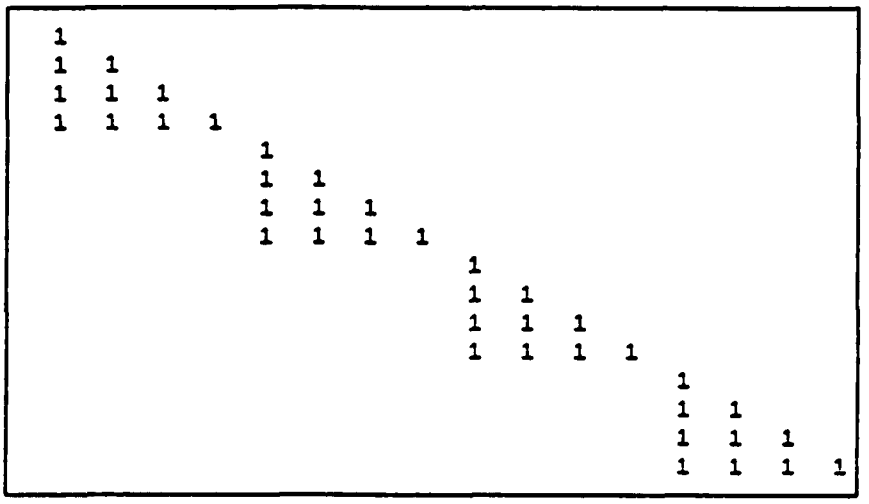
Traditional notation:

$$\sum_{ijk} d_{jk} x_{ijk}$$

Example 17 - Nonsimple arrays:

```
      R
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

      C
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

TABLE 2 GR (⊂[1]R) ((⍳4)∘.+[2]⍨⊂C) 1



SCALAR DISTRIBUTION

Scalar distribution forms problem elements from corresponding items of two identifier data arrays and a problem data array.

Syntax: *Z ← SD (R C D)*

The right argument is a list containing three items. The first item *R* is either i) a simple array of row identifiers, or ii) an array whose items are simple arrays of row identifiers. The second item *C* is either i) a simple array of column identifiers, or ii) an array whose items are simple arrays of column identifiers. The third item *D* is either i) a simple array of problem data, or ii) an array whose items are simple arrays of problem data. The result *Z* is an array of problem elements, formed from the leaves of *R*, *C* and *D*, in table-list sparse representation.

Conformability Requirement:

R, *C* and *D* must have the same shape. If any is a singular array, it is replicated to have the same shape as the other array(s). If *R* and *C* are both singular, *D* must be singular.

If *R*, *C* and *D* are simple arrays then

$$\rho R \leftrightarrow \rho C \leftrightarrow \rho D \qquad (4.7)$$

subject to replication of singular arrays.

If *R*, *C* and *D* are nonsimple arrays, then for all paths *P* defined on *R*, *C* and *D*,

$$\rho P \supset R \leftrightarrow \rho P \supset C \leftrightarrow \rho P \supset D \qquad (4.8)$$

subject to replication of singular items of *R*, *C* and *D*.

Definition:

If *R*, *C* and *D* are simple arrays, then let *P* be a path on *R*, *C* and *D* (subject to previous replication of singular arrays). Scalar distribution creates a problem element consisting of $P \supset R$, $P \supset C$ and $P \supset D$ for all *P* for which $P \supset R$, $P \supset C$ and $P \supset D$ are defined.

If *R*, *C* and *D* are nonsimple arrays, then scalar distribution is applied itemwise to the items of *R*, *C* and *D* after appropriate replication of singular items.

The result *Z* is an array of problem elements with two items. The first item of *Z* is a two-column table of tableau addresses, formed from the leaves of *R* and *C*. The second item of *Z* is a list of tableau coefficients formed from the leaves of *D*. *K* is the number of problem elements produced, where

$$K \leftarrow \supset \lceil / \rho \ddot{\cup} \ddot{\cup} R \; C \; D \qquad (4.9)$$

Thus,

$$K \leftarrow 1 \uparrow \rho \supset Z$$

and

```
K←⍴2⊃Z.
```

The result *Z* represents a tableau with shape `∪⍴⍤∪⍤R C`.

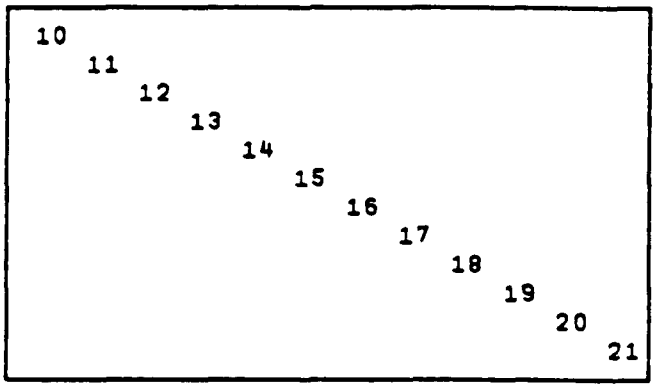
"Appendix 2: Distribution Function Algorithms" on page 170 gives the scalar distribution algorithm.

Examples:

Example 1 - Identifier and problem data arrays of equal rank and shape:

```
      R
1  2  3  4
5  6  7  8
9 10 11 12

      C
1  2  3  4
5  6  7  8
9 10 11 12

      D
10 11 12 13
14 15 16 17
18 19 20 21

      SD R C D
 1  1   10 11 12 13 14 15 16 17 18 19 20 21
 2  2
 3  3
 4  4
 5  5
 6  6
 7  7
 8  8
 9  9
10 10
11 11
12 12
```

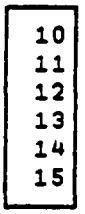
```
      TABLE SD R C D
```



Examples showing replication of identifier arrays.

Example 2:

```
      R
1 2 3
4 5 6

      C
1

      D
10 11 12
13 14 15

            SD R C D
 1 1   10 11 12 13 14 15
 2 1
 3 1
 4 1
 5 1
 6 1

      TABLE SD R C D
```



Example 3:

```
      R
1
```

```
       C
1   2   3   4
5   6   7   8
9  10  11  12

       D
10  11  12  13
14  15  16  17
18  19  20  21

      TABLE SD R C D
```

10	11	12	13	14	15	16	17	18	19	20	21

Example 4 - Replication of data array:

```
       R
1   2   3   4
5   6   7   8
9  10  11  12

       C
1   2   3   4
5   6   7   8
9  10  11  12

       D
11

      TABLE SD R C D
```

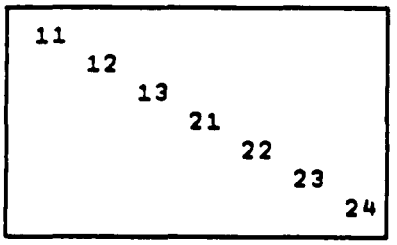
11											
	11										
		11									
			11								
				11							
					11						
						11					
							11				
								11			
									11		
											11

Examples showing nonsimple arrays.

Example 5:

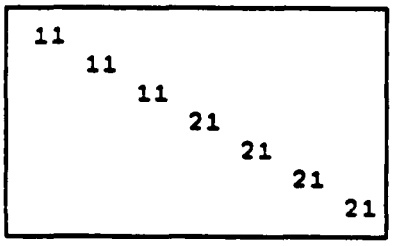
```
       R1
1 2 3
```

```
       R2
4 5 6 7

       C1
1 2 3

       C2
4 5 6 7

       SD (R1 R2) (C1 C2) ((11 12 13) (21 22 23 24))
 1 1    11 12 13 21 22 23 24
 2 2
 3 3
 4 4
 5 5
 6 6
 7 7

       TABLE SD (R1 R2) (C1 C2) ((11 12 13) (21 22 23 24))
```



Example 6:

```
       TABLE SD (R1 R2) (C1 C2) (11 21)
```



Example 7:

```
       R
1 2 3 4

       C
1  2  3  4
5  6  7  8
9 10 11 12

       D
10 11 12 13
```

```
      TABLE SD (⊂R) (⊂[2]C) (⊂D)
┌──────────────────────────────┐
│ 10          10          10   │
│    11          11          11│
│       12          12          12
│          13          13          13
└──────────────────────────────┘
```

Example 8:

```
      TABLE SD (⊂R) (⊂[2]C) (0 10 20 + ⊂D)
┌──────────────────────────────┐
│ 10          20          30   │
│    11          21          31│
│       12          22          32
│          13          23          33
└──────────────────────────────┘
```

ASSIGN DISTRIBUTION

Assign distribution forms problem elements from items of a problem data array and all possible pairs of items from two identifier data arrays.

Syntax: *Z ← AD (R C D)*

The right argument is a list containing three items. The first item *R* is either i) a simple array of row identifiers, or ii) an array whose items are simple arrays of row identifiers. The second item *C* is either i) a simple array of column identifiers, or ii) an array whose items are simple arrays of column identifiers. The third item *D* is either i) a simple array of problem data, or ii) an array whose items are simple arrays of problem data. The result *Z* is an array of problem elements, formed from the leaves of *R*, *C* and *D*, in table-list sparse representation.

Conformability Requirements:

If *R*, *C* and *D* are simple arrays, then

$$\rho D \leftrightarrow (\rho R),\rho C \tag{4.10}$$

If D is a simple scalar, it is replicated to the shape $(\rho R),\rho C$.

If R, C and D are nonsimple arrays, then R, C and D must have the same shape, subject to replication of singular arrays, and for all scalar paths P defined on R, C, and D,

$$\rho P\supset D \leftrightarrow (\rho P\supset R),\rho P\supset C \tag{4.11}$$

subject to replication of singular items of R, C and D.

If R and C are both singular, D must be singular.

Definition:

If R, C and D are simple arrays, then let I and J be scalar paths defined on R and C, respectively. Assign distribution creates a problem element consisting of $I\supset R$, $J\supset C$ and $(\subset(\supset I),\supset J)\supset D$ for all I and J for which $I\supset R$ and $J\supset C$ are defined.

If R, C, and D are nonsimple arrays, this definition is applied to the items of R, C and D after appropriate replication of singular items.

The result Z is an array of problem elements with two items. The first item of Z is a two-column table of tableau addresses, formed from the leaves of R and C. The second item of Z is a list of tableau coefficients formed from the leaves of D. K is the number of problem elements produced, where

$$K=((\rho,R)\times\rho,C)\lceil\rho,D \tag{4.12}$$

if R, C and D are simple arrays, or

$$K=\supset+/((\rho\text{¨},\text{¨}R)\times\rho\text{¨},\text{¨}C)\lceil\rho\text{¨},\text{¨}D \tag{4.13}$$

if R, C and D are nonsimple arrays. Thus,

$$K=1\uparrow\rho\supset Z$$

and

```
K←⍴2⊃Z.
```

The result *Z* represents a tableau with shape `∪⍴⍤∪⍤R C`.

"Appendix 2: Distribution Function Algorithms" on page 170 gives the assign distribution algorithm.

Examples:

Example set 1 - Simple arrays.

Example 1:

```
      R
1 2 3

      C
1 2 3 4

      D
10 11 12 13
14 15 16 17
18 19 20 21

      AD R C D
 1 1   10 11 12 13 14 15 16 17 18 19 20 21
 1 2
 1 3
 1 4
 2 1
 2 2
 2 3
 2 4
 3 1
 3 2
 3 3
 3 4

      TABLE AD R C D
┌──────────┐
│10 11 12 13│
│14 15 16 17│
│18 19 20 21│
└──────────┘
```

Example 2:

```
      R
1 2 3
4 5 6
```

```
      C
1  2  3  4
5  6  7  8
9 10 11 12

      D
10 11 12 13
14 15 16 17
18 19 20 21

22 23 24 25
26 27 28 29
30 31 32 33

34 35 36 37
38 39 40 41
42 43 44 45

46 47 48 49
50 51 52 53
54 55 56 57

58 59 60 61
62 63 64 65
66 67 68 69

70 71 72 73
74 75 76 77
78 79 80 81

      ⍴D
2 3 3 4

      TABLE AD R C D
```

10	11	12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54	55	56	57
58	59	60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79	80	81

Example set 2 - Nonsimple arrays.

Example 3:

```
      R1
1 2 3

      R2
4 5 6 7
```

```
       C1
1 2 3 4

       C2
5 6 7

       D1
10 11 12 13
14 15 16 17
18 19 20 21

       D2
30 31 32
33 34 35
36 37 38
39 40 41
```

```
      TABLE AD (R1 R2) (C1 C2) (D1 D2)
```

10	11	12	13			
14	15	16	17			
18	19	20	21			
				30	31	32
				33	34	35
				36	37	38
				39	40	41

Example 4:

```
       R
1 2 3
4 5 6
7 8 9

       C
1  2  3  4
5  6  7  8
9 10 11 12

       D
1 4 2 3
1 1 3 3
4 2 3 4
```

```
      TABLE AD (⊂[2]R) (⊂[2]C) (⊂D)
```

```
 1  4  2  3
 1  1  3  3
 4  2  3  4

                1  4  2  3
                1  1  3  3
                4  2  3  4

                               1  4  2  3
                               1  1  3  3
                               4  2  3  4
```

Example 5:

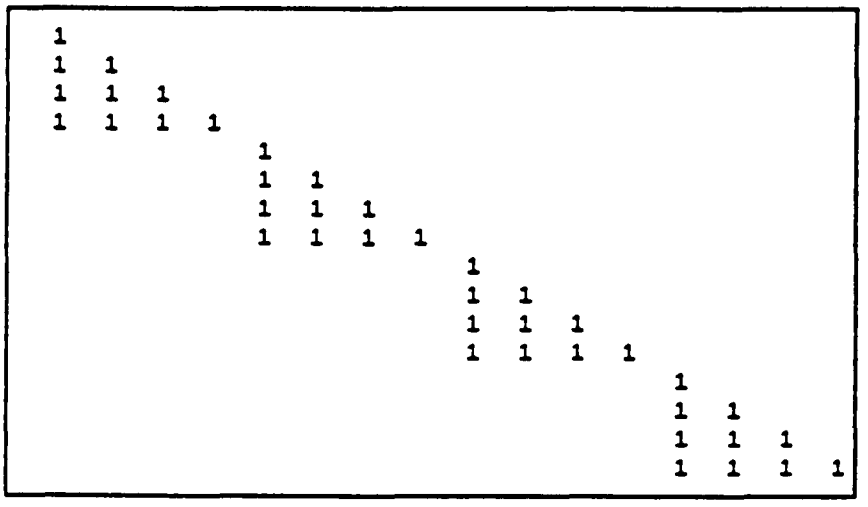
```
      TABLE AD (⊂[2]R) (⊂[2]C) ((⊂D)*1 2 3)
```

```
 1  4  2  3
 1  1  3  3
 4  2  3  4

               1 16  4  9
               1  1  9  9
              16  4  9 16

                              1 64  8 27
                              1  1 27 27
                             64  8 27 64
```

Example 6:

```
      R
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

      C
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

      D
1 0 0 0
1 1 0 0
1 1 1 0
1 1 1 1
```

TABLE AD (⊂[2]*R*) (⊂[2]*C*) (⊂*D*)



SCALAR NETWORK DISTRIBUTION

Scalar network distribution creates directed arcs indexed by column identifier arrays between corresponding elements of two node sets indexed by two row identifier arrays.

Syntax: *Z* ← *SN* (*C RF RT*)

The right argument is a list containing three items. The first item *C* is either i) a simple array of column identifiers, or ii) an array whose items are simple arrays of column identifiers. The second and third items *RF* and *RT* are either i) simple arrays of row identifiers, or ii) arrays whose items are simple arrays of row identifiers. The result *Z* is an array of problem elements, formed from the leaves of identifier arrays *C*, *RF* and *RT*, and the scalars 1 and ¯1, in table-list sparse representation.

Conformability Requirements:

C, *RF* and *RT* must have the same shape. If any is a singular array, it is replicated to have the same shape as the other array(s).

If *C*, *RF* and *RT* are simple arrays then

$$\rho C \leftrightarrow \rho RF \leftrightarrow \rho RT \qquad (4.14)$$

subject to replication of singular arrays.

If *C*, *RF* and *RT* are nonsimple arrays, then for all paths *P* defined on *C*, *RF* and *RT*,

$$\rho P \supset C \leftrightarrow \rho P \supset RF \leftrightarrow \rho P \supset RT \qquad (4.15)$$

subject to replication of singular items of *C*, *RF* and *RT*.

Definition:

If *C*, *RF* and *RT* are simple arrays, let *P* be a scalar path on *C*, *RF* and *RT*, subject to previous replication of singular arrays. *C* and *RF* address $\times/\rho,C$ problem elements from which directed arcs are incident. Problem elements are formed from the column identifier

$$P \supset C \ , \qquad (4.16)$$

the row identifier

$$P \supset RF \ , \qquad (4.17)$$

and the scalar value 1 for all *P* for which $P \supset C$ and $P \supset RF$ are defined. *C* and *RT* address $\times/\rho,C$ problem elements to which directed arcs are incident. Problem elements are formed from the column identifier

$$P \supset C \ , \qquad (4.18)$$

the row identifier

$$P \supset RT \ , \qquad (4.19)$$

and the scalar value $^{-}1$ for all *P* for which $P \supset C$ and $P \supset RT$ are defined.

If *C*, *RF* and *RT* are nonsimple identifier arrays, then the above definition is applied itemwise to the corresponding items of *C*, *RF* and *RT* after appropriate replication of singular items.

The result *Z* is an array of problem elements with two items. The first item of *Z* is a two-column table of tableau addresses, formed from the leaves of *RF*, *RT* and *C*. The second item of *Z* is a list of tableau coefficients formed from the scalars 1 and ¯1. *K* is the number of problem elements produced, where

$$K=2\times\supset\lceil/\rho\ddot{}\cup\ddot{}C\ RF\ RT \qquad (4.20)$$

Thus,

$$K=1\uparrow\rho\supset Z$$

and

$$K=\rho 2\supset Z.$$

The result *Z* represents a tableau with shape $(\rho\cup RF\ RT),\rho\cup C$.

"Appendix 2: Distribution Function Algorithms" on page 170 gives the scalar network distribution algorithm.

Examples:

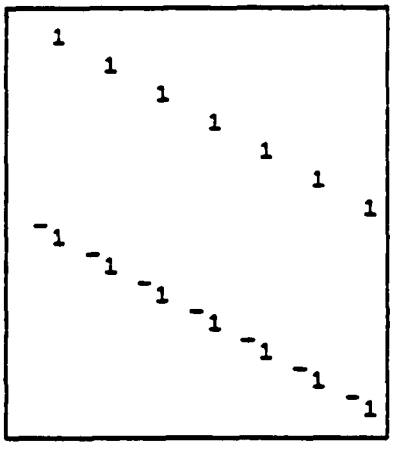
Example set 1 - Simple arrays.

Example 1:

```
      C
1 2 3 4 5 6 7

      RF
1 2 3 4 5 6 7

      RT
8 9 10 11 12 13 14
```

```
      SN C RF RT
 1  1    1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1
 2  2
 3  3
 4  4
 5  5
 6  6
 7  7
 8  1
 9  2
10  3
11  4
12  5
13  6
14  7
```

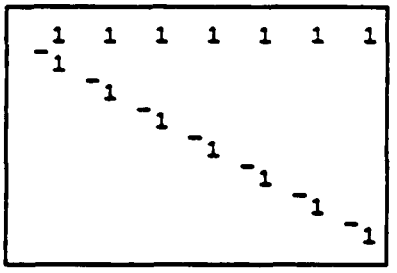
```
      TABLE SN C RF RT
```



Example 2:

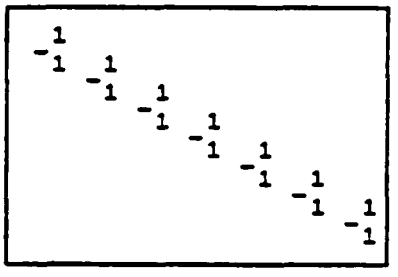
```
      C
1 2 3 4 5 6 7

      RF
1

      RT
2 3 4 5 6 7 8
```

```
      TABLE SN C RF RT
```



Example 3:

```
      C
1 2 3 4 5 6 7

      R
1 2 3 4 5 6 7 8

      SN C (¯1↓R) (1↓R)
 1 1    1 1 1 1 1 1 1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1 ¯1
 2 2
 3 3
 4 4
 5 5
 6 6
 7 7
 2 1
 3 2
 4 3
 5 4
 6 5
 7 6
 8 7

      TABLE SN C (¯1↓R) (1↓R)
```

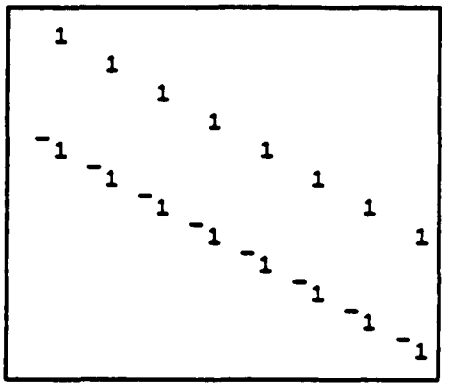


Example 4 - Nonsimple arrays:

```
      C
1 2 3 4
5 6 7 8
```

```
      R
1  2  3  4
5  6  7  8
9 10 11 12
```

```
      TABLE SN (⊂[2]C) (¯1↓⊂[2]R) (1↓⊂[2]R)
```



CARTESIAN NETWORK DISTRIBUTION

Cartesian network distribution creates directed arcs indexed by column identifier arrays between all possible pairs of elements of two node sets indexed by two row identifier arrays.

Syntax: *Z ← CN (C RF RT)*

The right argument is a list containing three items. The first item *C* is either i) a simple array of column identifiers, or ii) an array whose items are simple arrays of column identifiers. The second and third items *RF* and *RT* are either i) simple arrays of row identifiers, or ii) arrays whose items are simple arrays of row identifiers. The result *Z* is an array of problem elements, formed from the leaves of identifier arrays *C*, *RF* and *RT*, and the scalars 1 and ¯1, in table-list sparse representation.

Conformability Requirements:

If *C*, *RF* and *RT* are simple arrays, then

$$\rho C \leftrightarrow (\rho RF),\rho RT \tag{4.21}$$

If *C* is a simple scalar, it is replicated to the shape $(\rho RF),\rho RT$.

If *C*, *RF* and *RT* are nonsimple arrays, then *C*, *RF* and *RT* must have the same shape, subject to replication of singular arrays, and for all scalar paths *P* defined on *C*, *RF* and *RT*,

$$\rho P{\supset}C \leftrightarrow (\rho P{\supset}RF),\rho P{\supset}RT \tag{4.22}$$

subject to replication of singular items of *C*, *RF* and *RT*.

Definition:

If *C*, *RF* and *RT* are simple arrays, let *P* be a scalar path defined on *C*. For every *P* for which $P{\supset}C$ is defined, a problem element is formed from the column identifier

$$P{\supset}C \ , \tag{4.23}$$

the row identifier

$$(\subset(\rho\rho RF)\uparrow{\supset}P){\supset}RF \ , \tag{4.24}$$

and the scalar value 1; and a problem element is formed from the column identifier

$$P{\supset}C \ , \tag{4.25}$$

the row identifier

$$(\subset(-\rho\rho RT)\uparrow{\supset}P){\supset}RT \ , \tag{4.26}$$

and the scalar value $\bar{}1$.

If *C*, *RF* and *RT* are nonsimple arrays, then the above definition is applied itemwise to the corresponding items of *C*, *RF* and *RT* after appropriate replication of singular items.

The result *Z* is an array of problem elements with two items. The first item of *Z* is a two-column table of tableau addresses,

formed from the leaves of *RF*, *RT* and *C*. The second item of *Z* is a list of tableau coefficients formed from the scalars 1 and ¯1. *K* is the number of problem elements produced, where

```
K=((ρ,RF)×ρ,RT)⌈ρ,C                                    (4.27)
```

if *C*, *RF* and *RT* are simple arrays, or

```
K=+/((ρ¨,¨RF)×ρ¨,¨RT)⌈ρ¨,¨C                            (4.28)
```

if C, RF and RT are nonsimple arrays. Thus,

```
K=1↑ρ⊃Z
```

and

```
K=ρ2⊃Z.
```

The result *Z* represents a tableau with shape (ρ∪*RF RT*),ρ∪*C*.

"Appendix 2: Distribution Function Algorithms" on page 170 gives the cartesian network distribution algorithm.

<u>Example:</u>

```
      C
1  2  3  4
5  6  7  8
9 10 11 12

      RF
1 2 3

      RT
4 5 6 7

      TABLE CN C RF RT
┌────────────────────────────────────────┐
│ 1  1  1  1                             │
│             1  1  1  1                 │
│                         1  1  1  1     │
│¯1          ¯1          ¯1              │
│   ¯1          ¯1          ¯1           │
│      ¯1          ¯1          ¯1        │
│         ¯1          ¯1          ¯1     │
└────────────────────────────────────────┘
```

USING DISTRIBUTION FUNCTIONS

This Section describes how distribution functions are used in modeling and building LPO problems.

Generalized Reduction Distribution

When a real process is analyzed using LP, most people think of model components in terms of systems of linear equations and inequalities involving summation of array products. In traditional mathematical notation, these kinds of relationships are usually expressed using the Greek letter Sigma adorned with subscripts and superscripts, and array identifiers, both constants and variables, also adorned with qualifying coordinate identifiers. The generalized reduction distribution function is the array operation analog of such traditional mathematical notations. As such, it is the main component of most LP models expressed in terms of distribution functions. Any summation expression of array products in traditional mathematical notation can be expressed as a generalized reduction expression, either directly or by appropriate restructuring of its array arguments.

Scalar Distribution

The tableau representation of most LP problems contain multiple instances of identity matrices as subarrays. Scalar distribution produces structures that are generalized identity arrays. In its simplest form, scalar distribution produces an identity matrix, as for example:

```
      TABLE SD (⍳5) (⍳5) 1
```

```
1
  1
    1
      1
        1
```

Scalar distribution generalizes the identity matrix concept to arrays of higher rank.

Assign Distribution

Assign distribution plays two main roles in specifying LP models: i) the incorporation of a pre-structured array into an LP tableau, and ii) the replication of pre-structured arrays within a tableau, either directly or with a systematic array calculation. A simple example of the first case is:

```
      TABLE AD (⍳3) (⍳4) (3 4⍴10×⍳12)
```

```
 10  20  30  40
 50  60  70  80
 90 100 110 120
```

This easily generalizes to an example of the second case:

```
      TABLE AD (⊂[2]3 3⍴⍳9) (⊂[2]3 4⍴⍳12) (0 1 2+⊂3 4⍴10×⍳12)
```

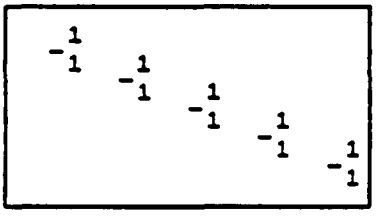
```
 10  20  30  40
 50  60  70  80
 90 100 110 120
                11  21  31  41
                51  61  71  81
                91 101 111 121
                                12  22  32  42
                                52  62  72  82
                                92 102 112 122
```

Assign distribution is especially useful for constructing components of LPO problems that defy structural definition in terms of other distribution functions.

Scalar Network Distribution

Scalar network distribution allows the construction of structures representing simple networks. Such structures are common in most models. For example, the construction of a relationship between activities in a time-period model for carrying over to successive time periods is:

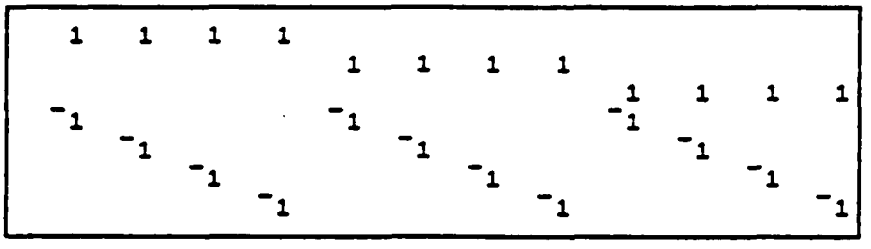
```
TABLE SN (⍳5) (¯1+⍳6) (1+⍳6)
```



Cartesian Network Distribution

Cartesian network distribution builds the structure representing a bipartite graph. This transportation problem-type substructure is common in many models:

```
TABLE CN (3 4⍴⍳12) (⍳3) (3+⍳4)
```



The next Section describes the implementation of distribution functions for describing LP models and building LPO problems.

IMPLEMENTATION: THE MODEL DESCRIPTION LANGUAGE (MDL)

The Model Description Language (MDL) is a high level programming language for describing LP models. The MDL interpreter executes MDL programs to produce LP optimization problems. MDL uses the concepts developed in "Distribution Functions" on page 88.

THE MDL ENVIRONMENT

MDL operates in an interactive programming mode under APL; all facilities normally available in APL are available in MDL. These include:

- The creation, analysis, and processing of data aggregates (arrays) interactively using primitive and defined APL functions, and operators.
- The easy creation of APL programs (defined functions) for various processing services and applications.
- A workspace where data arrays and user-defined functions reside in the same computing environment.
- File services for creation and access of out-of-workspace data aggregates.

- A simple interface between user and program for building interactive application systems.

MDL augments the APL environment with the following facilities:

- The MDL interpreter for processing MDL programs.
- A global data structure for the representation of LPO problems.
- A set of MDL utility functions for aiding the model verification and problem generation process.

The MDL interpreter processes MDL programs (models) that are present in the MDL environment as APL arrays. Models primarily reference two kinds of global arrays: problem data arrays and problem identifier arrays. The concept of these arrays here is identical to the concept developed in "Data" on page 70: Problem identifier data arrays are sets of indices for the rows and columns of an LP problem tableau. The structure of identifier data is related to structures in the underlying LP model of the process being optimized. Problem data arrays are structured aggregates of arbitrary real numbers that form the domain of LP tableau coefficients.

MDL PROBLEM REPRESENTATION

Linear programming optimization problems are represented in MDL by a set of global arrays that are created when the MDL environment is initialized. The arrays are augmented with information about the LPO problem by the MDL interpreter as it executes an MDL program. The initialization function *MDLINIT* is described in detail in "MDL Initialization" on page 131; the interpreter function *MDL* is described in "The MDL Interpreter" on page 144.

The following arrays are used to represent an LPO problem in MDL. (Assume that the problem being represented has *M* rows, including objective function, constraint, and bound rows, *N* columns, including activities and right-hand-sides, and *K* nonzero tableau entries.)

⍵IJ A matrix of integers with *K* rows and two columns. Each row of *⍵IJ* gives the row and column index, respectively, of a nonzero tableau coefficient.

⍵A A numeric vector of length *K* whose items are tableau coefficients. Row *L* of *⍵IJ* gives the row and column index in the LP tableau of *⍵A*[*L*].

⍵RT A vector of integers of length *M* giving the type of each row. Possible values are:

- 0 An objective function row.
- 1 A less-than-or-equal (≤) constraint row.
- 2 A greater-than-or-equal (≥) constraint row.
- 3 An equality (=) constraint row.
- 4 A lower bound vector.

5 An upper bound vector.

ZCT A vector of integers of length *N* giving the type of each column. Possible values are:

0 A right-hand-side (RHS) column.

1 A continuous LP activity.

2 A binary (zero-one) LP activity.

3 A general integer LP activity.

Note that *ZIJ* is constructed from problem identifier arrays and *ZA* is constructed from problem data arrays.

The transformation of this representation of an LPO problem in APL to the standard LP solution procedure input format is straightforward; see [Cro81].

MDL INITIALIZATION

The niladic function *MDLINIT* initializes the MDL environment; it must be invoked before an LPO problem can be built. Its syntax is:

MDLINIT

Executing *MDLINIT* results in the following actions:

- The LPO problem representation arrays are initialized to empty arrays.

  Note: Any problem represented in the current MDL environment is lost when *MDLINIT* is executed. "MDL Utility Functions" on page 148 describes facilities for saving and restoring LPO problems.

- Arrays are initialized for storing information about problem identifier arrays, which are defined by subsequent problem identifier definition statements (described below in "MDL Statements").
- The global array *HISTORY* is initialized to the empty matrix. The contents of *HISTORY* are saved in the global array *OLDHISTORY*. (See "The MDL Interpreter" below.)
- Other services for the MDL environment initialization.

Various ways of executing *MDLINIT* in the context of building LPO problems are described later in this Section.

MDL STATEMENTS

The Model Description Language has four types of statements for defining LP models and building LP optimization problems:

- Problem identifier definition statements
- Function statements
- APL execution statements
- Comment statements

This Section describes the syntax and use of these statements in MDL programs. Appendix 1 gives a BNF description of the Model Description Language.

Note: In syntax descriptions of MDL statements, optional parameters are denoted by enclosure in < >.

PROBLEM IDENTIFIER DEFINITION STATEMENTS

There are four types of problem identifier definition statements:

- Row identifier definition statements
- Column identifier definition statements
- Bound identifier definition statements
- Right-hand-side (RHS) identifier definition statements

These statements all create arrays of identifier data in the MDL environment for subsequent processing by MDL function statements.

Row identifier definition statements

The syntax of a row identifier definition statement is:

DEFROW: name <,shape <,attribute>> <; name ...>

where:

name — The name of the array of one or more row identifiers to be created in the MDL environment.

shape — Either i) a numeric constant scalar or vector giving the shape of the array to be created, ii) a variable whose value is a numeric vector or scalar, or iii) a parenthesized APL expression that, when evaluated in the MDL environment, yields a numeric scalar or vector. If the shape is omitted, the array will be a scalar whose only item is a single row identifier.

attribute — A character array giving the type of LP tableau row or rows being defined. Possible values are:

'N'	An objective function row.
'E' or '='	An equality row.
'L' or '≤'	A less-than-or-equal row.
'G' or '≥'	A greater-than-or-equal row.

The attribute can be either i) a character constant scalar, in which case all row identifiers being defined will have the indicated attribute, or ii) a vector constant or a parenthesized APL expression yielding a character result the same shape as the row identifier array being defined that indicates the type of each row identifier. If the attribute is not specified, all rows will be equality type (that is, *'E'*).

Examples

The statement

```
DEFROW: COSTS,2,'N'
```

defines a vector *COSTS* of length two that indexes a pair of objective function rows.

The statement

```
DEFROW: A,(⍴DATA),'L'; B,,'G'; C
```

defines three row identifier arrays: *A* is an array that indexes less-than-or-equal rows with the same shape as the array *DATA*, which must be defined in the MDL environment; *B* indexes a single greater-than-or-equal row; *C* indexes a single equality row.

The statement

```
DEFROW: RX,2,'GL'
```

defines a vector *RX* of length two; *RX*[1] indexes a greater-than-or-equal row; *RX*[2] indexes a less-than-or-equal row.

Column identifier definition statements

The syntax of a column identifier definition statement is:

DEFCOL: name <,shape <,attribute>> <; name ...>

where:

name — The name of the array of one or more column identifiers to be created in the MDL environment.

shape — Either i) a numeric constant scalar or vector giving the shape of the array to be created, ii) a variable whose value is a numeric vector or scalar, or iii) a parenthesized APL expression that, when evaluated in the MDL environment, yields a numeric scalar or vector. If the shape is omitted, the array will be a scalar whose only item is a single column identifier.

attribute — A character array giving the type of LP tableau column or columns being defined. Possible values are:

'*C*' A column for a continuous LP variable.

'*B*' A column for a binary (zero-one) variable.

'*I*' A column for a general integer variable.

The attribute can be either i) a character constant scalar, in which case all column identifiers being defined will have the indicated attribute, or ii) a vector constant or a parenthesized APL expression yielding a character result the same shape as the column identifier array being defined that indicates the type of each column identifier. If the attribute is not specified, all columns will be continuous type (that is, '*C*').

Examples

The statement

DEFCOL: *X*,5,'*B*'

defines a vector *X* of length five that contain indices of LP tableau columns for binary (zero-one) LP variables.

The statement

DEFCOL: *PRODUCTION*; *DISTRIBUTION*,(1↑⍴*WORK*),'*I*'; *SLACK*,3 4

defines three column identifier arrays: *PRODUCTION*, which indexes a scalar continuous variable; *DISTRIBUTION*, an array of column identifiers for general integer variables with shape the same as the first coordinate of *WORK*, which must be defined in the MDL environment; and *SLACK*, which is a matrix of column identifiers for continuous variables.

Bound identifier definition statements

The syntax of a bound identifier definition statement is:

DEFBND: name <,shape <,attribute>> <; name ...>

where:

name — The name of the array of one or more bound identifiers to be created in the MDL environment.

shape — Either i) a numeric constant scalar or vector giving the shape of the array to be created, ii) a variable whose value is a numeric vector or scalar, or iii) a parenthesized APL expression that, when evaluated in the MDL environment, yields a numeric scalar or vector. If the shape is omitted, the array will be a scalar whose only item is a single bound identifier.

attribute A character array giving the type of LP tableau bound or bounds being defined. Possible values are:

'*U*' An upper bound.

'*L*' A lower bound.

The attribute can be either i) a character constant scalar, in which case all bound identifiers being defined will have the indicated attribute, or ii) a vector constant or a parenthesized APL expression yielding a character result the same shape as the bound identifier array being defined that indicates the type of each bound identifier. If the attribute is not specified, all bounds will be upper type (that is, '*U*').

Example

The statement

DEFBND: *UPPER*, 3 4; *LOWER*,(⍴*BOAT*),'*L*'

defines two bound identifier arrays: *UPPER* is a matrix that contains indices of upper bound rows; *LOWER* is an array containing indices of lower bounds the same shape as *BOAT*, which must be defined in the MDL environment.

RHS identifier definition statements

The syntax of a RHS identifier definition statement is:

DEFRHS: name <,shape> <; name ...>

where:

name The name of the array of one or more RHS identifiers to be created in the MDL environment.

shape Either i) a numeric constant scalar or vector giving the shape of the array to be created, ii) a variable whose

value is a numeric vector or scalar, or iii) a parenthesized APL expression that, when evaluated in the MDL environment, yields a numeric scalar or vector. If the shape is omitted, the array will be a scalar whose only item is a single RHS identifier.

Example

The statement

DEFRHS: *B*1; *B*2,2

defines two RHS identifier arrays: *B*1, which is a scalar, and *B*2, which is a vector of length two.

MDL FUNCTION STATEMENTS

Function statements process identifier arrays and problem data arrays to produce an LP optimization problem. All arrays referenced in function statements must be defined in the MDL environment: identifier arrays are created with problem identifier definition statements, problem data arrays are created explicitly or implicitly by the user.

Function statements produce problem elements of a LPO problem either by row or by column, using the functions developed in "Distribution Functions" on page 88. MDL function statements produce implicit results by updating the sparse tableau representation arrays of the LPO problem in the MDL environment.

Function statements are either simple or compound. We will first describe simple function statements and then show how they can be combined to form compound statements.

Simple MDL function statements

There are two general forms of MDL function statements. The first form is used to carry out generalized reduction, scalar, and assign distribution; it has the following syntax:

keyword: id-array1; <data-array> function id-array2

where:

keyword — Either *ROW* or *COL* (*R* and *C* are valid abbreviations).

id-array1 — Either i) a variable name that is a previously defined problem identifier array, or ii) a parenthesized APL expression that performs selection from a previously defined problem identifier array. If keyword is ROW then id-array1 must be a row or bound identifier array; if keyword is COL then id-array1 must be a column or RHS identifier array.

data-array — Either i) a constant numeric scalar or vector, ii) a variable whose value is a numeric array, or iii) a parenthesized APL expression that yields a numeric array. If data-array is omitted, the value scalar 1 is used for the data value in the statement.

function — One of the following:

- Δ*GR*[*A*] Generalized reduction distribution with axis specification.
- Δ*GR* Generalized reduction distribution along the first coordinate (the same as Δ*GR*[1]).
- Δ*SD* Scalar distribution.
- Δ*AD* Assign distribution.

id-array2 Either i) a variable name that is a previously defined problem identifier array, or ii) a parenthesized APL expression that performs selection from a previously defined problem identifier array. If keyword is ROW then id-array2 must be a column or RHS identifier array; if keyword is COL then id-array2 must be a row or bound identifier array.

The second general form of an MDL function statement is used to carry out scalar network and cartesian network distribution; it has the following form:

keyword: id-array1 ; id-array2 function id-array3

where:

keyword Either *ROW* or *COL* (*R* and *C* are valid abbreviations).

id-array1 Either i) a variable name that is a previously defined problem identifier array, or ii) a parenthesized APL expression that performs selection from a previously defined problem identifier array. If keyword is ROW then id-array1 must be a row or bound identifier array; if keyword is COL then id-array1 must be a column or RHS identifier array.

id-array2 Each of these is either i) a variable name that is a
id-array3 previously defined problem identifier array, or ii) a parenthesized APL expression that performs selection from a previously defined problem identifier array. If keyword is ROW then id-array2 and id-array3 must be

column or RHS identifier arrays; if keyword is COL then id-array2 and id-array3 must be row or bound identifier arrays.

function One of the following:

ΔSN Scalar network distribution.

ΔCN Cartesian network distribution.

The following table shows the relationship between the functions defined in "Distribution Functions" on page 88 and the corresponding MDL function statements.

Distribution Function	MDL Function Statement
Z←A GR (R C D)	*ROW: R; D ΔGR[A] C*
Z←1 GR (R C D)	*ROW: R; D ΔGR C*
Z←SD (R C D)	*ROW: R; D ΔSD C*
Z←AD (R C D)	*ROW: R; D ΔAD C*
Z←SN (R CF CT)	*ROW: R; CF ΔSN CT*
Z←CN (R CF CT)	*ROW: R; CF ΔCN CT*
Z←A GR (C R D)	*COL: C; D ΔGR[A] R*
Z←1 GR (C R D)	*COL: C; D ΔGR R*
Z←SD (C R D)	*COL: C; D ΔSD R*
Z←AD (C R D)	*COL: C; D ΔAD R*
Z←SN (C RF RT)	*COL: C; RF ΔSN RT*
Z←CN (C RF RT)	*COL: C; RF ΔCN RT*

where: *R, RF, RT* - Row identifier arrays
C, CF, CT - Column identifier arrays
D - Problem data array
A - Axis index list
Z - Resulting problem elements

The conformability requirement for distribution functions in MDL function statement form are the same as for the corresponding functions described in "Distribution Functions" on page 88.

Compound MDL function statements

Consider the two simple MDL function statements:

```
ROW: A; D1 ΔAD C1
ROW: A; D2 ΔSD C2
```

where *A* is a row identifier array, *D1* and *D2* are problem data arrays, and *C1* and *C2* are column identifier arrays. These two simple statements can be combined to form the compound MDL function statement

```
ROW: A; D1 ΔAD C1; D2 ΔSD C2
```

In general, several simple MDL function statements defining problem elements by row (column) can be combined to form a compound MDL function statement if the row (column) identifier array argument is the same in each simple statement.

Example

Given the following identifier data definitions:

```
DEFROW: A,4,'L'; B,3
DEFCOL: X,3 4; Y,3
```

and the problem data array *D*:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

the MDL function statements

```
ROW: A; D ΔGR X; (⍉D) ΔAD Y
ROW: B; D ΔGR[2] X; ΔSD Y
```

would produce the following tableau, show pictorially as produced by the utility function *SHOW* (see "MDL Utility Functions" below):

	X								Y						
A	1				5				9				1	5	9	≤
		2				6				10			2	6	10	≤
			3				7				11		3	7	11	≤
				4				8				12	4	8	12	≤
B	1	2	3	4									1			=
					5	6	7	8						1		=
									9	10	11	12			1	=

APL EXECUTION STATEMENTS

APL execution statements are used for the execution of in-line APL statements in an MDL program. The syntax is:

```
⍝ APL-statement
```

where APL-statement is any valid APL statement in the context of the current MDL environment.

Examples

The following MDL program fragments show typical applications of APL execution statements.

Creation of a temporary array to save expression executions:

```
 ...
⍝ ITEMP←1↑⍴DATA
DEFROW: A,ITEMP; B,ITEMP; C,ITEMP
 ...
```

User prompting function *USERPROMPT* in an interactive application to obtain problem data:

```
 ...
⍝ DATA←USERPROMPT
ROW: ROW1; DATA ΔAD C1; DATA ΔAD C2
 ...
```

COMMENT STATEMENTS

A comment statement, denoted by prefixing with the symbol ⍝, can occur any place in an MDL program, either as a statement by itself or with another statement. In any MDL statement, everything to the right of the symbol ⍝ is considered a comment.

Examples

```
⍝ THIS IS A COMMENT ...
DEFROW: A;B;C   ⍝ ... AND SO IS THIS.
```

THE MDL INTERPRETER

The APL function *MDL* processes MDL programs, consisting of MDL statements, to produce an LPO problem.

Syntax: *MDL PROG*

The argument *PROG* is an array of MDL statements in one of three forms:

1. A simple character vector consisting of a single MDL statement.
2. A simple character matrix, each row of which is an MDL statement.
3. A scalar or vector, each item of which is a simple character vector consisting of a single MDL statement.

Definition

The results of executing *MDL* are implicit and depend on the type of MDL statement or statements processed. The result of executing a multi-statement program (form 2 or 3 above) is

equivalent to executing each statement individually in sequence. The action and effect on the MDL environment of executing each type of MDL statement is summarized below.

Problem identifier definition statements:

An integer identifier array with the indicated name and shape is created in the MDL environment. Any array with the same name is replaced. The contents of the created array are tableau row indices (for DEFROW and DEFBND statements) or column indices (for DEFCOL and DEFRHS statements).

The array *ZRT* (for rows) or *ZCT* (for columns) is updated with the type of row or column being defined.

Function statements:

The LPO problem arrays *ZIJ* and *ZA* are updated with tableau information according to the action of the function statement.

APL execution statements:

The APL statement is executed in the context of the current MDL environment.

Comment statements:

No operation; no change of the MDL environment.

Recursive MDL execution

The MDL function can be invoked from within an MDL program using the APL execution statement. This, in effect, allows MDL subprogram calls.

MDL history

Each MDL statement that is executed by *MDL* is appended as a new row to a global simple character matrix named *HISTORY*. This gives a convenient record of all MDL statements executed to build an LPO problem. *HISTORY* is initialized to an empty array by *MDLINIT*.

Notes

1. Execution of *MDLINIT* followed by execution of a program by *MDL* starts building a new LPO problem in the MDL environment. Execution of two successive programs by *MDL* without an intervening *MDLINIT* execution is a continuation; the actions of both programs are reflected in the resulting LPO problem. The effect is that any MDL program can be executed one statement at a time by individual invocations of *MDL*.
2. The *MDLINIT* function can be invoked from within an MDL program using the APL execution statement. Thus, an MDL program can contain its own initialization of the MDL environment.
3. Conditional processing can be archived by making selective and conditional calls to *MDL* from within an APL program.

Examples

The following examples show two formulations of the single commodity transportation problem described in "Constructing Problems: An Example" on page 74. In addition, this problem has upper bounds on the transportation activity levels.

The first uses generalized reduction distribution:

```
⍝ TRANSPORTATION PROBLEM FORMULATION USING ΔGR
⍝ GLOBAL PROBLEM DATA ARRAYS:
⍝   COST - MATRIX OF TRANSPORTATION COSTS.
⍝   AVL  - VECTOR OF SOURCE AVAILABILITIES  ⍴AVL ←→ 1↑⍴COST
⍝   REQ  - VECTOR OF DEST. REQUIREMENTS  ⍴REQ ←→ 1↓⍴COST
⍝   LIM  - UPPER BOUND ON TRANSPORT. ACTIVITY  ⍴LIM ←→ ⍴COST

∇ MDLINIT    ⍝ MDL INITIALIZATION

⍝ ROW AND BOUND DEFINITIONS:
    DEFROW: OBJ,,'N'; A,(⍴AVL),'L'; R,(⍴REQ),'G'
    DEFBND: UPPER
⍝ COLUMN AND RHS DEFINITIONS:
    DEFCOL: X,(⍴COST)
    DEFRHS: B
⍝ CONSTRAINTS:
ROW: OBJ; COST ΔGR[1 2] X       ⍝ OBJECTIVE FUNCTION
ROW: A; ΔGR[2] X; AVL ΔAD B     ⍝ SOURCE CONSTRAINTS
ROW: R; ΔGR X; REQ ΔAD B        ⍝ DESTINATION CONSTRAINTS
ROW: UPPER; LIM ΔSD X           ⍝ BOUNDS
```

The second uses cartesian network distribution:

```
⍝ TRANSPORTATION PROBLEM FORMULATION USING ΔCN
⍝ GLOBAL PROBLEM DATA ARRAYS:
⍝   COST - MATRIX OF TRANSPORTATION COSTS.
⍝   AVL  - VECTOR OF SOURCE AVAILABILITIES  ⍴AVL ←→ 1↑⍴COST
⍝   REQ  - VECTOR OF DEST. REQUIREMENTS  ⍴REQ ←→ 1↓⍴COST
⍝   LIM  - UPPER BOUND ON TRANSPORT. ACTIVITY  ⍴LIM ←→ ⍴COST

∇ MDLINIT    ⍝ MDL INITIALIZATION

⍝ ROW AND BOUND DEFINITIONS:
    DEFROW: OBJ,,'N'; A,(⍴AVL),'L'; R,(⍴REQ),'L'
    DEFBND: UPPER
⍝ COLUMN AND RHS DEFINITIONS:
    DEFCOL: X,(⍴COST)
    DEFRHS: B
⍝ CONSTRAINTS:
COL: X; COST ΔSD OBJ; A ΔCN R; LIM ΔSD UPPER   ⍝ BY ACTIVITY
COL: B; (AVL,-REQ) ΔSD (A,R)                   ⍝ RHS
```

MDL UTILITY FUNCTIONS

This Section describes a set of utility functions in the MDL environment that aid the modeling process. The examples used to illustrate these functions will use following arrays:

```
      TRANS
∩ TRANSPORTATION PROBLEM FORMULATION USING ΔCN
∆MDLINIT
DEFROW: OBJ,,'N'; A,(⍴AVL),'L'; R,(⍴REQ),'L'
DEFBND: UPPER
DEFCOL: X,(⍴COST)
DEFRHS: B
COL: X; COST ΔSD OBJ; A ΔCN R; LIM ΔSD UPPER
COL: B; (AVL,-REQ) ΔSD (A,R)

      ⍴TRANS
8 46

      AVL
7 3 6

      REQ
4 3 7 2

      COST
 2 12  7  8
 4  1 11 11
15  6  8 13

      LIM
6 4 3 3
2 7 4 4
5 2 3 5
```

Function *IDS*

The function *IDS* lists the problem identifier arrays currently defined in the MDL environment.

Syntax: *Z←IDS*

The result Z is a simple character matrix that lists the row, column, RHS, and bound identifier arrays currently defined in the MDL environment, along with their shape and attributes.

Example

```
      MDL TRANS
      IDS
ROWS:
OBJ          N
A      3     L
R      4     L
COLS:
X      3 4   C
RHSS:
B
BNDS:
UPPER        U
```

Function *SHOW*

The function *SHOW* produces a pictorial form of the currently defined LPO problem.

Syntax: *Z←SHOW*

The result *Z* is a simple character matrix showing the current LPO problem in pictorial format.

Example

```
      MDL TRANS
      SHOW
```

	X													*B*
OBJ	2	12	7	8	4	1	11	11	15	6	8	13		
A	1	1	1	1									≤	7
					1	1	1	1					≤	3
									1	1	1	1	≤	6
R	-1				-1				-1				≤	-4
		-1				-1				-1			≤	-3
			-1				-1				-1		≤	-7
				-1				-1				-1	≤	-2
UPPER	6	4	3	3	2	7	4	4	5	2	3	5	↑	

Function *PATTERN*

The function *PATTERN* is like *SHOW* except that only the sign (+ or -) of the tableau entries is shown.

Syntax: *Z←PATTERN*

The result *Z* is a simple character matrix showing the pattern of nonzero coefficients in the current LPO problem.

Example

```
      MDL TRANS
      PATTERN
┌──────────────┐   ┌─┐
│++++++++++++  │   │ │
│++++          │ ≤ │+│
│    ++++      │ ≤ │+│
│        ++++  │ ≤ │+│
│-   -   -     │ ≤ │-│
│ -   -   -    │ ≤ │-│
│  -   -   -   │ ≤ │-│
│   -   -   -  │ ≤ │-│
│++++++++++++  │   │ │
└──────────────┘   └─┘
```

Function *PROBLEM*

The function *PROBLEM* gives information about the current LPO problem tableau.

Syntax: *Z←PROBLEM*

The result *Z* is a simple character matrix giving the number and type of tableau row and column identifier, and information about the tableau coefficients.

Example

```
      MDL TRANS
      PROBLEM
IDS:
ROWS -
  N              1
  L              7
  G              0
  E              0
  TOTAL          8
COLUMNS -
  CONTINUOUS    12
  BINARY         0
  INTEGER        0
  TOTAL         12
RHS -            1
BOUNDS -
  U              1
  L              0
  TOTAL          1
MATRIX:
COEFFICIENTS    55
UNIQUE          17
DENSITY(%)      72.37
```

Function *CPOOL*

The function *CPOOL* gives information about the current LPO problem tableau coefficients.

Syntax: *Z←CPOOL*

The result *Z* is a simple character matrix that lists the unique tableau elements, their count, and gives the total number of nonzero elements, the number of unique elements, and the density of the tableau.

Example

```
      MDL TRANS
      CPOOL
 1.  _1 13
 2.  -1 12
 3.   3  4
 4.   4  4
 5.   2  3
 6.   6  3
 7.   7  3
 8.   5  2
 9.   8  2
10.  11  2
11.  -7  1
12.  -4  1
13.  -3  1
14.  -2  1
15.  12  1
16.  13  1
17.  15  1

TOTAL           55
UNIQUE          17
DENSITY(%)      72.37
```

Function *MDLSAVE*

The function *MDLSAVE* is used to save the LPO problem currently defined in the MDL environment.

Syntax: ***MDLSAVE NAME***

The argument *NAME* is a character scalar or vector. The currently defined LPO problem is saved.

Example

```
MDLSAVE 'MODEL1'
```

The LPO problem currently defined in the MDL environment is saved with name *MODEL*1.

Function *MDLRESTORE*

The function *MDLRESTORE* restores an LPO problem previously saved with the *MDLSAVE* function.

Syntax: *MDLRESTORE NAME*

The argument *NAME* is a character scalar or vector. The LPO problem previously saved by *MDLSAVE* under the name *NAME* is restored as the currently defined LPO problem in the MDL environment. A problem that is currently defined before execution of *MDLRESTORE* is purged from the MDL environment.

Example

```
MDLRESTORE 'MODEL1'
```

The LPO problem named *MODEL1* is restored in the current MDL environment.

Function *CORDER*

The function *CORDER* is used to sort the currently defined LPO problem into column order.

Syntax: *CORDER*

The currently defined arrays *ZIJ* and *ZA* are rearranged so that they list the coefficients of the LP tableau in column order.

Example

```
      ZIJ
3  4
3  3
2  1
1  3
2  4
3  2
2  2
3  1
1  2
1  4
1  1
2  3

      ZA
111  110  104  102  107  109  105  108  101  103  100  106

      CORDER

      ZIJ
1  1
2  1
3  1
1  2
2  2
3  2
1  3
2  3
3  3
1  4
2  4
3  4

      ZA
100  104  108  101  105  109  102  108  110  103  107  111
```

Function *RORDER*

The function *RORDER* is used to sort the currently defined LPO problem into row order.

Syntax: ***RORDER***

The currently defined arrays *ZIJ* and *ZA* are rearranged so that they list the coefficients of the LP tableau in row order.

Example

```
      ZIJ
3 4
3 3
2 1
1 3
2 4
3 2
2 2
3 1
1 2
1 4
1 1
2 3

      ZA
111 110 104 102 107 109 105 108 101 103 100 106

      RORDER

      ZIJ
1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
3 1
3 2
3 3
3 4

      ZA
100 101 102 103 104 105 106 107 108 109 110 111
```

MDL APPLICATION EXAMPLES

This Section demonstrates use of the Model Description Language on real linear programming applications.

PRODUCTION PLANNING

A production process is carried out over two time periods (for example, first six months of the year, last six months) in two production modes (regular and overtime), producing three types of products, on three types of machines. In the following discussion, time periods will be indexed by I (that is, I such that $I \in 1..2$), production modes will be indexed by J, product types by K, and machine types by L.

The following table gives the relevant problem data for the production process:

Name	Shape	Typical item	Description
T	2 2 3 3	*T*[*I*;*J*;*K*;*L*]	Time in hours required to produce one unit of product *K* on machine *L*, in time period *I*, using production mode *J*.
A	2 2 3	*A*[*I*;*J*;*L*]	Machine availability in hours for machine *L* in period *I* and in production mode *J*.
P	2 3	*P*[*I*;*K*]	Selling price for product type *K* in time period *I*.
D	2 3	*D*[*I*;*K*]	Demand for product type *K* in time period *I*.
S	3	*S*[*K*]	Storage cost of one unit of product type *K* for one time period.
H	3	*H*[*K*]	Storage capacity for product type *K* for one time period.
R	3	*R*[*K*]	Resale value of one unit of product type *K* stored for one time period.
C	2 2 3 3	*C*[*I*;*J*;*K*;*L*]	Production cost for one unit of product *K* on machine *L*, in time period *I*, using production mode *J*.

The activity levels to be determined by LP are given by the following table:

Name	Shape	Typical item	Description
X	2 2 3 3	*X*[*I*;*J*;*K*;*L*]	The level of production for product type *K* on machine *L*, in time period *I*, using production mode *J*.
Y	2 3	*Y*[*I*;*K*]	The quantity of product type *K* stored in time period *I*.
Z	2 3	*Z*[*I*;*K*]	The quantity of product type *K* sold in time period *I*.

The formulation of the LP problem is in terms of the successive plus-reduction function *SPR* defined in "Inner Product and Linear Programming" on page 52. The production planning model is formulated as follows:

1. The objective function is to

   Maximize *NET_PROFIT* + *RESALE* - *STORAGE*

   where:

   NET_PROFIT ←→ 1 2 3 4 *SPR X* × *P*-[1 3]*C*

   Production level (*X*) times the difference between selling price (*P*) and production cost (*C*), summed over all coordinates. The quantity *P*-[1 3]*C* is *P*[*I*;*J*;*K*;*L*]-*P*[*I*;*K*] for all *I*, *J*, *K*, and *L*.

   RESALE ←→ 1 *SPR Y*[2;] × *R*-*P*[2;]

   Storage level for the second time period (*Y*[2;]) times the difference between resale value (*R*) and selling price in the second time period (*P*[2;]), summed over all product types.

   STORAGE ←→ 1 *SPR Y*[1;] × *S*

   Storage level for the first time period (*Y*[1;]) times storage costs (*S*), summed over all product types.

2. Machine availability constraint:

   A ≥ 3 *SPR X* × *T*

   Machine availability (*A*) is not less than the product level (*X*) times the production time, summed over all machine types.

3. Stock balance constraint (in both time periods):

   (2 4 *SPR X*) = *Y* + *Z*

   The quantity 2 4 *SPR X* is the production level (*X*) summed over production mode and machine type; this gives the amount of

product produced per time period. This quantity must equal the sum of the level of product stored per time period (*Y*) plus the level of product sold per time period (*Z*).

4. Minimum demand constraint:

$$Z \geq D$$

The amount of products sold per time period (*Z*) must meet the demand (*D*).

5. Upper bound restriction on storage:

$$Y \leq[2] H$$

The product storage level (*Y*) must not exceed to storage capacity (*H*) in each time period.

A model for this problem is given by the following MDL program:

```
⍝ PRODUCTION PLANNING MODEL
⍝ MDLINIT
⍝ COORDINATE EXTENTS:
⍝     I←1⌷⍴T                      ⍝ number of time periods
⍝     J←2⌷⍴T                      ⍝ number of production modes
⍝     K←3⌷⍴T                      ⍝ number of product types
⍝     L←4⌷⍴T                      ⍝ number of machines
⍝ IDENTIFIER DEFINITIONS:
      DEFCOL: X,(I,J,K,L)         ⍝ production activity
      DEFCOL: Y,(I,K)             ⍝ storage activity
      DEFCOL: Z,(I,K)             ⍝ sales activity
      DEFROW: OBJ,,'N'            ⍝ objective function
      DEFROW: MA,(I,J,L),'L'      ⍝ machine avail. constraints
      DEFROW: SB,(I,K),'E'        ⍝ stock balance constraints
      DEFROW: DMD,(I,K),'G'       ⍝ min. demand constraint
      DEFBND: STOR                ⍝ storage bound
      DEFRHS: B
⍝ FUNCTIONS:
ROW: OBJ;(P-[1 3]C) ΔGR[⍳4] X     ⍝ net profit
ROW: OBJ; (R-P[2;]) ΔGR (Y[2;])   ⍝ resale
ROW: OBJ; (-S) ΔGR (Y[1;])        ⍝ storage
ROW: MA ; T ΔGR[3] X; A ΔSD B     ⍝ mach. avail.
ROW: SB ; ΔGR[2 4] X; ¯1 ΔSD Y; ¯1 ΔSD Z ⍝ stock balance
ROW: DMD; ΔSD Z; D ΔSD B          ⍝ min. demand
ROW: STOR; (⊂H) ΔSD (⊂[2]Y)       ⍝ storage bound
```

VLSI DESIGN: GRAPH PARTITIONING

The following problem arises in very large scale integration (VLSI) circuit design:

We are given a Boolean matrix *A*, each row and column of which contains at least one 1. We want to color the elements of *A* with value 1 two colors: red and blue. If *A* is so colored, a row of *A* that contains all red 1's is a red row. A row of *A* that contains all blue 1's is a blue row. And a row of *A* that contains both red and blue 1's is a bicolored row. Similar definitions on the columns of *A* give red columns, blue columns, and bicolored columns.

The objective is to find a coloring of the 1's of *A* that will

Minimize maximum { no. of bicolored rows, no. of bicolored cols }

A pure zero-one LP formulation of this problem has the following activities:

Name	Shape	Typical item	Description
XB	1↑⍴*A*	*XB*[*I*]	If row *I* contains at least one blue 1, then 1; otherwise 0.
XR	1↑⍴*A*	*XR*[*I*]	If row *I* contains at least one red 1, then 1; otherwise 0.
YB	1↑⍴*A*	*YB*[*J*]	If column *J* contains at least one blue 1, then 1; otherwise 0.
YR	1↑⍴*A*	*YR*[*J*]	If column *J* contains at least one red 1, then 1; otherwise 0.
Z	⍴*A*	*Z*[*I*;*J*]	1 if *A*[*I*;*J*] colored blue, 0 if *A*[*I*;*J*] colored red, for all *A*[*I*;*J*]≠0.
C			The minimum of the maximum of the number of bicolored rows and bicolored columns.

The problem has the following constraints:

1. Logical implications -

```
Z[I;J] ≤ XB[I]
```

Ensures that if A[I;J] is colored blue, then row I contains blue.

```
Z[I;J] ≤ YB[J]
```

Ensures that if A[I;J] is colored blue, then column J contains blue.

```
(1-Z[I;J]) ≤ XR[I]
```

Ensures that if A[I;J] is colored red, then row I contains red.

```
(1-Z[I;J]) ≤ YR[J]
```

Ensures that if A[I;J] is colored red, then column J contains red.

2. Color smoothing -

```
B1 ≥ +/XB
B1 ≥ +/XR
B2 ≥ +/YB
B2 ≥ +/YR
```

where

```
B1 = 2*K such that (2*K) < (1↑⍴A) ≤ 2*K+1
B2 = 2*L such that (2*L) < (1↑⍴A) ≤ 2*L+1
```

These constraints insure an even allocation of colors.

3. Summation constraints -

```
(+/XB,XR) ≤ C + 1↑⍴A
(+/YB,YR) ≤ C + 1↑⍴A
```

The sum of row indicator variables X and the sum of column indicator variables Y must not exceed the number of rows and columns, respectively.

4. Objective function -

Minimize *C*

The following MDL program models this problem, using only the scalar distribution function:

```
⍝ MDLINIT
⍝  DEFINE USEFUL MEASUREMENTS:
⍝ NZ←+/,A                ⍝ NUMBER OF NONZEROS IN A
⍝ M←1↑⍴A                 ⍝ NUMBER OF ROWS OF A
⍝ N←1↓⍴A                 ⍝ NUMBER OF COLS OF A
⍝ RT←(,A)/,⍉(⍳⍴A)∘.=⍳M   ⍝ VECTOR OF ROW INDICES OF NONZEROS IN A
⍝ CT←(,A)/,(⍴A)∘.=⍳N     ⍝ VECTOR OF COL INDICES OF NONZEROS IN A
⍝ B1←2*⌊2⍟M              ⍝ NEXT POWER OF 2 BELOW M
⍝ B2←2*⌊2⍟N              ⍝ NEXT POWER OF 2 BELOW N

DEFCOL: XB,M,'B'; XR,M,'B'; YB,N,'B'; YR,N,'B'
DEFCOL: Z,NZ,'B'; C

DEFROW: L1BLUE,NZ,'L'; L1RED,NZ,'L'
DEFROW: L2BLUE,NZ,'G'; L2RED,NZ,'G'
DEFROW: S1,,'L'; S2,,'L'; S3,,'L'; S4,,'L'
DEFROW: ROWSUM,,'L'; COLSUM,,'L'
DEFROW: OBJECTIVE,,'N'

DEFRHS: RHS

⍝ LOGICAL IMPLICATIONS...
ROW: L1BLUE; ¯1 ΔSD (XB[RT]); ΔSD Z
ROW: L1RED;  ¯1 ΔSD (XR[RT]); ΔSD Z
ROW: L2BLUE;    ΔSD (YB[CT]); ΔSD Z; ΔSD RHS
ROW: L2RED;     ΔSD (YR[CT]); ΔSD Z; ΔSD RHS

⍝ SMOOTHING...
ROW: S1; ΔSD XB; B1 ΔSD RHS
ROW: S2; ΔSD XR; B1 ΔSD RHS
ROW: S3; ΔSD YB; B2 ΔSD RHS
ROW: S4; ΔSD YR; B2 ΔSD RHS

⍝ SUMS...
ROW: ROWSUM; ΔSD (XB,XR); ¯1 ΔSD C; M ΔSD RHS
ROW: COLSUM; ΔSD (YB,YR); ¯1 ΔSD C; N ΔSD RHS

⍝ OBJECTIVE...
ROW: OBJECTIVE; ΔSD C
```

THE INSTALLATION SCHEDULING PROBLEM

The installation scheduling problem is often a difficult aspect of capital investment programs. Simply stated, it involves finding an orderly schedule for installing many systems of different sizes and types over time so as to optimize some measure (for example, initial capital investment), subject to various resource and time constraints. Examples of this problem are scheduling the installation of point-of-sale terminals in supermarkets and retail stores, and teller terminals in banks.

For a more detailed description of the installation scheduling problem and its formulation as a mixed integer LP problem, see [Che78].

In general, the installation scheduling problem arises when a decision must be made about when to perform all or part of a set of actions (typically, money investments) over a given time interval. The interval is divided into several time periods; one time period is the approximate time required to perform an action. The problem has the following characteristics:

- Each action has associated costs and benefits determined by functions. The cost function specifies the amount of capital required to perform the action; this cost is dependent on the time period in which the action is performed. At a given time period, for those actions that were carried out in previous time periods, the benefit function determines the amount of capital that is made currently available. In general, benefit functions will generate nonnegative returns, although this is

not a restriction. Both costs and benefits are estimates that take into account such factors as possible future inflation, investment tax credits, tax rates, and depreciation.

- The problem has an associated initial capital investment that is used to start performing actions. At any time period, capital is expended by performing actions, the amount of capital used being determined by the appropriate cost function. At the same time, capital benefits are derived in two ways: i) any capital not expended at the previous time period is carried forward, and ii) capital benefits arise from previously performed actions, as determined by the appropriate benefit functions.

The problem data for the mixed integer LP formulation of the installation scheduling problem is summarized by the following table.

Name	Shape	Typical item	Description
N			The number of types of facilities to installed.
T			The number of time periods allotted for installing all facilities.
M	*N*	*M[I]*	The number of facilities of type *I* to be installed.
E			The amount of the initial capital investment.
C	*N,T*	*C[I;J]*	The cost of installing a facility of type *I* in time period *J*.
B	*N,T-1* 1	*B[I;J;K]*	The capital benefit available at time period *J* for a facility of type *I* that was installed at time period *J-K*.

The activities to be determined are summarized in the following table (as before, *N* is the number of facility types and *T* is the number of time periods).

Name	Shape	Typical item	Description
X	*N*,*T*	*X*[*I*;*J*]	The number of facilities of type I installed in time period *J*. Fractional parts of *X* values indicate that installations are started during one time period and finished during the next period; see [Che78].
D	*N*,(*T*-1)	*D*[*I*;*J*]	The number of starts of facility installations of type *I* up to and including time period *J*. These activities are required to be integer.
P	*T*	*P*[*J*]	The net capital position at the end of time period *J*. The quantity *P*[*T*] is the capital position at the end of the time interval under consideration by the model; the objective is to maximize *P*[*T*].

The LP model for the installation scheduling problem is given by the following MDL program.

```
* THE INSTALLATION SCHEDULING PROBLEM
*    (*) "Solving the installation scheduling problem using
*         mixed integer linear programming," by R. Chen,
*         H. Crowder, and E.L. Johnson, IBM Systems Journal 17,
*         (1978) pp. 82-91.
$MDLINIT
DEFCOL: X,(N,T); P,T; D,(N,T-1),'I'
DEFRHS: RHS
*  Note: The following relation references are to the formulation
*        in (*), page 85.

*  Relation (1):
DEFROW R1,,'N'  * OBJ FUNCTION
ROW: R1; ΔSD (P[T])

* Relation (2):
DEFROW: R2,N
ROW: R2; ΔGR[2] X; M ΔSD RHS
```

```
⍝ Relation (3):
DEFROW: R3
ROW: R3; (C[;1]) ΔSD (X[;1]); ΔSD (P[1]); E ΔSD RHS

⍝ Relation (4):
DEFROW: R4,(T-1)
⍝ I←⍳T-1
⍝ BB←I∘.+[2]¨⊃¨I∘.[2]¨⊂B
⍝ XX←(-I)∘.+[2]¨⊂⍤⊃X
ROW: R4; BB ΔSD XX

⍝ Relation (5):
DEFROW: R5,(N,T-1),'L'
ROW: (⊂[1]R5); ΔGR[2] ((⍳T-1)∘.+[2]¨⊂X); ⍨1 ΔSD (⊂[1]D)

⍝ Relation (6):
DEFROW: R6,(N,T-2),'G'
ROW: (⊂[1]R6); ΔGR[2] (1+⍳T-2)∘.+[2]¨⊂X); ⍨1 ΔSD (⊂[1]⍨1+[2]D)
```

APPENDIX 1: BNF DESCRIPTION OF THE MDL SYNTAX

What follows is a description of the MDL syntax in Backus-Naur Form (BNF).

```
<MDL-statement>          :=  <id-def-statement>
                             <function-statement>
                             <APL-exec-statement>
                             <comment-statement>

<id-def-statement>       :=  <id-keywrd1> ; <id-phrase1>
                             <id-keywrd2> ; <id-phrase2>

<id-keywrd1>             :=  DEFROW
                             DEFCOL
                             DEFBND

<id-phrase1>             :=  <id-string1>
                             <id-phrase1> ; <id-string1>

<id-string1>             :=  <id-name> , <shape> , <attribute>

<id-keywrd2>             :=  DEFRHS

<id-phrase2>             :=  <id-string2>
                             <id-phrase2> ; <id-string2>

<id-string2>             :=  <id-name> , <shape>

<id-name>                :=  APL object name

<shape>                  :=  APL numeric scalar or vector constant
                             APL variable
                             (APL expression)
                             <null>

<attribute>              :=  APL character scalar or constant
                             APL variable
                             (APL expression)
                             <null>

<function-statement>     :=  <f-keywrd> : <id-array> ; <f-phrase>

<f-keyword>              :=  ROW
                             R
                             COL
                             C
```

```
<id-array>              :=  <id-name>
                            (APL expression)

<f-phrase>              :=  <f-string>
                            <f-phrase> ; <f-string>

<f-string>              :=  <data> <function1> <id-array>
                            <id-array> <function2> <id-array>

<data>                  :=  APL numeric scalar or vector constant
                            APL variable
                            (APL expression)
                            <null>

<function1>             :=  ΔGR<axis>
                            ΔSD
                            ΔAD

<axis>                  :=  [APL numeric scalar or vector constant]
                            [APL variable]
                            [APL expression]
                            <null>

<function2>             :=  ΔSN
                            ΔCN


<APL-exec-statement>    :=  ⍝ APL expression


<comment-statement>     :=  ⍝ Comment
```

APPENDIX 2: DISTRIBUTION FUNCTION ALGORITHMS

This Appendix defines the actions of the functions described in "Distribution Functions" on page 88. In particular, the results produced by distribution functions are derived in terms of the function's arguments.

Representing Results

The result of distribution functions is a matrix in sparse format, represented by a 2 column matrix *⍉IJ* and a list *⍉A*. Recall that *⍉IJ[K;]* gives the address (that is, row and column index) of *⍉A[K]*. In the discussions that follow, we will, for simplicity, assign to the variable *I* the contents of *⍉IJ[;1]* (that is, the list of resulting row indices) and to the variable *J* the contents of ⍉IJ[;2] (that is, the list of resulting column indices). The understanding is, of course, that *⍴I* ↔ *⍴J* and *⍉IJ* ↔ *⍉(2,⍴I)⍴I,J* .

Implicit Conformability

In the discussions that follow, we assume that all arguments are conformable according to the rules in "Distribution Functions" on page 88. In particular, we will not be concerned with the replication of singular arguments and, in the case of Generalized

Reduction Distribution, with replication and extension of axes (these latter topics were coved in detail in "Generalized Reduction Distribution" on page 90).

Generalized Reduction Distribution

Arguments:

R array of row identifier data

C array of column identifier data

D array of problem data

A list of coordinate indices of *C* and *D*

Conformability:

```
(⍴C)≡⍴D
(⍴R)≡(⍴C)[(~(⍳⍴⍴C)∈A)/⍳⍴⍴C]
```

Algorithm:

```
ZA ← ,(⍋⍋(⍳⍴⍴D)∈A)⍉D
I  ← ,(¯1⌽⍳⍴⍴R)⍉R←((×/((⍳⍴⍴C)∈A)/⍴C),⍴R)⍴R
J← ← ,(⍋⍋(⍳⍴⍴D)∈A)⍉C
```

Scalar Distribution

Arguments:

R array of row identifier data

C array of column identifier data

D array of problem data

Conformability:

```
(⍴R)≡⍴C
(⍴C)≡⍴D
```

Algorithm:

```
ZA ← ,D
I  ← ,R
J  ← ,C
```

Assign Distribution

Arguments:

R array of row identifier data

C array of column identifier data

D array of problem data

Conformability:

```
(ρD)≡(ρR),ρC
```

Algorithm:

```
ZA ← ,D
I  ← ,⍳(⊃ρD)ρ⍳R
J  ← ,(ρD)ρC
```

Scalar Network Distribution

Arguments:

RF array of row identifier data

RT array of row identifier data

C array of column identifier data

Conformability:

```
(ρRF)≡ρRT
(ρRF)≡ρC
```

Algorithm:

```
ZA ← ((⍴,RF)ρ1),(⍴,RT)⍴¯1
I  ← (,RF),,RT
J  ← (,C),,C
```

Cartesian Network Distribution

Arguments:

RF array of row identifier data

RT array of row identifier data

C array of column identifier data

Conformability:

```
(ρC)≡(ρRF),ρRT
```

Algorithm:

```
F ← ,((-ρρRT)φ⍳ρρF)⌽F←((ρRT),ρRF)ρRF
T ← ,((ρRF),ρRT)ρRT

ZA ← ((ρF)ρ1),(ρT)ρ¯1
I  ← F,T
J  ← (,C),,C
```

REFERENCES

[Bis77] Bisschop, J. and A. Meeraus, "Towards a General Algebraic Modeling System," Development Research Center, World Bank (Washington, D.C., 1977)

[Bro71] Brown, J.A., A Generalization of APL, Doctoral Thesis (Syracuse University, 1971)

[Bro79] Brown, J.A., "Evaluating Extensions to APL," APL Quote Quad 9 (1979) pp. 148-155

[Che78] Chen, R., H. Crowder, and E.L. Johnson, "Solving the Installation Scheduling Problem Using Mixed Integer Linear Programming," IBM Systems Journal 17 (1978) pp. 82-91

[Cro81] Crowder, H., "An APL-MPSX/370 Linear Programming Data Interface," IBM Research Report RC9014 (Yorktown Heights, 1981)

[Dan63] Dantzig, G.B., Linear Programming and Extensions (Princeton University Press, 1963)

[Ela81] Elam, J.J., and D. Klingman, "NETGEN-II: A System for Generating Structured Network-Based Mathematical Programming Test Problems," in Proceedings, Evaluating Mathematical Programming Techniques, J.M. Mulvey, Editor (Springer-Verlag, 1982)

[Fal73] Falkoff, A.D., and K.E. Iverson, "The Design of APL," IBM Journal of Research and Development 17 (1973) pp. 324-334

[For79] Forrest, J.J.H., and H. Crowder, "Current State of Computer Codes for Discrete Optimization," Annals of Discrete Mathematics 5 (1979) pp. 271-274

[Fou78] Fourer, R. and M. J. Harrison, "A Modern Approach to Computer Systems for Linear Programming," Working Paper 988-78, Alfred P. Sloan School of Management (Massachusetts Institute of Technology, 1978)

[Gha73] Ghandour, Z., and J. Mezei, "General Arrays, Operators and Functions," IBM Journal of Research and Development 17 (1973) pp. 335-352

[Gul79] Gull, W.E., and M.A. Jenkins, "Recursive Data Structures in APL," Communications of the ACM 22 (1979) pp. 79-96

[Hav76] Haverly Systems, Inc., Omni Linear Programming System: User and Operating Manual (First Edition, 1976)

[Hed75] Hedges, T. S., "Mathematical Programming Language: MPL/C Reference Manual," Systems Optimization Laboratory Technical Report SOL 75-22 (Stanford University, 1975)

[Ibm74] IBM Corporation, Matrix Generator and Report Writer (MGRW) Program Reference Manual. Form number SH19-5014 (1974)

[Ibm78] IBM Corporation, APL Language. Form number GC26-3847 (1978)

[Ibm79] IBM Corporation, IBM Mathematical Programming System Extended/370 (MPSX/370) Program Reference Manual. Form number SH19-1095 (1979)

[Ibm82] IBM Corporation, APL2 Language Manual. Form number SB21-3015 (1982)

[Ive62] Iverson, K.E., A Programming Language (Wiley, 1962)

[Ive76] Iverson, K.E., Elementary Analysis (APL Press, 1976)

[Ive78] Iverson, K.E., "Operators and Functions," IBM Research Report RC7091 (Yorktown Heights, 1978)

[Jar78] Jarvis, J. and C. Papaconstadopoulos, "A Methodology for Designing High Level Computer Input Systems for Mathematical Programming Models," Industrial and Systems Engineering Report Series (Georgia Institute of Technology, 1978)

[Kat80] Katz, S., L.J. Risman and M. Rodeh, "A System for Constructing Linear Programming Models," IBM Systems Journal 19 (1980) pp. 505-520

[Mil77a] Mills, R.E., R.B. Fetter, and R.F. Averill, "A Computer Language for Mathematical Program Formulation," Decision Sciences 8 (1977) pp. 427-444

[Mil77b] Mills, R. E., R. B. Fetter, and R. F. Averill, Conversational Modeling Language Reference Manual (Yale University Institution for Social and Policy Studies, 1977)

[Mor73] More, T., "Axioms and Theorems for a Theory of Arrays," IBM Journal of Research and Development 17 (1973) pp. 135-175

[Mor75] More, T., "A Theory of Arrays with Applications to Databases," IBM Cambridge Scientific Center Report G320-2107 (1975)

[Mor76a] More, T., "Types and Prototypes in a Theory of Arrays," IBM Cambridge Scientific Center Report G320-2112 (1976)

[Mor76b] More, T., "On the Composition f Array-Theoretic Operations," IBM Cambridge Scientific Center Report G320-2113 (1976)

[Mor79a] More, T., "The Nested Rectangular Array as a Model of Data," APL Quote Quad 9 (1979) pp. 55-73

[Mor79b] More, T., "Nested Rectangular Arrays for Measures, Addresses, and Paths," APL Quote Quad 9 (1979) pp. 156-163

[Mor81] More, T., "Notes on the Diagrams, Logic and Operations of Array Theory," IBM Cambridge Scientific Center Report G320-2137 (1981)

[Pie74] Project Independence Report, Federal Energy Administration (Washington, D.C., 1974)

[Pol75] Polivka, R.P., and S. Pakin, APL: The Language and Its Usage (Prentice-Hall, 1975)

[Rab82] Rabenhorst, D.A., private communication.

[Wil78] Williams, H.P., Model Building in Mathematical Programming (Wiley, 1978)

[Woo76] Woods, D. R., "Mathematical Programming Language -- User's Guide," Computer Science Department Report STAN-CS-76-561 (Stanford University, 1976)