CONTENTS

INTRODUCTION 1

1. PARTITION FUNCTIONS 3

1.1 A Programming Technique for Non-Rectangular Data 3

1.2 Using Partition Functions 11
1.2.1 An Example 11
1.2.2 Conventions 12
1.2.3 Errors 13
1.2.4 Table of Corresponding Names 13

1.3 Shift and Compare Functions *N*∆ and *P*∆ 15
1.3.1 Identities 16
1.3.2 Examples 16

2. REFERENCE TABLES 17

2.1 Composition Tables for the Relational and Logical Functions 17

2.2 Equivalences of Non-Boolean Functions on Boolean Data 17

2.3 Simplifications for Boolean Functions Using Tilde 18

2.4 Monadic Equivalents of Dyadic Boolean Functions with One Argument Constant 18

2.5 Equivalences for *A*⊚*A*⊚*B* in the Form of *A*⊚*B* 19

2.6 Distributive Identities for Boolean Functions 20

2.7 Inverse Distributive Identities for Boolean Functions 21

2.8 Interpretations of Reductions on a Boolean Vector Where the Result Is Zero or One 22

2.9 Similarities between Reduction and Scan of Boolean Vectors 23

2.10 Fast Algorithms for Implementing Relational and Logical Scans Other Than ≠\ and =\ 24

2.11 Eight Different Definitions of Scan, with Examples 24

2.12 Identities Involving Scan 24

2.13 Results of Operations Using Boolean Vectors 25

2.14 Altering the Order of Arguments Using Conditional Arrays 25

3. SELECTED FUNCTIONS THAT ILLUSTRATE BOOLEAN TECHNIQUES 27

INTRODUCTION

One of APL's unique contributions to computing languages is its ability to generate and manipulate Boolean arrays. Although a binary (or bit) datatype is available in many other languages, it is often clumsy to use, requiring conversion and indirect manipulation. APL, on the other hand, handles Boolean data directly and efficiently; the data is stored compactly (one value per bit) and processed rapidly.

Boolean values are most often generated by one of the six relational functions (< ≤ = ≥ > ≠) or by membership (∊). Five other functions perform logical calculations: and (∧), or (∨), nand (⍲), nor (⍱), not (~). Together, these provide all the nontrivial functions of logical calculus. For instance, exclusive-or is ≠ and logical implication is ≤.

APL's real strength with Boolean values lies in its ability to use them with every primitive function that allows numeric arguments. Some functions, such as compression (/) and expansion (\), are designed specifically for Boolean arguments; these allow us to select from, and insert values into, arrays. Other functions for which Boolean arguments are most useful are rotate (⌽), grade up (⍋), grade down (⍒), base value (⊥), and several scalar dyadic functions (+ - × ÷ | ! ⋆). The reduction and scan operators also gain new significance with Boolean arguments.

One of the most powerful applications of Boolean values in APL is their use in partition functions. Partition functions are user-defined functions that apply APL primitive functions independently to different parts of an array. They allow the user to manipulate a collection of arrays as independent objects, similar to the way this is done in the APL*PLUS Nested Arrays System (see <u>APL*PLUS Nested Arrays System Reference Manual</u>, STSC, 1981).

<u>Boolean Functions and Techniques</u>, Second Edition, serves two purposes. First, it is an introduction to the utility workspace 6 *PARTFNS* on STSC's APL*PLUS† System. Second, it provides the experienced programmer with useful reference tables and formal definitions using Boolean data.

----

†APL*PLUS is a service mark and trademark of STSC, Inc., registered in the United States Patent and Trademark Office.

Chapter 1 includes transcriptions of the online documentation for workspace 6 *PARTFNS* as well as a paper discussing the theory and use of partition functions. This chapter should be read in one sitting, but parts of it may also be used for reference.

Chapter 2 is intended primarily as a reference for the experienced APL programmer. The charts in this chapter contain practical information that can be applied by the programmer as needed.

Chapter 3 presents sample functions using Boolean techniques. These are intended to give the reader some idea of the practical uses of Boolean techniques and spur him on to develop his own applications.

The second edition of Boolean Functions and Techniques supersedes Working Memorandum No. 106, Boolean Functions and Techniques -- Special Topics Seminar. The editor acknowledges the text contributions and review comments of Bob Smith, who wrote the earlier version of this publication. The editor is indebted to Roy A. Sykes, Jr. for his technical assistance, text contributions, and review suggestions. In producing this publication, the editor gratefully acknowledges the assistance of Andrea G. Kenner, Deborah Richardson, and Suzanne Yanchulis.

Notational Conventions

The following syntax conventions are used throughout this publication:

- Function and operator symbols are shown exactly as they appear in actual usage.
- The symbol ⊚ is defined as any logical or relational function.
- The double-headed arrow (↔) indicates that two expressions have the same meaning or return identical results.
- The symbol ∝ is defined as any primitive function.
- Definitions and examples are given in origin 1 unless otherwise noted.

CHAPTER 1

PARTITION FUNCTIONS

This chapter includes a paper by Bob Smith on the theory and use of partition functions and the documentation for partition functions and *N∆P∆GRP* from workspace 6 *PARTFNS*.

Bob Smith's paper was published in the APL79 Conference Proceedings, Part 1 (Association for Computing Machinery, 1979). It is reprinted here by kind permission of the Association for Computing Machinery. The paper provides a concise discussion of the need for partition functions and also illustrates their use.

The two items of online documentation transcribed in this chapter are normally accessed through the function *DETAIL* in workspace 6 *PARTFNS*. Together, they provide an overview of the capabilities available on STSC's APL*PLUS system using the utility functions in that workspace. Section 1.2 includes a table that relates each partition function to its corresponding APL expression, and Section 1.3 includes a chart that illustrates the use of *N∆* and *P∆* with Boolean functions.

1.1 A Programming Technique for Non-Rectangular Data

The text appearing on pages 4 through 10 was reprinted by permission of the Association for Computing Machinery.

A PROGRAMMING TECHNIQUE FOR NON-RECTANGULAR DATA

Bob Smith
APL Application Analyst
Scientific Time Sharing Corporation
21243 Ventura Blvd., Suite 240
Woodland Hills, CA 91364
(213) 347-1633

Abstract

A programming technique is developed to deal with certain common operations on non-rectangular (ragged) data. A representation of such data in current *APL* is defined; a notation for applying primitive functions to this data is developed; and user-defined functions that simulate these operations for certain primitive monadic functions are illustrated.

Motivation

The coordinates of matrices and higher-dimensional arrays serve to delimit the values of the rectangular data into distinct "lines" (e.g., the rows of a matrix). Primitive functions (e.g., reduction and scan) then may be applied independently to each distinct line to produce various results.

Occasionally, however, it becomes necessary to apply a primitive function independently to successive parts of a vector (called the argument vector). These parts are analogous to the lines of a higher-dimensional array. For example, the problem may be to plus-reduce certain parts of a numeric vector. If the argument vector were ⍳10 and if it were broken into several parts as follows:

```
1 2 3   4   5 6 7 8   9 10
```

then the desired result would be 6 4 26 19.

In this case, the non-rectangular data is represented as an argument vector along with a vector of the lengths which partition the argument vector (here, the implicit lengths are 3 1 4 2). One might think of the argument vector as a vector of vectors.

Previously, this kind of problem has been solved by first expanding and reshaping the argument vector into a matrix. This transformation allows the programmer to take advantage of the delimiting property of the coordinates of the matrix and apply the primitive function to the entire matrix to obtain the result. If the argument vector were stored in *B* and the lengths of the successive parts stored in *A*, a solution is

```
+/((⍴A),⌈/A)⍴(,A∘.≥⍳⌈/A)\B .
```

As a general method, the argument vector-to-matrix approach has several drawbacks.

First, the expansion result may not fit in the workspace. Because the length of the expansion result is the number of parts times the length of the longest part, it only takes one long part and many small parts to create a nuisance. *WS FULL* is most programmers' least favorite error message.

Second, the expansion process may pad each part (on the way to creating a row of the matrix) with an inappropriate value. The 0's that expansion uses as fillers for numeric structures won't bother +/, but ×/ and ∧/ won't like them. This problem can be circumvented; however, it's one more thing with which the programmer must be bothered.

Third, this approach can be expensive because of all the extra work involved in inserting fill elements via expansion.

The following sections discuss an alternate approach to solving this class of problems. Essentially, these problems involve representing non-rectangular data as a vector of vectors along with a partition, and applying a primitive function to each independent part. First, we must characterize the concept of "parts", and then combine that with the primitive function and the argument vector to produce the result.

Partition Vectors

A simple and unique characterization of any partitioning of an argument vector is as a Boolean vector of the same length (called a partition vector). Each 1 in the partition vector corresponds to the beginning of a part and the following 0's correspond to the remaining elements in that part.

The partition of ⍳10 used above would be characterized as

```
       1 2 3   4   5 6 7 8   9 10
       1 0 0   1   1 0 0 0   1  0
```

It is easy to see that this characterization is unique and that every Boolean vector of the same length as the argument vector and whose first element is 1 represents a partition of the argument vector. There are 2⋆*N*-1 such partitions of length *N*. Note that the number of parts represented by the partition vector *P* is +/*P*.

The characterization as a Boolean vector is chosen over that of a vector of successive lengths of the parts for several reasons. The Boolean vector often is easily computed from and stored with the argument vector (as in blanks in a character vector); has a simpler conformability test (*P*[1]∧(⍴*P*)=⍴*B* vs. ((+/*P*)=⍴*B*)∧∧/*P*≥0); often uses less storage than an integer vector; and combines more naturally with the user-defined functions illustrated later.

Although a representation as a vector of lengths would allow zero lengths, there is no real added capability. The zero lengths have no effect in conjunction with shape-preserving functions like scan and grade. For reduction, presumably one fills in the result with the appropriate identity element. I prefer to leave that job to the expansion function.

The Partition Operator

We now need to combine the partition vector, the primitive function, and the argument vector. One method of doing this is to combine the partition vector and the primitive function into a derived function via an operator. This operator (called the partition operator) is then dyadic, with a partition vector on the left and a primitive function on the right. The result is a derived function (a partition function) which has the same valence as the original primitive function. That is, the resulting function is dyadic if the original primitive function is dyadic; monadic otherwise.

For simplicity, a new symbol is used for this operator. It is not the goal of this paper to suggest that this operator or symbol be implemented; it is introduced only as notation to simplify the presentation.

Definition: partition operator †

(which Jim Ryan calls the "dagger" symbol)

Let ⍺ be any primitive monadic function, and *P* and *B* any vectors of equal length where *P* is a Boolean vector whose first element is 1.

```
      R ← P†⍺ B
```

is the result of applying ⍺ to each part of *B*, where *P* defines the partition. Specifically,

```
       R←⍳0
       I←0
      L1:→((+/P)<I←I+1)/0
       R←R, ⍺ (I=+\P)/B
       →L1
```

The shape of *R* depends upon ⍺. If ⍺ is a reduction function, the shape is +/*P* (one value per part since reduction reduces vectors to scalars); if ⍺ is a shape-preserving function, the shape is ⍴*P* (e.g., scan preserves shape). If ⍺ is defined for higher-dimensional arrays, then so is *P*†⍺, and in the same way. Notationally, this would be written as *P*†(⍺[*K*]) *B*. For example, *P*†(+\[2]) *B*.

If ⍺ is a dyadic function, the syntax of the partition operator is

```
      R ← A (P†⍺) B
```

and the result is calculated as

```
       R←⍳0
       I←0
      L1:→((+/P)<I←I+1)/0
       R←R,((I=+\P)/A) ⍺ (I=+\P)/B
       →L1
```

See Iverson [1] for a discussion of operators which take data as arguments.

The above example of a partitioned plus reduction of ⍳10 would be written as

```
      1 0 0 1 1 0 0 0 1 0 †(+/) ⍳10
6 4 26 19
```

or as a function of *A* and *B* with *B*←⍳10 and *A*←3 1 4 2 (the partition lengths) as

```
      ((⍳+/A)∊1++\¯1↓0,A)†(+/) B
6 4 26 19
```

Applications

I. In accounting applications, one encounters the problem of accumulating amounts of money into bins. For example, *AMT* might be a vector of amounts, and *ACCT* might be a vector of corresponding account codes. Potentially, there are amounts in

AMT to be charged to the same account code; thus *ACCT* contains duplicate values. For each distinct account code, how much is to be charged to it?

A solution is first to reorder *ACCT* (and in the same way reorder *AMT*) such that equal account codes are adjacent.

```
      ORD←⍋ACCT
      AMT←AMT[ORD]
      ACCT←ACCT[ORD]
```

Next, create a partition vector of equal account codes,

```
      PV←ACCT≠¯1⌽ACCT
      PV[1]←1 .
```

For example,

```
 AMT   ↔   3.11 ¯6.20  9.12 ¯4.50  6.41  7.93
ACCT   ↔     83    83    83   101   101   201
  PV   ↔      1     0     0     1     0     1
```

At this point, *PV* selects the distinct account codes.

```
      PV/ACCT
83 101 201
```

Finally, apply a partitioned plus reduction to *AMT* using *PV* to obtain the amounts to charge to each distinct account code.

```
      PV†(+/) AMT
6.03 1.91 7.93
```

II. A tape contains several records, each of which begins with a record mark. Individual records may contain errors that invalidate the remaining information in that record, but not other records. Which records contain errors?

If the Boolean vector *RM* marks the beginning of each record, and the Boolean vector *ERR* selects the errors, then a selection vector on the indices of the records that contain errors is

```
      A←RM†(∨/) ERR
```

For example,

```
       RM   ↔  1 0 0 1 1 0 1 0 0 0
      ERR   ↔  0 0 1 1 0 0 0 0 1 0
        A   ↔  1     1 0   1
```

To return a Boolean vector that selects leading error-free information in each record, use

```
      B←RM†(∧\) ~ERR
```

```
       RM   ↔  1 0 0 1 1 0 1 0 0 0
      ERR   ↔  0 0 1 1 0 0 0 0 1 0
        B   ↔  1 1 0 0 1 1 1 1 0 0
```

Problem Isolation

One advantage of partition functions is their ability to be used to extend isolated solutions to more general solutions. That is, when working with non-rectangular data represented as a vector of vectors, often one can solve the problem at hand on an isolated part, and then substitute in the appropriate partition functions to obtain a solution on the entire partitioned vector.

For example, say one has a character vector (*TXT*) partitioned by new-line characters (carrier returns); thus, the first character is a new-line character. Each part is the character image of a line of some user-defined function. The problem is to select text in comments and in character constants. Assume new-line characters are not allowed in character constants, but that comments are allowed at the end of a line with an executable statement.

Let's solve this problem for a single part of the character vector, say, *PART*.

Because a character constant is delimited by quotes, text within a character constant has to its left an odd number of quotes. Characters preceded by an odd number of quotes can be selected by ≠\ as in

```
      A←≠\PART='''' .
```

Comment delimiters are then simply lamp symbols outside of character constants, that is

```
      A<PART='⍝' .
```

The text in comments is then the characters to the right of a comment delimiter, that is

```
      ∨\A<PART='⍝' .
```

Finally, text in comments and character constants is selected by

```
      A←≠\PART=''''
      CQ←A∨∨\A<PART='⍝' .
```

To generalize this solution on a single part to a solution on the entire vector without looping is a matter of substituting in the appropriate partition functions as in

```
      P←TXT=CR
      A←P†(≠\) TXT=''''
      CQ←A∨P†(∨\) A<TXT='⍝'   .
```

Relationship with General Arrays

Broadly speaking, general arrays are a proposed extension to *APL* to handle non-rectangular data (see, for example, Gull and Jenkins [2]). As one would expect, general arrays handle more naturally the problems which partition functions address. Specifically, a vector of vectors is represented as a single data structure in general arrays rather than as separate argument and partition vectors; in other words, the partition is implicit in the structure.

The correspondence between partition functions and general arrays is provided by a handful of functions defined on general arrays.

1. a function to create a vector of vectors from a pair of argument and partition vectors;
2. a function to apply a primitive function independently to each part of the vector of vectors; and
3. a function to rejoin the separate parts of a vector of vectors into a simple vector.

The first function is discussed in reference [2] as the dyadic function split which takes a partition vector on the left and an argument vector on the right. The result is a vector of vectors. Another approach is to define a dyadic form of the function seal (see reference [2]) called partitioned seal. I prefer the latter approach because it combines in one symbol two closely related functions.

The second function is discussed by Iverson [1] as a dyadic dual operator.

The third function is akin to an unseal (an inverse seal) of a catenate reduction.

For example, the problem mentioned in the last section is solved using general arrays as follows, where the symbols mean

```
    P⊂A          partitioned seal
     ⍤̈           dual operator
     ⊃           unseal.

     P←TXT=CR
     A←≠\⍤̈⊃P⊂TXT=''''
     CQ←⊃,/A∨∨\⍤̈⊃A<P⊂TXT='⍝'  .
```

Nevertheless, although general arrays provide a more general solution to the problems addressed by partition functions, until they are available on your *APL* system, use simulations of partition functions.

Tools and Techniques

This section describes user-defined functions which simulate monadic partition functions. The limitation to monadic partition functions is made only to simplify the function's syntax: only two data arguments are required (the partition and argument vectors).

The set of monadic primitives of interest as partition functions are the thirteen primitive scalar monadic functions, reduction, scan, grade up, grade down, and reversal. For the thirteen primitive scalar monadic functions, *P*⍭α ↔ α; that is, if the primitive function has no interaction between elements, the partitioning has no effect.

Simulations of eight of the twenty-one primitive scalar dyadic functions in both scan and reduction form are listed in Appendix A. Partition functions which simulate grade up, grade down, and reversal are also included. Each function listed in Appendix A is non-looping and works on vector arguments only; All work in either origin; and *P*⍭⍋ and *P*⍭⍒ return results sensitive to the origin.

A tool used in simulating these functions is the function *N*Δ (compare with the function *PAIR* in Halpern [3]). The function is defined as follows:

```
      'α' NΔ V  ↔  Vα¯1↓(α/⍳0),V
```

The left argument must be a character scalar symbol for a primitive scalar dyadic function with a right-hand identity element. Essentially, using the function α, *N*Δ compares *V* with itself shifted one to the right (that is, negatively; hence the "*N*" in its name). The shift of *V* drops off the last element and catenates the identity element of α to the beginning. For example,

```
      '≠' NΔ V  ↔  V≠¯1↓0,V
```

A companion function, *P*Δ, shifts to the left (positively) and again fills the hole (this time at the end) in the shifted vector with the identity element of α.

```
      'α' PΔ V  ↔  Vα1↓V,α/⍳0
```

In a sense, *N*Δ and *P*Δ are operators that take a primitive scalar dyadic function with a right-hand identity element and return a monadic function which performs a "differencing" (hence the "Δ" in their names). Halpern also notes that *PAIR* is the inverse of certain scans, as is *N*Δ. For example, '≠' *N*Δ and ≠\ are left and right inverses of each other as are '=' *N*Δ and =\; '-' *N*Δ and +\; and '÷' *N*Δ and ×\. See Appendix B for specific examples. Iverson [1] has proposed a more general function in dyadic scan. In particular,

```
      'α' NΔ V  ↔  ¯2 α\ V

      'α' PΔ V  ↔  ⌽¯2 α\ ⌽V
                ↔  ¯2 α\⍤̈⌽ V
```

The techniques used in simulating the partition functions are interesting enough to warrant stepping through some examples. Note that for a partition vector *P*, the 1's in *P* are the starting points of each partition, while the 1's in 1⌽*P* are the corresponding endpoints.

```
I.   P†(+\) B  ↔   +\B-P\'-' NΔ P/+\¯1↓0,B

Let
     P           1  0  0     1     1  0  0  0
     B           1  2  3     4     5  6  7  8
1.   +\¯1↓0,B    0  1  3     6    10 15 21 28
2.   P/          0           6    10
3.   '-' NΔ      0           6     4
4.   P\          0  0  0     6     4  0  0  0
5.   B-          1  2  3    ¯2     1  6  7  8
6.   +\          1  3  6     4     5 11 18 26
```

Steps 1 and 2 compute, for each partition of *B*, the cumulative sum of all the previous partitions. Step 3 isolates, for each partition, the contributions of the immediately preceding partition. Step 4 puts the result of Step 3 into the positions of the beginning of each partition, and Step 5 subtracts this from *B*. The result at this point is similar to *B*, but with the initial value in each partition short by the sum reduction of the immediately preceding partition. After Step 6 is executed, each partition has the bias from any previous partition accounted for correctly.

Pleasantly, *P*†(≠\) *B* falls out of *P*†(+\) *B* because for a Boolean vector *B*, ≠*B* ↔ 2|+*B*. Thus,

```
P†(≠\) B  ↔  2|P†(+\) B
          ↔  2|+\B-P\'-' NΔ P/+\¯1↓0,B
          ↔  ≠\2|B-P\'-' NΔ P/+\¯1↓0,B
          ↔  ≠\B≠2|P\'-' NΔ P/+\¯1↓0,B
          ↔  ≠\B≠P\2|'-' NΔ P/+\¯1↓0,B
          ↔  ≠\B≠P\'≠' NΔ 2|P/+\¯1↓0,B
          ↔  ≠\B≠P\'≠' NΔ P/2|+\¯1↓0,B
          ↔  ≠\B≠P\'≠' NΔ P/≠\¯1↓0,B
```

The simulation for *P*†(=\) *B* is derived using the identity =*B* ↔ ~≠\~*B* .

```
II.  P†(+/) B  ↔  '-' NΔ(1⌽P)/+\B
```

This code finds the cumulative sum at the endpoints of each partition and differences adjacent pairs of values to obtain the result. Appendix A contains another function (*PPLREDB̲*) to simulate *P*†(+/) *B* that requires less workspace storage than the above method when *B* also is a Boolean vector.

The simulation for *P*†(≠/) *B* is derived in the same way as *P*†(≠\) *B* was derived.

```
III. P†(∨\) B  ↔  ≠\(P∨B)\'≠' NΔ(P∨B)/B

Let
     P           1 0 0   1   1 0 0 0   1 0
     B           0 1 0   0   1 0 0 1   0 0
1.   P∨B         1 1 0   1   1 0 0 1   1 0
2.   (P∨B)/B     0 1     0   1     1   0
3.   '≠' NΔ      0 1     1   1     0   1
4.   (P∨B)\      0 1 0   1   1 0 0 0   1 0
5.   ≠\          0 1 1   0   1 1 1 1   0 0
```

Step 1 points out that the important information in *P* and *B* is where either of them is 1. Where both of them are 0, the value in the corresponding position in the result equals the value in the immediately preceding position in the result. The leading 1's in each set of consecutive 1's in Step 2 indicate a 1 in *B* where the preceding elements in that same partition of *B* were all 0. This position in *B* represents a point at which we wish to begin smearing 1's to at least the end of that partition. The actual point at which we should stop smearing 1's is where the initial element in some partition of *B* is 0. These stopping points are exactly the points in *B* corresponding to the 0's in Step 2.

Step 3 results in a 1 at both the start and stop points of each smear of 1's. Step 4 puts these values back into place relative to *B*, and Step 5 performs the smear.

The relationships between the various steps in this solution are interesting. Steps 2 and 4 are a compression and expansion using the same left argument. Steps 3 and 5 are inverses of each other.

The simulation for *P*†(∧\) *B* is derived using the identity ∧*B* ↔ ~∨\~*B* .

Similarly, the simulation for *P*†(</) *B* is derived using *P*†(</) *B* ↔ *P*†(∧/) *B*=1⌽*P* .

In Reverse

Occasionally, one needs to perform a partitioned scan or reduction on each reversed part, and, if scanning, to reverse the scan result again. The problem might be, given a character vector partitioned by new-line characters, to delete trailing blanks in each partition. One would like to or-scan each partition of *TXT*≠' ', except from the right rather than the left. This can be accomplished by reversing, or-scanning, and again reversing each partition. That is,

```
        PV←TXT=CR     (assuming TXT begins with
                        a new-line character)

        TXT←(PV†⌽ PV†(∨\) PV†⌽ TXT≠' ')/TXT
```

There is a simpler way, still using partition functions, that does not require two partitioned reversals. Notice that since each partition is treated independently from every other partition, one could reverse the entire vector rather than each partition, then scan or reduce (using a different partition vector), and, finally, reverse the entire result rather than each partition. This translates into

```
        ⌽(⌽1⌽PV)†α ⌽V
```

where α is either a scan or reduction function. Further, in terms of the functions in Appendix A, one can move this code inline and, in most cases, eliminate all occurrences of reversal by applying the appropriate identities. For example, comparing

```
    R1←      P +(∨\)  B   with
    R2←⌽(⌽1⌽P)+(∨\) ⌽B

R1  ↔  ≠\(P∨B)\'≠' N∆(P∨B)/B
R2  ↔  ⌽≠\((⌽1⌽P)∨⌽B)\ '≠' N∆((⌽1⌽P)∨⌽B)/⌽B
       ⌽≠\(⌽B∨1⌽P)\ '≠' N∆ (⌽B∨1⌽P)/⌽B
       ⌽≠\(⌽B∨1⌽P)\ '≠' N∆⌽(B∨1⌽P)/B
       ⌽≠\(⌽B∨1⌽P)\⌽'≠' P∆ (B∨1⌽P)/B
       ⌽≠\⌽(B∨1⌽P)\ '≠' P∆ (B∨1⌽P)/B
```

Finally, `⌽≠\⌽V  ↔  (¯1↓Z)≠¯1↓0,Z` with `Z←≠\V`.

Similar substitutions can be made for the other functions.

References

[1] Iverson, K. E., Operators and Functions, IBM Yorktown Heights Research Division, Research Report RC 7091, (April 78).

[2] Gull, W. E., and Jenkins, M. A., Recursive Data Structures in *APL*, Comm. ACM 22, 2 (Feb 79), 79-96.

[3] Halpern, M. M., Studies in *APL*: Algebra, Scan, Arithmetic, Permutations, IBM Philadelphia Scientific Center, Technical Report No. 320-3023, (June 1973).

APPENDIX A

The user-defined functions listed in this appendix simulate the partition operator in combination with various primitive functions. In particular, where *P* and *B* are Boolean vectors and *V* is a numeric vector, the correspondence is as follows:

```
P⍭(∧/) B  ↔  P PANDRED B           P⍭(⌈/) V  ↔  P PMAXRED V
P⍭(∧\) B  ↔  P PANDSCAN B          P⍭(⌈\) V  ↔  P PMAXSCAN V

P⍭(∨/) B  ↔  P PORRED B            P⍭(⌊/) V  ↔  P PMINRED V
P⍭(∨\) B  ↔  P PORSCAN B           P⍭(⌊\) V  ↔  P PMINSCAN V

P⍭(=/) B  ↔  P PEQRED B            P⍭⍋ V     ↔  P PGRADEUP V
P⍭(=\) B  ↔  P PEQSCAN B           P⍭⍒ V     ↔  P PGRADEDOWN V

P⍭(≠/) B  ↔  P PNERED B            P⍭(+/) V  ↔  P PPLRED V
P⍭(≠\) B  ↔  P PNESCAN B           P⍭(+\) V  ↔  P PPLSCAN V
                                   P⍭(+/) B  ↔  P PPLREDB B
P⍭(</) B  ↔  P PLTRED B
P⍭(<\) B  ↔  P PLTSCAN B           P⍭⌽ V     ↔  P PREVERSE V
```

```
     ∇ Z←P PANDRED V                          ∇ Z←P PANDSCAN V
[1]    Z←(V≤P)/P ⋄ Z←(Z/1⌽Z)∧P/V      [1]    Z←~≠\(V≤P)\'≠' NΔ~(V≤P)/V
     ∇                                        ∇

     ∇ Z←P PORRED V                           ∇ Z←P PORSCAN V
[1]    Z←(V∨P)/P ⋄ Z←(Z/1⌽Z)≤P/V      [1]    Z←≠\(V∨P)\'≠' NΔ(V∨P)/V
     ∇                                        ∇

     ∇ Z←P PEQRED V                           ∇ Z←P PEQSCAN V
[1]    Z←'=' NΔ(1⌽P)/=\V              [1]    Z←=\V≠P\'≠' NΔ~P/=\¯1↓1,V
     ∇                                        ∇

     ∇ Z←P PNERED V                           ∇ Z←P PNESCAN V
[1]    Z←'≠' NΔ(1⌽P)/≠\V              [1]    Z←≠\V≠P\'≠' NΔ P/≠\¯1↓0,V
     ∇                                        ∇

     ∇ Z←P PLTRED V                           ∇ Z←P PLTSCAN V
[1]    Z←(P≥V=1⌽P)/P                  [1]    Z←(V∧P)∨(V∨P)\'>' NΔ(V∨P)/V
[2]    Z←(Z/1⌽Z)∧P/V=1⌽P                      ∇
     ∇

     ∇ Z←P PMAXRED V                          ∇ Z←P PMAXSCAN V
[1]    Z←V[(⍒V)[P/⍋(+\P)[⍒V]]]        [1]    Z←⍋(⍋V)[⍋(+\P)[⍋V]]
     ∇                                [2]    Z←V[Z⍳⌈\Z]
                                              ∇

     ∇ Z←P PMINRED V                          ∇ Z←P PMINSCAN V
[1]    Z←V[(⍋V)[P/⍋(+\P)[⍋V]]]        [1]    Z←⍋(⍒V)[⍋(+\P)[⍒V]]
     ∇                                [2]    Z←V[Z⍳⌈\Z]
                                              ∇

     ∇ Z←P PGRADEUP V                         ∇ Z←P PGRADEDOWN V
[1]    Z←⎕IO+(⍋V)[⍋(+\P)[⍋V]]-⌈\P×⍳⍴P [1]    Z←⎕IO+(⍒V)[⍋(+\P)[⍒V]]-⌈\P×⍳⍴P
     ∇                                        ∇

     ∇ Z←P PPLRED V                           ∇ Z←P PPLSCAN V
[1]    Z←'-' NΔ(1⌽P)/+\V              [1]    Z←+\V-P\'-' NΔ P/+\¯1↓0,V
     ∇                                        ∇

     ∇ Z←P PPLREDB V
[1]    Z←((V∨P)/P),1                          ∇ Z←P PREVERSE V
[2]    Z←(1↓'-' NΔ Z/⍳⍴Z)-~P/V        [1]    Z←V[⌽⍒+\P]
     ∇                                        ∇
```

1.2 Using Partition Functions

Partition functions are user-defined functions that apply certain APL primitive functions independently to each partition of an array. They are used to manipulate a collection of arrays as independent objects, though the collection is stored in a single array.

A vector (a row of an array) can be viewed as a collection of smaller vectors called partitions. For example, ⍳10 can be partitioned into smaller vectors as follows:

```
    1 2 3    4    5 6 7 8    9 10
```

This partitioning can be represented by a Boolean vector of the same length with a 1 marking the beginning of each partition:

```
    1 0 0    1    1 0 0 0    1  0              (partition vector)
    1 2 3    4    5 6 7 8    9 10
```

The partition function for plus reduction (*PPLRED*) performs +/ on each part of the vector.

```
    1 0 0 1 1 0 0 0 1 0 PPLRED 1 2 3 4 5 6 7 8 9 10
6 4 26 19
```

By definition, the first element of a partition vector is 1.

1.2.1 An Example

A paragraph of text can be represented as a character vector (*V*) with new-line characters (*⎕TCNL*) separating the lines of the paragraph. A partitioning of this vector into its separate paragraph lines is

```
   P←¯1↓1,V=⎕TCNL
```

We would like to count the number of blanks in each line of the paragraph by applying the function +/ to each partition of *V*=' '. This can be accomplished by using *PPLRED*, the partition function corresponding to +/, as in

```
   P PPLRED V=' '
```

To count the number of leading blanks in each line of the paragraph, apply +/∧\ to each partition of *V*=' ':

```
   P PPLRED P PANDSCAN V=' '
```

1.2.2 Conventions

- The name of each partition function begins with the letter P̲.
- Each partition function is dyadic with the partition vector on the left and the argument array on the right.
- The result of each partition function is the result of applying the primitive function (as specified in the name of the partition function) independently to each partition of the right argument.
- Each partition function extends scalar or one-element vector right arguments to the length of the left argument (for example, to compute the lengths of the partitions in *P*, use *P P̲PLRED* 1).
- Each partition function requires the left argument to be a Boolean scalar or one-element vector, or a Boolean vector with ¯1↑⍴*B* elements (where *B* is the right argument). A scalar or one-element vector left argument is extended to a vector with ¯1↑⍴*B* elements. The first element of the left argument must be 1.
- Each partition function partitions the last coordinate of an arbitrary rank right argument. To apply the partition to another coordinate, transpose the right argument and result as appropriate. In particular, to apply any partition function *P̲FN* to the *K*th coordinate of *A*, use

```
        (⍋K=⍳⍴⍴A)⍉P P̲FN (⍋⍋K=⍳⍴⍴A)⍉A
```

- Each partition function checks both arguments for errors; if an error is found, it is signalled to the calling environment.
- In all partition functions, ⎕*ELX* is localized to signal errors to the calling environment as in

```
        ⎕ELX←'⎕ERROR(∧\⎕DM≠⎕TCNL)/⎕DM'
```

- The results of the following functions are sensitive to the setting of ⎕*CT*:

```
P̲DRANKUP      P̲HRANKUP      P̲LRANKUP
P̲DRANKDOWN    P̲HRANKDOWN    P̲LRANKDOWN
```

  In all other functions, ⎕*CT* is localized and set to 0.

- In the following functions, ⎕*IO* is localized and set to 0:

```
P̲ROTATE1     P̲SORTUP      P̲LJUST    P̲PLREDB    P̲REVERSE
P̲ROTATEN1    P̲SORTDOWN    P̲RJUST
```

  All other functions are either origin free or origin sensitive.

- The results of all partition functions with index, grade, or rank in their names are sensitive to the setting of ⎕*IO*.
- The following functions work on both numeric and character arrays:

*PSHIFT*1	*PROTATE*1	*PLJUST*	*PREVERSE*
*PSHIFTN*1	*PROTATEN*1	*PRJUST*	

All other functions work on numeric arrays only.

1.2.3 Errors

RANK ERROR	The left argument is not a scalar or vector.
LENGTH ERROR	The length of the left argument is not equal to the length of the last coordinate of the right argument.
DOMAIN ERROR	The left argument is not Boolean-valued, the first element of the left argument is not 1, or the value of the right argument is not suitable for the corresponding primitive function (e.g., *PPLRED* on a character array or *PANDRED* on a non-Boolean array).

1.2.4 Table of Corresponding Names

Table 1.1 lists the partition functions in workspace 6 *PARTFNS* and the corresponding APL expressions that are independently applied to each partition of the argument array *A*.

In particular, given *P* as the partition vector and *A* as the array right argument, let *IP* be the *I*th partition of *A*:

```
IP←(I=+\P)/A                    (in origin 1)
```

The result of each partition function is the catenation of the expression to the right of each partition function for each partition of *A*. If *IP* is a matrix or higher-dimensional array, consider the expressions below to illustrate what happens to each row of *IP*.

Table 1.1 -- Partition Functions and Corresponding APL Expressions

Function	APL Expression	Function	APL Expression
PLTRED	</*IP*	*PLTSCAN*	<\IP
PLERED	≤/*IP*	*PLESCAN*	≤\IP
PEQRED	=/*IP*	*PEQSCAN*	=\IP
PGERED	≥/*IP*	*PGESCAN*	≥\IP
PGTRED	>/*IP*	*PGTSCAN*	>\IP
PNERED	≠/*IP*	*PNESCAN*	≠\IP
PORRED	∨/*IP*	*PORSCAN*	∨\IP
PANDRED	∧/*IP*	*PANDSCAN*	∧\IP
PNORRED	⍱/*IP*	*PNORSCAN*	⍱\IP
PNANDRED	⍲/*IP*	*PNANDSCAN*	⍲\IP
PPLRED	+/*IP*	*PPLSCAN*	+\IP
PPLREDB	+/*IP* (See 1)		
PMINUSRED	-/*IP*	*PMINUSSCAN*	-\IP
PPRRED	×/*IP*	*PPRSCAN*	×\IP
PMINRED	⌊/*IP*	*PMINSCAN*	⌊\IP
PMAXRED	⌈/*IP*	*PMAXSCAN*	⌈\IP
PLJUST	(+/∧\IP=' ')⌽*IP* or (+/∧\IP=0)⌽*IP*	*PRJUST*	(-+/∧\\⌽*IP*=' ')⌽*IP* or (-+/∧\\⌽*IP*=0)⌽*IP*
PSHIFTN1	¯1↓0,*IP* or 1↓' ',*IP*	*PSHIFT1*	1↓*IP*,0 or 1↓*IP*,' '
PROTATEN1	¯1⌽*IP*	*PROTATE1*	1⌽*IP*
PREVERSE	⌽*IP*		
PINDEXUP	(See 2 and 3)	*PINDEXDOWN*	(See 2 and 3)
PSORTUP	*IP*[⍋*IP*]	*PSORTDOWN*	*IP*[⍒*IP*]
PGRADEUP	⍋*IP* (See 3)	*PGRADEDOWN*	⍒*IP* (See 3)
PRANKUP	⍋⍋*IP* (See 3 and 4)	*PRANKDOWN*	⍋⍒*IP* (See 3 and 4)
PDRANKUP	1 *NDRANKR IP* (See 3 and 4)	*PDRANKDOWN*	¯1 *NDRANKR IP* (See 3 and 4)
PLRANKUP	1 *NLRANKR IP* (See 3 and 4)	*PLRANKDOWN*	¯1 *NLRANKR IP* (See 3 and 4)
PHRANKUP	1 *NHRANKR IP* (See 3 and 4)	*PHRANKDOWN*	¯1 *NHRANKR IP* (See 3 and 4)
PARANKUP	1 *NARANKR IP* (See 3 and 4)	*PARANKDOWN*	¯1 *NARANKR IP* (See 3 and 4)

Notes for Table 1.1:

(1) For Boolean values only; uses less work area than *P̲PLRED*.

(2) *P̲INDEXUP* and *P̲INDEXDOWN* independently grade each partition of a numeric array, and return a permutation array that would sort each partition in the composite array. For partition vector *P* and numeric array *A*,

```
        (,A)[P P̲INDEXUP A]     ↔  P P̲SORTUP A
        (,A)[P P̲INDEXDOWN A]   ↔  P P̲SORTDOWN A
```

(3) Results of these functions are sensitive to the setting of ⎕*IO*.

(4) The rank functions listed above correspond to the functions of similar name from workspace 6 *SORT*.

1.3 Shift and Compare Functions *N*Δ and *P*Δ

The group *N*Δ*P*Δ*GRP* consists of the two shift and compare functions *N*Δ (negative shift) and *P*Δ (positive shift):

```
      '∝' NΔ V  ↔  V ∝ ¯1↓ (∝/⍳0) , V
      '∝' PΔ V  ↔  V ∝  1↓   V    , ∝/⍳0
```

The left argument is a character scalar or vector representing any primitive scalar dyadic function with a right-hand identity element (i.e., one of ∧ ∨ > ≥ = ≠ + - × ÷ * ⌈ ⌊). The right argument is a numeric vector.

The following are executable copies of *N*Δ and *P*Δ.

```
    ∇ Z←A NΔ B
[1]   Z←⍎'B',A,'¯1↓(',A,'/⍳0),B'
    ∇

    ∇ Z←A PΔ B
[1] ∇ Z←⍎'B',A,'1↓B,',A,'/⍳0'
    ∇
```

*N*Δ shifts vector *V* one position to the right; it drops the last element of *V* and catenates the identity element of ∝ to the beginning to preserve the length. This shifted vector is then compared to *V* using the function denoted as ∝.

*P*Δ shifts vector *V* one position to the left; it drops the first element of *V* and catenates the identity element of ∝ to the end to preserve the length. This shifted vector is then compared to *V* using the function denoted as ∝.

1.3.1 Identities

For *B* a Boolean vector and *V* a numeric vector,

```
(1)  B  ↔  ≠\'≠' NΔ B  ↔  '≠' NΔ ≠\B
     B  ↔  =\'=' NΔ B  ↔  '=' NΔ =\B
     V  ↔  +\'-' NΔ V  ↔  '-' NΔ +\V

(2)  '∝' NΔ V  ↔  ⌽'∝' PΔ ⌽V
     '∝' PΔ V  ↔  ⌽'∝' NΔ ⌽V

(3)  '∝' NΔ B  ↔  ~'ω' NΔ ~B
     '∝' PΔ B  ↔  ~'ω' PΔ ~B
```

where ∝ and ω may be selected from the following table:

∝	ω
=	≠
≠	=
≥	>
>	≥
∨	∧
∧	∨

1.3.2 Examples

Listed below are examples of *NΔ* and *PΔ* using Boolean functions.

```
------------------------------------------------|-------------------------------------------------
SET TO 1 THE FIRST 1 IN, AND THE FIRST 0 AFTER,|  SET TO 1 THE FIRST 0 BEFORE, AND THE LAST 1 IN,
EACH SERIES OF 1'S; ALL ELSE 0.                 |  EACH SERIES OF 1'S; ALL ELSE 0.
                                                |
        B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1   |          B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
'≠' NΔ B = 0 0 1 0 0 0 1 0 1 0 0 1 0 0 1 1 1   |  '≠' PΔ B = 0 1 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1
------------------------------------------------|-------------------------------------------------
SET TO 0 THE FIRST 0 IN, AND THE FIRST 1 AFTER,|  SET TO 0 THE FIRST 0 BEFORE, AND THE LAST 0 IN,
EACH SERIES OF 0'S; ALL ELSE 1.                 |  EACH SERIES OF 0'S; ALL ELSE 1.
                                                |
        B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1   |          B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
'=' NΔ B = 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0   |  '=' PΔ B = 1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 1
------------------------------------------------|-------------------------------------------------
SET TO 1 THE FIRST 1 IN EACH SERIES OF 1'S;     |  SET TO 1 THE LAST 1 IN EACH SERIES OF 1'S;
ALL ELSE 0.                                     |  ALL ELSE 0.
                                                |
        B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1   |          B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
'>' NΔ B = 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1   |  '>' PΔ B = 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1
------------------------------------------------|-------------------------------------------------
SET TO 0 THE FIRST 0 IN EACH SERIES OF 0'S;     |  SET TO 0 THE LAST 0 IN EACH SERIES OF 0'S;
ALL ELSE 1.                                     |  ALL ELSE 1.
                                                |
        B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1   |          B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
'≥' NΔ B = 0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1   |  '≥' PΔ B = 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1
------------------------------------------------|-------------------------------------------------
SET TO 1 THE FIRST 0 AFTER EACH SERIES OF 1'S;  |  SET TO 1 THE FIRST 0 BEFORE EACH SERIES OF 1'S;
ALL ELSE AS BEFORE.                             |  ALL ELSE AS BEFORE.
                                                |
        B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1   |          B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
'∨' NΔ B = 0 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1   |  '∨' PΔ B = 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 1
------------------------------------------------|-------------------------------------------------
SET TO 0 THE FIRST 1 AFTER EACH SERIES OF 0'S;  |  SET TO 0 THE FIRST 1 BEFORE EACH SERIES OF 0'S;
ALL ELSE AS BEFORE.                             |  ALL ELSE AS BEFORE.
                                                |
        B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1   |          B = 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1
'∧' NΔ B = 0 0 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0   |  '∧' PΔ B = 0 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 1
------------------------------------------------|-------------------------------------------------
```

CHAPTER 2

REFERENCE TABLES

This chapter contains practical tools that the programmer should find useful in his daily work. Included are truth tables, equivalence tables, identities, and algorithms involving Boolean values.

2.1 Composition Tables for the Relational and Logical Functions

<	0	1
0	0	1
1	0	0

≤	0	1
0	1	1
1	0	1

=	0	1
0	1	0
1	0	1

≥	0	1
0	1	0
1	1	1

>	0	1
0	0	0
1	1	0

≠	0	1
0	0	1
1	1	0

∨	0	1
0	0	1
1	1	1

∧	0	1
0	0	0
1	0	1

⍱	0	1
0	1	0
1	0	0

⍲	0	1
0	1	1
1	1	0

2.2 Equivalences of Non-Boolean Functions on Boolean Data

A×B	↔	A∧B
A⌊B	↔	A∧B
A⌈B	↔	A∨B
A*B	↔	A≥B
A|B	↔	A<B
A!B	↔	A≤B

2.3 Simplifications for Boolean Functions Using Tilde

A∝B	A∝~B	(~A)∝B	(~A)∝~B	~A∝B	~A∝~B	~(~A)∝B	~(~A)∝~B
A < B	A ⍱ B	A ∧ B	A > B	A ≥ B	A ∨ B	A ⍲ B	A ≤ B
A ≤ B	A ⍲ B	A ∨ B	A ≥ B	A > B	A ∧ B	A ⍱ B	A < B
A = B	A ≠ B	A ≠ B	A = B	A ≠ B	A = B	A = B	A ≠ B
A ≥ B	A ∨ B	A ⍲ B	A ≤ B	A < B	A ⍱ B	A ∧ B	A > B
A > B	A ∧ B	A ⍱ B	A < B	A ≤ B	A ⍲ B	A ∨ B	A ≥ B
A ≠ B	A = B	A = B	A ≠ B	A = B	A ≠ B	A ≠ B	A = B
A ∨ B	A ≥ B	A ≤ B	A ⍲ B	A ⍱ B	A < B	A > B	A ∧ B
A ∧ B	A > B	A < B	A ⍱ B	A ⍲ B	A ≤ B	A ≥ B	A ∨ B
A ⍱ B	A < B	A > B	A ∧ B	A ∨ B	A ≥ B	A ≤ B	A ⍲ B
A ⍲ B	A ≤ B	A ≥ B	A ∨ B	A ∧ B	A > B	A < B	A ⍱ B

2.4 Monadic Equivalents of Dyadic Boolean Functions with One Argument Constant

0 < A ↔ A	1 < A ↔ 0
0 ≤ A ↔ 1	1 ≤ A ↔ A
0 = A ↔ ~A	1 = A ↔ A
0 ≥ A ↔ ~A	1 ≥ A ↔ 1
0 > A ↔ 0	1 > A ↔ ~A
0 ≠ A ↔ A	1 ≠ A ↔ ~A
0 ∨ A ↔ A	1 ∨ A ↔ 1
0 ∧ A ↔ 0	1 ∧ A ↔ A
0 ⍱ A ↔ ~A	1 ⍱ A ↔ 0
0 ⍲ A ↔ 1	1 ⍲ A ↔ ~A

2.5 Equivalences for A⊚A⊚B in the Form of A⊚B

A<A<B	↔	A<B	A≠A<B	↔	A∨B
A<A≤B	↔	~A	A≠A≤B	↔	A⍲B
A<A=B	↔	A⍱B	A≠A=B	↔	~B
A<A≥B	↔	A⍱B	A≠A≥B	↔	A⍱B
A<A>B	↔	0	A≠A>B	↔	A∧B
A<A≠B	↔	A<B	A≠A≠B	↔	B
A<A∨B	↔	A<B	A≠A∨B	↔	A<B
A<A∧B	↔	0	A≠A∧B	↔	A>B
A<A⍱B	↔	A⍱B	A≠A⍱B	↔	A≥B
A<A⍲B	↔	~A	A≠A⍲B	↔	A≤B
A≤A<B	↔	~A	A∨A<B	↔	A∨B
A≤A≤B	↔	A≤B	A∨A≤B	↔	1
A≤A=B	↔	A≤B	A∨A=B	↔	A≥B
A≤A≥B	↔	1	A∨A≥B	↔	A≥B
A≤A>B	↔	A⍲B	A∨A>B	↔	A
A≤A≠B	↔	A⍲B	A∨A≠B	↔	A∨B
A≤A∨B	↔	1	A∨A∨B	↔	A∨B
A≤A∧B	↔	A≤B	A∨A∧B	↔	A
A≤A⍱B	↔	~A	A∨A⍱B	↔	A≥B
A≤A⍲B	↔	A⍲B	A∨A⍲B	↔	1
A=A<B	↔	A⍱B	A∧A<B	↔	0
A=A≤B	↔	A∧B	A∧A≤B	↔	A∧B
A=A=B	↔	B	A∧A=B	↔	A∧B
A=A≥B	↔	A∨B	A∧A≥B	↔	A
A=A>B	↔	A⍲B	A∧A>B	↔	A>B
A=A≠B	↔	~B	A∧A≠B	↔	A>B
A=A∨B	↔	A≥B	A∧A∨B	↔	A
A=A∧B	↔	A≤B	A∧A∧B	↔	A∧B
A=A⍱B	↔	A<B	A∧A⍱B	↔	0
A=A⍲B	↔	A>B	A∧A⍲B	↔	A>B
A≥A<B	↔	A≥B	A⍱A<B	↔	A⍱B
A≥A≤B	↔	A	A⍱A≤B	↔	0
A≥A=B	↔	A∨B	A⍱A=B	↔	A<B
A≥A≥B	↔	A∨B	A⍱A≥B	↔	A<B
A≥A>B	↔	1	A⍱A>B	↔	~A
A≥A≠B	↔	A≥B	A⍱A≠B	↔	A⍱B
A≥A∨B	↔	A≥B	A⍱A∨B	↔	A⍱B
A≥A∧B	↔	1	A⍱A∧B	↔	~A
A≥A⍱B	↔	A∨B	A⍱A⍱B	↔	A<B
A≥A⍲B	↔	A	A⍱A⍲B	↔	0
A>A<B	↔	A	A⍲A<B	↔	1
A>A≤B	↔	A>B	A⍲A≤B	↔	A⍲B
A>A=B	↔	A>B	A⍲A=B	↔	A⍲B
A>A≥B	↔	0	A⍲A≥B	↔	~A
A>A>B	↔	A∧B	A⍲A>B	↔	A≤B
A>A≠B	↔	A∧B	A⍲A≠B	↔	A≤B
A>A∨B	↔	0	A⍲A∨B	↔	~A
A>A∧B	↔	A>B	A⍲A∧B	↔	A⍲B
A>A⍱B	↔	A	A⍲A⍱B	↔	1
A>A⍲B	↔	A∧B	A⍲A⍲B	↔	A≤B

2.6 Distributive Identities for Boolean Functions

```
(A<B)<A<C ↔ A<B<C ↔ A⍱B≥C
(A<B)≤A<C ↔ A≥B>C ↔ A∨B≤C
(A<B)=A<C ↔ A≥B≠C ↔ A∨B=C
(A<B)≥A<C ↔ A≥B<C ↔ A∨B≥C
(A<B)>A<C ↔ A<B>C ↔ A⍱B≤C
(A<B)≠A<C ↔ A<B≠C ↔ A⍱B=C
(A<B)∨A<C ↔ A<B∨C ↔ A⍱B⍱C
(A<B)∧A<C ↔ A<B∧C ↔ A⍱B⍲C
(A<B)⍱A<C ↔ A≥B∨C ↔ A∨B⍱C
(A<B)⍲A<C ↔ A≥B∧C ↔ A∨B⍲C
(A≤B)<A≤C ↔ A>B≥C ↔ A∧B<C
(A≤B)≤A≤C ↔ A≤B≤C ↔ A⍲B>C
(A≤B)=A≤C ↔ A≤B=C ↔ A⍲B≠C
(A≤B)≥A≤C ↔ A≤B≥C ↔ A⍲B<C
(A≤B)>A≤C ↔ A>B≤C ↔ A∧B>C
(A≤B)≠A≤C ↔ A>B=C ↔ A∧B≠C
(A≤B)∨A≤C ↔ A≤B∨C ↔ A⍲B⍱C
(A≤B)∧A≤C ↔ A≤B∧C ↔ A⍲B⍲C
(A≤B)⍱A≤C ↔ A>B∨C ↔ A∧B⍱C
(A≤B)⍲A≤C ↔ A>B∧C ↔ A∧B⍲C
(A=B)<A=C ↔ No Identity
(A=B)≤A=C ↔ No Identity
(A=B)=A=C ↔ B=C
(A=B)≥A=C ↔ No Identity
(A=B)>A=C ↔ No Identity
(A=B)≠A=C ↔ B≠C
(A=B)∨A=C ↔ No Identity
(A=B)∧A=C ↔ No Identity
(A=B)⍱A=C ↔ No Identity
(A=B)⍲A=C ↔ No Identity
(A≥B)<A≥C ↔ A<B>C ↔ A⍱B≤C
(A≥B)≤A≥C ↔ A≥B<C ↔ A∨B≥C
(A≥B)=A≥C ↔ A≥B≠C ↔ A∨B=C
(A≥B)≥A≥C ↔ A≥B>C ↔ A∨B≤C
(A≥B)>A≥C ↔ A<B<C ↔ A⍱B≥C
(A≥B)≠A≥C ↔ A<B≠C ↔ A⍱B=C
(A≥B)∨A≥C ↔ A≥B∧C ↔ A∨B⍲C
(A≥B)∧A≥C ↔ A≥B∨C ↔ A∨B⍱C
(A≥B)⍱A≥C ↔ A<B∧C ↔ A⍱B⍲C
(A≥B)⍲A≥C ↔ A<B∨C ↔ A⍱B⍱C
(A>B)<A>C ↔ A>B≤C ↔ A∧B>C
(A>B)≤A>C ↔ A≤B≥C ↔ A⍲B<C
(A>B)=A>C ↔ A≤B=C ↔ A⍲B≠C
(A>B)≥A>C ↔ A≤B≤C ↔ A⍲B>C
(A>B)>A>C ↔ A>B≥C ↔ A∧B<C
(A>B)≠A>C ↔ A>B=C ↔ A∧B≠C
(A>B)∨A>C ↔ A>B∧C ↔ A∧B⍲C
(A>B)∧A>C ↔ A>B∨C ↔ A∧B⍱C
(A>B)⍱A>C ↔ A≤B∧C ↔ A⍲B⍲C
(A>B)⍲A>C ↔ A≤B∨C ↔ A⍲B⍱C
```

```
(A≠B)<A≠C ↔ No Identity
(A≠B)≤A≠C ↔ No Identity
(A≠B)=A≠C ↔ B=C
(A≠B)≥A≠C ↔ No Identity
(A≠B)>A≠C ↔ No Identity
(A≠B)≠A≠C ↔ B≠C
(A≠B)∨A≠C ↔ No Identity
(A≠B)∧A≠C ↔ No Identity
(A≠B)⍱A≠C ↔ No Identity
(A≠B)⍲A≠C ↔ No Identity
(A∨B)<A∨C ↔ A<B<C ↔ A⍱B≥C
(A∨B)≤A∨C ↔ A≥B>C ↔ A∨B≤C
(A∨B)=A∨C ↔ A≥B≠C ↔ A∨B=C
(A∨B)≥A∨C ↔ A≥B<C ↔ A∨B≥C
(A∨B)>A∨C ↔ A<B>C ↔ A⍱B≤C
(A∨B)≠A∨C ↔ A<B≠C ↔ A⍱B=C
(A∨B)∨A∨C ↔ A≥B⍱C ↔ A∨B∨C
(A∨B)∧A∨C ↔ A≥B⍲C ↔ A∨B∧C
(A∨B)⍱A∨C ↔ A<B⍱C ↔ A⍱B∨C
(A∨B)⍲A∨C ↔ A<B⍲C ↔ A⍱B∧C
(A∧B)<A∧C ↔ A>B≥C ↔ A∧B<C
(A∧B)≤A∧C ↔ A≤B≤C ↔ A⍲B>C
(A∧B)=A∧C ↔ A≤B=C ↔ A⍲B≠C
(A∧B)≥A∧C ↔ A≤B≥C ↔ A⍲B<C
(A∧B)>A∧C ↔ A>B≤C ↔ A∧B>C
(A∧B)≠A∧C ↔ A>B=C ↔ A∧B≠C
(A∧B)∨A∧C ↔ A>B⍱C ↔ A∧B∨C
(A∧B)∧A∧C ↔ A>B⍲C ↔ A∧B∧C
(A∧B)⍱A∧C ↔ A≤B⍱C ↔ A⍲B∨C
(A∧B)⍲A∧C ↔ A≤B⍲C ↔ A⍲B∧C
(A⍱B)<A⍱C ↔ A<B>C ↔ A⍱B≤C
(A⍱B)≤A⍱C ↔ A≥B<C ↔ A∨B≥C
(A⍱B)=A⍱C ↔ A≥B≠C ↔ A∨B=C
(A⍱B)≥A⍱C ↔ A≥B>C ↔ A∨B≤C
(A⍱B)>A⍱C ↔ A<B<C ↔ A⍱B≥C
(A⍱B)≠A⍱C ↔ A<B≠C ↔ A⍱B=C
(A⍱B)∨A⍱C ↔ A<B⍲C ↔ A⍱B∧C
(A⍱B)∧A⍱C ↔ A<B⍱C ↔ A⍱B∨C
(A⍱B)⍱A⍱C ↔ A≥B⍲C ↔ A∨B∧C
(A⍱B)⍲A⍱C ↔ A≥B⍱C ↔ A∨B∨C
(A⍲B)<A⍲C ↔ A>B≤C ↔ A∧B>C
(A⍲B)≤A⍲C ↔ A≤B≥C ↔ A⍲B<C
(A⍲B)=A⍲C ↔ A≤B=C ↔ A⍲B≠C
(A⍲B)≥A⍲C ↔ A≤B≤C ↔ A⍲B>C
(A⍲B)>A⍲C ↔ A>B≥C ↔ A∧B<C
(A⍲B)≠A⍲C ↔ A>B=C ↔ A∧B≠C
(A⍲B)∨A⍲C ↔ A≤B⍲C ↔ A⍲B∧C
(A⍲B)∧A⍲C ↔ A≤B⍱C ↔ A⍲B∨C
(A⍲B)⍱A⍲C ↔ A>B⍲C ↔ A∧B∧C
(A⍲B)⍲A⍲C ↔ A>B⍱C ↔ A∧B∨C
```

2.7 Inverse Distributive Identities for Boolean Functions

```
A<B<C  ↔  (A<B)<A<C  ↔  (A∨B)<A∨C  ↔  (A≥B)>A≥C  ↔  (A⍱B)>A⍱C
A≥B<C  ↔  (A≥B)≤A≥C  ↔  (A⍱B)≤A⍱C  ↔  (A<B)≥A<C  ↔  (A∨B)≥A∨C
A∧B<C  ↔  (A≤B)<A≤C  ↔  (A∧B)<A∧C  ↔  (A>B)>A>C  ↔  (A⍲B)>A⍲C
A⍲B<C  ↔  (A>B)≤A>C  ↔  (A⍲B)≤A⍲C  ↔  (A≤B)≥A≤C  ↔  (A∧B)≥A∧C
A≤B≤C  ↔  (A≤B)≤A≤C  ↔  (A∧B)≤A∧C  ↔  (A>B)≥A>C  ↔  (A⍲B)≥A⍲C
A>B≤C  ↔  (A>B)<A>C  ↔  (A⍲B)<A⍲C  ↔  (A≤B)>A≤C  ↔  (A∧B)>A∧C
A∨B≤C  ↔  (A<B)≤A<C  ↔  (A∨B)≤A∨C  ↔  (A≥B)≥A≥C  ↔  (A⍱B)≥A⍱C
A⍱B≤C  ↔  (A≥B)<A≥C  ↔  (A⍱B)<A⍱C  ↔  (A<B)>A<C  ↔  (A∨B)>A∨C
A≤B=C  ↔  (A≤B)=A≤C  ↔  (A>B)=A>C  ↔  (A∧B)=A∧C  ↔  (A⍲B)=A⍲C
A>B=C  ↔  (A≤B)≠A≤C  ↔  (A>B)≠A>C  ↔  (A∧B)≠A∧C  ↔  (A⍲B)≠A⍲C
A∨B=C  ↔  (A<B)=A<C  ↔  (A≥B)=A≥C  ↔  (A∨B)=A∨C  ↔  (A⍱B)=A⍱C
A⍱B=C  ↔  (A<B)≠A<C  ↔  (A≥B)≠A≥C  ↔  (A∨B)≠A∨C  ↔  (A⍱B)≠A⍱C
A≤B≥C  ↔  (A>B)≤A>C  ↔  (A⍲B)≤A⍲C  ↔  (A≤B)≥A≤C  ↔  (A∧B)≥A∧C
A>B≥C  ↔  (A≤B)<A≤C  ↔  (A∧B)<A∧C  ↔  (A>B)>A>C  ↔  (A⍲B)>A⍲C
A∨B≥C  ↔  (A≥B)≤A≥C  ↔  (A⍱B)≤A⍱C  ↔  (A<B)≥A<C  ↔  (A∨B)≥A∨C
A⍱B≥C  ↔  (A<B)<A<C  ↔  (A∨B)<A∨C  ↔  (A≥B)>A≥C  ↔  (A⍱B)>A⍱C
A<B>C  ↔  (A≥B)<A≥C  ↔  (A⍱B)<A⍱C  ↔  (A<B)>A<C  ↔  (A∨B)>A∨C
A≥B>C  ↔  (A<B)≤A<C  ↔  (A∨B)≤A∨C  ↔  (A≥B)≥A≥C  ↔  (A⍱B)≥A⍱C
A∧B>C  ↔  (A>B)<A>C  ↔  (A⍲B)<A⍲C  ↔  (A≤B)>A≤C  ↔  (A∧B)>A∧C
A⍲B>C  ↔  (A≤B)≤A≤C  ↔  (A∧B)≤A∧C  ↔  (A>B)≥A>C  ↔  (A⍲B)≥A⍲C
A<B≠C  ↔  (A<B)≠A<C  ↔  (A≥B)≠A≥C  ↔  (A∨B)≠A∨C  ↔  (A⍱B)≠A⍱C
A≥B≠C  ↔  (A<B)=A<C  ↔  (A≥B)=A≥C  ↔  (A∨B)=A∨C  ↔  (A⍱B)=A⍱C
A∧B≠C  ↔  (A≤B)≠A≤C  ↔  (A>B)≠A>C  ↔  (A∧B)≠A∧C  ↔  (A⍲B)≠A⍲C
A⍲B≠C  ↔  (A≤B)=A≤C  ↔  (A>B)=A>C  ↔  (A∧B)=A∧C  ↔  (A⍲B)=A⍲C
A<B∨C  ↔  (A<B)∨A<C  ↔  (A≥B)⍲A≥C
A≤B∨C  ↔  (A≤B)∨A≤C  ↔  (A>B)⍲A>C
A≥B∨C  ↔  (A≥B)∧A≥C  ↔  (A<B)⍱A<C
A>B∨C  ↔  (A>B)∧A>C  ↔  (A≤B)⍱A≤C
A∨B∨C  ↔  (A∨B)∨A∨C  ↔  (A⍱B)⍲A⍱C
A∧B∨C  ↔  (A∧B)∨A∧C  ↔  (A⍲B)⍲A⍲C
A⍱B∨C  ↔  (A⍱B)∧A⍱C  ↔  (A∨B)⍱A∨C
A⍲B∨C  ↔  (A⍲B)∧A⍲C  ↔  (A∧B)⍱A∧C
A<B∧C  ↔  (A<B)∧A<C  ↔  (A≥B)⍱A≥C
A≤B∧C  ↔  (A≤B)∧A≤C  ↔  (A>B)⍱A>C
A≥B∧C  ↔  (A≥B)∨A≥C  ↔  (A<B)⍲A<C
A>B∧C  ↔  (A>B)∨A>C  ↔  (A≤B)⍲A≤C
A∨B∧C  ↔  (A∨B)∧A∨C  ↔  (A⍱B)⍱A⍱C
A∧B∧C  ↔  (A∧B)∧A∧C  ↔  (A⍲B)⍱A⍲C
A⍱B∧C  ↔  (A⍱B)∨A⍱C  ↔  (A∨B)⍲A∨C
A⍲B∧C  ↔  (A⍲B)∨A⍲C  ↔  (A∧B)⍲A∧C
A<B⍱C  ↔  (A⍱B)∧A⍱C  ↔  (A∨B)⍱A∨C
A≤B⍱C  ↔  (A⍲B)∧A⍲C  ↔  (A∧B)⍱A∧C
A≥B⍱C  ↔  (A∨B)∨A∨C  ↔  (A⍱B)⍲A⍱C
A>B⍱C  ↔  (A∧B)∨A∧C  ↔  (A⍲B)⍲A⍲C
A∨B⍱C  ↔  (A≥B)∧A≥C  ↔  (A<B)⍱A<C
A∧B⍱C  ↔  (A>B)∧A>C  ↔  (A≤B)⍱A≤C
A⍱B⍱C  ↔  (A<B)∨A<C  ↔  (A≥B)⍲A≥C
A⍲B⍱C  ↔  (A≤B)∨A≤C  ↔  (A>B)⍲A>C
A<B⍲C  ↔  (A⍱B)∨A⍱C  ↔  (A∨B)⍲A∨C
A≤B⍲C  ↔  (A⍲B)∨A⍲C  ↔  (A∧B)⍲A∧C
A≥B⍲C  ↔  (A∨B)∧A∨C  ↔  (A⍱B)⍱A⍱C
A>B⍲C  ↔  (A∧B)∧A∧C  ↔  (A⍲B)⍱A⍲C
A∨B⍲C  ↔  (A≥B)∨A≥C  ↔  (A<B)⍲A<C
A∧B⍲C  ↔  (A>B)∨A>C  ↔  (A≤B)⍲A≤C
A⍱B⍲C  ↔  (A<B)∧A<C  ↔  (A≥B)⍱A≥C
A⍲B⍲C  ↔  (A≤B)∧A≤C  ↔  (A>B)⍱A>C
```

2.8 Interpretations of Reductions on a Boolean Vector Where the Result Is Zero Or One

Let *B* be any Boolean vector.

Reduction	Interpretation
∧/*B*	0 if 0∊*B*; 1 otherwise.
∨/*B*	1 if 1∊*B*; 0 otherwise.
⍲/*B*	>/*B* if *B* is of the form *N*≠⍳*N* or *N*⍴1; otherwise ~2|+/∧\1=*B* (the reverse parity of the number of leading 1's in *B*).
⍱/*B*	≥/*B* if *B* is of the form *N*=⍳*N* or *N*⍴0; otherwise 2|+/∧\0=*B* (the parity of the number of leading 0's in *B*).
</*B*	1 if *B* is of the form *N*=⍳*N*; 0 otherwise.
≤/*B*	0 if *B* is of the form *N*≠⍳*N*; 1 otherwise.
=/*B*	~2|+/0=*B* (the reverse parity of the number of 0's in *B*).
≥/*B*	~2|+/∧\0=*B* (the reverse parity of the number of leading 0's in *B*).
>/*B*	2|+/∧\1=*B* (the parity of the number of leading 1's in *B*).
≠/*B*	2|+/1=*B* (the parity of the number of 1's in *B*).
×/*B*	0 if 0∊*B*; 1 otherwise.
⌈/*B*	1 if 1∊*B*; 0 otherwise.
⌊/*B*	0 if 0∊*B*; 1 otherwise.
*/*B*	~2|+/∧\0=*B* (the reverse parity of the number of leading 0's in *B*).
|/*B*	1 if *B* is of the form *N*=⍳*N*; 0 otherwise.
!/*B*	0 if *B* is of the form *N*≠⍳*N*; 1 otherwise.

2.9 Similarities between Reduction and Scan of Boolean Vectors

The table below illustrates notational similarities between APL expressions for reduction and scan of Boolean vectors. The functions included are all the primitive scalar dyadic relational and logical functions. Origin dependent expressions should be evaluated in origin 1.

Simpler statements exist for some of the expressions. For instance,

```
≥/B ↔ ~2|+/∧\~B ↔  2|B⍳1
>/B ↔  2|+/∧\B  ↔ ~2|B⍳0
```

The form using ∧\ was not chosen to point out the similarities with the expressions illustrating other reductions and scan. Also, ∧\ is one of the objects being illustrated and should not be used to define other expressions. The other equivalent form was not chosen since it has no simple extension to an expression for a scan using that function.

```
     Reduction                               Scan

∧/B  ↔  (B⍳0)>⍴B                   ∧\B  ↔  (B⍳0)>⍳⍴B

∨/B  ↔  (B⍳1)≤⍴B                   ∨\B  ↔  (B⍳1)≤⍳⍴B

⍲/B  ↔  ((B⍳0)<⍴B)=~2|             ⍲\B  ↔  ((B⍳0)<⍳⍴B)=~2|
        +/(B⍳0)>⍳⍴B                         +\(B⍳0)>⍳⍴B

⍱/B  ↔  ((B⍳1)<⍴B)=2|              ⍱\B  ↔  ((B⍳1)<⍳⍴B)=2|
        +/(B⍳1)>⍳⍴B                         +\(B⍳1)>⍳⍴B

</B  ↔  (B⍳1)=⍴B                   <\B  ↔  (B⍳1)=⍳⍴B

≤/B  ↔  (B⍳0)≠⍴B                   ≤\B  ↔  (B⍳0)≠⍳⍴B

=/B  ↔  ~2|+/~B                    =\B  ↔  ~2|+\~B

≥/B  ↔  ~2|+/(B⍳1)>⍳⍴B             ≥\B  ↔  ~2|+\(B⍳1)>⍳⍴B

>/B  ↔  2|+/(B⍳0)>⍳⍴B              >\B  ↔  2|+\(B⍳0)>⍳⍴B

≠/B  ↔  2|+/B                      ≠\B  ↔  2|+\B
```

2.10 Fast Algorithms for Implementing Relational and Logical Scans Other Than ≠\ and =\

```
Note:  A̲ ↔ (⍴B)-A ⋄ ⎕IO ← 1

R←∨\B     A←¯1+B⍳1 ⋄ R←(A⍴0 0),A̲⍴1
R←∧\B     A←¯1+B⍳0 ⋄ R←(A⍴1 1),A̲⍴0
R←≥\B     A←¯1+B⍳1 ⋄ R←(A⍴0 1),A̲⍴~2|A
R←>\B     A←¯1+B⍳0 ⋄ R←(A⍴1 0),A̲⍴ 2|A
R←<\B     A←¯1+B⍳1 ⋄ R←(A⍴0 0),A̲⍴0     ⋄ IF A≠⍴B THEN R[A+1]←1
R←≤\B     A←¯1+B⍳0 ⋄ R←(A⍴1 1),A̲⍴1     ⋄ IF A≠⍴B THEN R[A+1]←0
R←⍱\B     A←¯1+B⍳1 ⋄ R←(A⍴0 1),A̲⍴ 2|A ⋄ IF A≠⍴B THEN R[A+1]←~2|A
R←⍲\B     A←¯1+B⍳0 ⋄ R←(A⍴1 0),A̲⍴~2|A ⋄ IF A≠⍴B THEN R[A+1]← 2|A
```

2.11 Eight Different Definitions of Scan, with Examples

```
   R ← ∝\V                        +\⍳5                 -\⍳5
R[I] ← ∝/     I ↑V    |  1   3   6  10  15  |  1  ¯1  2  ¯2  3
R[I] ← ∝/ (I-1)↓V     | 15  14  12   9   5  |  3  ¯2  4  ¯1  5
R[I] ← ∝/  (-I)↑V     |  5   9  12  14  15  |  5  ¯1  4  ¯2  3
R[I] ← ∝/ (1-I)↓V     | 15  10   6   3   1  |  3  ¯2  2  ¯1  1
R[I] ← ∝/⌽    I ↑V    |  1   3   6  10  15  |  1   1  2   2  3
R[I] ← ∝/⌽(I-1)↓V     | 15  14  12   9   5  |  3   2  4   1  5
R[I] ← ∝/⌽ (-I)↑V     |  5   9  12  14  15  |  5   1  4   2  3
R[I] ← ∝/⌽(1-I)↓V     | 15  10   6   3   1  |  3   2  2   1  1
```

2.12 Identities Involving Scan

```
<\B ↔ ~≤\~B
≤\B ↔ ~<\~B
≥\B ↔ ~>\~B ↔ =\∨\B
>\B ↔ ~≥\~B ↔ ≠\∧\B
=\B ↔ ~≠\~B
≠\B ↔ ~=\~B
∨\B ↔ ~∧\~B
∧\B ↔ ~∨\~B
⍱\B ↔ ~⍲\~B ↔ (≥\B)=(∨\B)≤<\B
⍲\B ↔ ~⍱\~B ↔ (>\B)≠(∧\B)<≤\B
```

2.13 Results of Operations Using Boolean Vectors

The following example illustrates the use of exponentiation by a logical vector to change 0's to 1's in an array.

```
      A← 6 7 0 5 3 0 5 0 0 7
      B← 3 4 0 2 3 0 4 5 8 2

      B*B≠0
3 4 1 2 3 1 4 5 8 2

      A÷B ⋄ A÷B*B≠0
2 1.75 1 2.5 1 1 1.25 0 0 3.5
2 1.75 0 2.5 1 0 1.25 0 0 3.5
```

Here exponentiation by a logical vector is used to change 0's to 1's because 0÷1 produces the desired 0. *A*÷*B*+*B*=0 could also have been used in this case.

The table below shows the most useful manipulations and what they produce when *BOOLEAN* is assigned 0 or 1.

Operation	Result When *BOOLEAN*←0	Result When *BOOLEAN*←1
A×*BOOLEAN*	0	*A*
*A***BOOLEAN*	1	*A*
BOOLEAN!*A*	1	*A*
BOOLEAN|*A*	*A*	0 (only for integer *A*)
A×~*BOOLEAN*	*A*	0
*A**~*BOOLEAN*	*A*	1

2.14 Altering the Order of Arguments Using Conditional Arrays

Often when dealing with arrays, we wish to process part of the array while leaving the rest untouched. For instance, add 3 percent to only those bills over 30 days old:

```
      BILLS←BILLS×1.03*DAYSOLD>30
```

Other times, we may wish to alter the order of the arguments of the computation itself based on a conditional array (e.g., *A*-*B* or *B*-*A*). For functions with an inverse, we may wish to conditionally alter the function used in a computation (e.g., *A*+*B* or *A*-*B*). If a function has an inverse, we can use ¯1 and one of × * | ! on the conditional array to give us the proper answer. Or, as shown above, we can use the conditional array to control whether the operation is performed.

In the table below, C is the conditional array. C controls the function used in the first two lines, the order of the function's arguments in the next two lines, and whether or not the function is performed in the last three lines.

If C=0 Return	If C=1 Return	By Using This Expression
$A+B$	$A-B$	$A+B\ \times^{-}1\star C$
$A\times B$	$A\div B$	$A\times B\ \star^{-}1\star C$
$A-B$	$B-A$	$(A-B)\times^{-}1\star C$
$A\div B$	$B\div A$	$(A\div B)\star^{-}1\star C$
A	$A+B$	$A+B\times C$
A	$A\times B$	$A\times B\star C$
A	$A\star B$	$A\star B\star C$

CHAPTER 3

SELECTED FUNCTIONS THAT ILLUSTRATE BOOLEAN TECHNIQUES

This chapter comprises a number of functions that illustrate the practical use of the techniques described thus far in the publication.

```
    ∇ Z←A BINARYADDER B;C;D;E
[1]   ⍝ CALCULATES THE ARITHMETIC SUM (WITH CARRY PROPAGATION) OF
[2]   ⍝ TWO BINARY VECTORS.
[3]   ⍝ GLOBALS: F - NΔ, PΔ.
[4]   ⍝ ORIGIN FREE.
[5]   ⍝ WRITTEN BY ROBERT A. SMITH, STSC, 20 JULY 74.
[6]    C←-1+(⍴A)⌈⍴B ⋄ A←C↑A ⋄ B←C↑B
[7]    D←A∧B ⋄ E←A=B ⋄ C←'≠' NΔ~E
[8]    Z←(E\1↓(E/D),0)≠≠\C\'∧' PΔ~C/D
    ∇
```

```
    ∇ Z←CMB X
[1]   ⍝ COMPRESS MULTIPLE BLANKS
[2]   ⍝ ORIGIN FREE
[3]   ⍝ GLOBALS:  (NONE)
[4]   ⍝ WRITTEN BY ROBERT A. SMITH, 17 JUL 78.
[5]    X←' ',X,' ' ⋄ Z←1↓¯1↓(~X ⎕SS '  ')/X
    ∇
```

```
    ∇ Z←DLB X;A;B
[1]   ⍝ DELETE LEADING BLANKS
[2]   ⍝ WORKS ON VECTORS, DELETING LEADING BLANKS BETWEEN ⎕TCNL'S,
[3]   ⍝ OR ON MATRICES (RETURNING A VECTOR) DELETING LEADING
[4]   ⍝ BLANKS IN EACH ROW.
[5]   ⍝ ORIGIN FREE.
[6]   ⍝ GLOBALS: F - NΔ.
[7]   ⍝ WRITTEN BY ROBERT A. SMITH, STSC, 28 JAN 74.
[8]    X←,⎕TCNL,X ⋄ A←X=⎕TCNL ⋄ Z←X=' ' ⋄ B←A≥Z
[9]    Z←1↓(A∨≠\B\'≠' NΔ~B/A∨Z)/X
    ∇
    ∇ Z←DTB X;A
[1]   ⍝ DELETE TRAILING BLANKS
[2]   ⍝ WORKS ON VECTORS DELETING TRAILING BLANKS BETWEEN ⎕TCNL'S,
[3]   ⍝ OR ON MATRICES (RETURNING A VECTOR) DELETING TRAILING
[4]   ⍝ BLANKS IN EACH ROW.
[5]   ⍝ ORIGIN FREE.
[6]   ⍝ GLOBALS: F - N∆, P∆.
[7]   ⍝ WRITTEN BY ROBERT A. SMITH, STSC, 28 JAN 74.
[8]    X←¯1↓,X,⎕TCNL ◇ A←'≠' N∆ X=' '
[9]    Z←(~≠\A\'≠' P∆ A/X=⎕TCNL)/X
    ∇


    ∇ Z←∆FMTNZ M;A;BL;PT;RM
[1]   ⍝ SUPPRESSES TRAILING 0'S IN <M> UP TO AND INCLUDING
[2]   ⍝ THE DECIMAL POINT.
[3]   ⍝ GLOBALS: F - N∆, P∆.
[4]   ⍝ ORIGIN FREE.
[5]   ⍝ WRITTEN BY ROBERT A. SMITH, STSC, 29 MAR 74.
[6]    RM← 0 1 +⍴M ◇ M←,M,' '
[7]    BL←M=' ' ◇ PT←M='.'
[8]    A←PT∨'>' N∆ BL
[9]    A←'≠' N∆(M∊'0.')∧≠\A\'≠' N∆ A/PT
[10]   A←~≠\A\'≠' P∆ A/BL
[11]   Z← 0 ¯1 ↓RM⍴A\A/M
    ∇


    ∇ Z←A STREPL B;C;R;⎕IO
[1]   ⍝ REPLACES IN <B> ALL OCCURRENCES OF <1↑A> BY <1↓A>.
[2]   ⍝ GLOBALS: F - N∆.
[3]   ⍝ ORIGIN 0 DEPENDENT.
[4]   ⍝ WRITTEN BY ROBERT A. SMITH, STSC, 10 NOV 74.
[5]    ⎕IO←0 ◇ A←,A ◇ C←B=1⍴A ◇ Z←C/⍳⍴B ◇ R←((⍴B)+(⍴Z)×¯2+⍴A)⍴1
[6]    R[(Z+(¯2+⍴A)×⍳⍴Z)∘.+⍳¯2+⍴A]←0 ◇ Z←R\B
[7]    R←('∧' N∆ R)≤R\C ◇ Z[R/⍳⍴R]←(+/R)⍴1↓A
    ∇


    ∇ R←A SUMSCANBOOLCOMP B;C;⎕IO
[1]   ⍝ COMPUTES  A/+\B  FOR A LONG BOOLEAN VECTOR <B>.
[2]   ⍝ AVOIDS  WS FULL  PROBLEMS BY NOT CREATING  +\B .
[3]   ⍝ GLOBALS: (NONE)
[4]   ⍝ ORIGIN 1 DEPENDENT.
[5]   ⍝ WRITTEN BY ROBERT A. SMITH, STSC, 9 FEB 74,
[6]   ⍝  SPEEDUP BY GERRY BAMBERGER, STSC, 8 JAN 78.
[7]    ⎕IO←1 ◇ C←(B∨A)/A
[8]    R←(C/⍳⍴C)-+\~A/B
    ∇
    ∇ R←B SUMSCANBOOLSUB A;C;D
[1]   ⍝ COMPUTES  (+\B)[A]  FOR A LONG BOOLEAN VECTOR <B>.
[2]   ⍝ AVOIDS  WS FULL  PROBLEMS BY NOT CREATING  +\B .
[3]   ⍝ ASSUMPTIONS:  <A> CONTAINS NO DUPLICATES, AND
[4]   ⍝               IS IN ASCENDING ORDER.
[5]   ⍝ GLOBALS: (NONE)
[6]   ⍝ ORIGIN SENSITIVE.
[7]   ⍝ WRITTEN BY ROBERT A. SMITH, STSC, 9 FEB 74,
[8]   ⍝  SPEEDUP BY GERRY BAMBERGER, STSC, 8 JAN 78.
[9]    C←(⍴B)⍴0 ⋄ C[A]←1
[10]   D←(B∨C)/B≤C ⋄ D←(D/⍳⍴D)-~⎕IO
[11]   R←D-+\~C/B
[12] ⍝ (FROM ROY SYKES, STSC, 7 JULY 75)
[13] ⍝ IF <A> CONTAINS NO DUPLICATES, BUT IS NOT IN ASCENDING ORDER:
[14] ⍝ [12]  R←R[⍋⍋A]  ∇
[15] ⍝ IF <A> CONTAINS DUPLICATES, BUT IS IN ASCENDING ORDER:
[16] ⍝ [12]  R←R[+\A≠¯1↓(1↑(~⎕IO)↑A),A]  ∇
[17] ⍝ IF <A> CONTAINS DUPLICATES, AND IS NOT IN ASCENDING ORDER:
[18] ⍝ [12]  R←R[(C/⍳⍴C)⍳A]  ∇
    ∇
```