Reference Manual

APL★PLUS† Nested Arrays System

by
Carl M. Cheney

STSC, Inc.
7316 Wisconsin Avenue
Bethesda, Maryland 20014
301-657-8220

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

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

PREFACE

This reference manual describes STSC's Nested Arrays System, an experimental research implementation of *APL* developed to explore extensions to the *APL* language. Among other proposals this system offers nested arrays, arrays in which each position can contain an array of any rank. Nested arrays can conveniently represent data which is difficult to represent in conventional *APL* data structures.

The purpose of this research is to develop extensions to *APL* which will make *APL* programmers more productive. The concepts in this manual can also be used to model processes which will later be rewritten in *APL* or another language. Explore and evaluate these concepts and use them to extend your thought tools.

The process of implementing the Nested Arrays System has focused on producing a fully functional system rather than an efficient system. Operating efficiency will be addressed at a later time.

STSC currently plans to offer the research implementation until January 1982. Selected features from this system will be provided in STSC's production *APL*PLUS* VM System.

Among the many people whose ideas are included in this system are J. A. Brown, Kenneth E. Iverson, Michael A. Jenkins, E. E. McDonnell, Trenchard More, and Bob Smith. The Nested Arrays System was implemented by Bob Smith.

I would like to thank those who spent many hours reviewing earlier versions of this manual: Tom Edelson, Paul Jackson, Bob Korsan, Gene Mannacio, Bob McGhee, Don Scheer, and Roy Sykes, Jr. I am especially grateful to John Gilmore, Graham Prindle, Laurie Russell, Barbara Schick, and Bob Smith who contributed many useful items.

Carl Cheney
San Francisco
March 1981

CONTENTS

CHAPTER 1

INTRODUCTION

This manual describes STSC's Nested Arrays System, a research implementation of *APL* designed to explore extensions to the *APL* language. Both the system and the manual are intended for use by experienced *APL* programmers.

The manual is divided into six chapters. Chapter 1 introduces the major concepts of the system in an informal tutorial manner. In addition, Section 1.4 describes how to gain access to the system, and Section 1.5 describes the syntax conventions used in the remainder of the manual.

Chapters 2 through 5 contain detailed reference material for the functions, operators, and features provided in the system. Chapter 6 explains how conventional *APL* functions and operators work on nested arrays. Following Chapter 6 is a glossary and a list of references.

1.1 What Are Nested Arrays?

In conventional *APL*, each position of an array can contain only a simple scalar. In a nested array, each position can contain an array of any rank. The term for the contents of a position of an array is item. For example,

```
      A
(1 2 3) (2 4 6) (3 6 9) (4 8 12)
      ⍴A
4
      X
( 1 2 3  (  7  8  9
  4 5 6)   10 11 12)
      ⍴X
2
```

The variable *A* is a four-item vector, each of whose items is a three-item vector. The parentheses indicate where items begin and end. The variable *X* contains a two-item vector, each of whose items is a two-row, three-column matrix.

These arrays are said to be nested or nonsimple. A simple array is one in which there is no nesting; that is, each position contains an unnested scalar. The simple function, represented by the symbol ≡ used monadically, returns 1 if its argument is simple and 0 otherwise.

```
      ≡A
0
      ≡⍳5
1
```

In addition to the simple function, the Nested Arrays System provides other tools for dealing with nested arrays. Sections 1.1.1 through 1.1.5 introduce five of these tools: four new functions (enclose, disclose, pick, and type) and a new feature (strand notation). These tools are covered in more detail in Chapters 3 and 5.

1.1.1 Enclose

The enclose function, represented by the symbol ⊂ used monadically, creates a nested scalar out of any array which is not a simple scalar. A simple scalar is returned unchanged.

```
      B←⊂1 2 3
      B
(1 2 3)
      ⍴⍴B
0
      ≡B
0
      ⊂⊂1 2
((1 2))
      ⊂4
4
      ≡⊂4
1
      C←(⊂'WHAT'),(⊂'HATH'),(⊂'GOD'),⊂'WROUGHT'
      ⍴C
4
      C
(WHAT) (HATH) (GOD) (WROUGHT)
```

1.1.2 Disclose

The disclose function (monadic ⊃) undoes the work of the enclose function. It removes the outermost nesting from a nested scalar and returns the enclosed item.

```
      ⊃B
1 2 3
      ⍴⊃B
3
      ≡B
0
      ≡⊃B
1
      ⊃⊂'ABC'
ABC
      ⊃⊂5
5
      ⊃5
5
      A[2]
(2 4 6)
      ⊃A[2]
2 4 6
```

When disclose is applied to an array that has more than one item, it returns only the first item; in fact, used in this context another name for disclose is first.

```
      ⊃2 3 4
2
      ⊃A
1 2 3
```

1.1.3 Pick

The pick function (dyadic ⊃) extracts a portion of an array. Each successive item of the left argument selects a successively deeper-nested item of the right argument.

```
      A
(1 2 3) (2 4 6) (3 6 9) (4 8 12)
      3⊃A
3 6 9
      3 2⊃A
6
```

When picking from an array that is not a vector, the corresponding item of the left argument must have a shape equal to the rank of the array. For instance, to select an item from a matrix, the item in the left argument must be a two-item vector.

```
      X
( 1 2 3  (  7  8  9
  4 5 6)   10 11 12)
      1⊃X
 1 2 3
 4 5 6
      (1,⊂2 3)⊃X
6
```

1.1.4 Strand Notation

Strand notation is a means of representing both simple and nonsimple vectors. It is based on the familiar notation for entering a constant numeric vector (for instance, 4 8 12). Strand notation takes effect when two or more of the following constructs appear in an expression without an intervening function:

- A numeric value; for example, 256.
- A character value; for example, '*SLITHY TOVES*'.
- An expression; for example, (2*0.5).

When two or more such constructs appear with no function separating them, each becomes an item.

```
      (2×3) (3+4)
6 7
      ρE←'PICK' 'A' 'CARD'
3
      E
(PICK) A (CARD)
      E←2 ◇ F←3 ◇ G←5
      E F G
2 3 5
      E F G×2
4 6 10
      (2 3) (4 5 6) (7 8 9 10)
(2 3) (4 5 6) (7 8 9 10)
```

Notice that when an item evaluates to a simple scalar, it remains simple.

1.1.5 Type

The type function (monadic ⊤) returns a result with the same structure as its argument. However, numeric values are replaced with zeros and character values are replaced with blanks.

```
      ⊤⍳4
0 0 0 0
      ' '=⊤'PONDER'
1 1 1 1 1 1
      ⊤A
(0 0 0) (0 0 0) (0 0 0) (0 0 0)
      ⊤C
(    ) (    ) (   ) (        )
```

1.2 How Existing Functions and Operators Work on Nested Arrays

Scalar functions are pervasive; that is, they apply through all levels of nesting.

```
      A
(1 2 3) (2 4 6) (3 6 9) (4 8 12)
      A+A
(2 4 6) (4 8 12) (6 12 18) (8 16 24)
      A÷1 2 3 4
(1 2 3) (1 2 3) (1 2 3) (1 2 3)
      A-1
(0 1 2) (1 3 5) (2 5 8) (3 7 11)
      1 2 3 4×⊂1 2 3
(1 2 3) (2 4 6) (3 6 9) (4 8 12)
```

Structural functions apply to the outermost level of a nested array.

```
      2 2⍴A
 (1 2 3)  (2 4 6)
 (3 6 9) (4 8 12)
      ⌽A
(4 8 12) (3 6 9) (2 4 6) (1 2 3)
```

Operators can also be applied to nested arrays. In the following example, the addition function has been applied to the items of *A* in the pattern specified by the reduction and scan operators.

```
      +/A
(10 20 30)
      +\A
(1 2 3) (3 6 9) (6 12 18) (10 20 30)
```

Two new features in the Nested Arrays System allow operators to call user-defined functions and all primitive functions.

When user-defined functions are called by operators, powerful yet easily understood array-oriented control structures are provided for function calls. This new feature can also be used to explore the behavior of an operator as in the following example.

```
    ∇ Z←L MINUS R
[1]    Z←L-R
[2]    ,'I2,<->,I2,<=>,I2' ⎕FMT(L;R;Z)
    ∇
       5 MINUS 3
 5- 3= 2
2
       -/⍳5
3
       MINUS/⍳5
 4- 5=¯1
 3-¯1= 4
 2- 4=¯2
 1-¯2= 3
3
```

By allowing operators to call any primitive function, the restriction of operators to scalar functions is eliminated.

```
       ,/'MARES' 'EAT' 'OATS'
(MARESEATOATS)
       ⊃,/'MARES' 'EAT' 'OATS'
MARESEATOATS
       1 2 3 ∘., 4 5
 (1 4) (1 5)
 (2 4) (2 5)
 (3 4) (3 5)
```

Operators have restrictions about the valence (the number of arguments) of the functions which are their arguments; for instance, reduction requires that its left argument be a dyadic function.

1.3 New Operators

Many new operators are included in the Nested Arrays System; only a few are described in the following sections. New operators are covered in detail in Chapter 4.

1.3.1 Dyadic Reduction

The left argument of dyadic reduction indicates the width of the window on which reduction is performed. For example, a left argument of 2 indicates that reduction should be performed on consecutive pairs of items in the right argument.

```
       2 +/ 1 1 1 1
2 2 2
       3 +/ 1 1 1 1
3 3
       4 +/ 1 1 1 1
4
       2 </ 1 2 3 4
1 1 1
       ∧/ 2</ 1 2 2 3    Returns 1 if argument is strictly ascending.
0
       2 -/ 4 3 2 1
1 1 1
```

If the left argument is negative, each window is reversed before being reduced.

```
      ¯2 -/ 4 3 2 1
¯1 ¯1 ¯1
```

1.3.2 Dyadic Scan

Dyadic scan is similar to dyadic reduction. With dyadic scan, however, leading items of the result are produced by reducing vectors shorter than the length specified in the left argument. The window starts with a single item and grows, one item at a time, until it reaches the length specified in the left argument. Because of this, the result has the same shape as the right argument.

```
      3 +\ 1 1 1 1 1
1 2 3 3 3
      2 <\ 3 4 2
3 1 0
      ⎕←A←≠\0 0 1 0 0 0 1 0 1 1 0
0 0 1 1 1 1 0 0 1 0 0
      2 ≠\ A                    Inverse of ≠\ on Boolean arguments.
0 0 1 0 0 0 1 0 1 1 0
```

1.3.3 Each

The each operator applies a function to the items of its argument or between the items of its arguments to produce the items of its result. The each operator is represented by the dieresis symbol (¨).

```
      1 2 3 ⍴¨ 4 5 6
(4) (5 5) (6 6 6)
      1 2 3 ,¨ 4 5 6
(1 4) (2 5) (3 6)
      ⎕←R←(2 3 5) (7 11 13)
(2 3 5) (7 11 13)
      ⌽R
(7 11 13) (2 3 5)
      ⌽¨ R
(5 3 2) (13 11 7)
      ⌽ ⌽¨ R
(13 11 7) (5 3 2)
```

The each operator can accept as its argument a derived function such as plus-reduction. In fact, any operator can accept any derived function.

```
      +/¨ R
10 31
```

The next example builds a five-item vector, each of whose items is a two-item vector. Each two-item vector is an argument to the ⎕FREAD function. The result is a five-item vector, each of whose items is a component read from the file.

```
      FILE←⎕FREAD¨ ⎕← 2 ,¨ ⍳5
(2 1) (2 2) (2 3) (2 4) (2 5)
```

1.3.4 Power Limit

Power limit, represented by overstriking the symbols ¨ and *, executes a monadic function repeatedly using the result of one iteration as the argument to the next until the result is identical twice in a row. This result is returned.

Power limit can be used for converging series calculations or for loop-until-done computations.

```
    ∇ R←INPUT DATA
[1]   R←DATA,⎕
    ∇

      RESULT←INPUT⍣∘ ⍳0
⎕:
      2 3 5
⎕:
      7 11 13 17
⎕:
      ⍳0
      ⍴RESULT
7
```

1.4 Gaining Access

To gain access to the Nested Arrays System, log on to STSC's *APL*PLUS* VM System and enter the following commands:

```
~~DEFINE STORAGE 1M~~
NARS
```

The system will respond with:

```
   APL*PLUS SERVICE
  NESTED ARRAYS SYSTEM

CLEAR WS
```

1.5 Syntax Conventions Used in This Manual

Function and operator symbols are shown exactly as they appear in actual usage. Symbols such as parentheses are shown if required. Other symbols that appear in syntax displays and definitions are

A	An operator's left argument when an array.
B	An operator's right argument when an array.
f	An operator's left argument when a function.
g	An operator's right argument when a function.
h	The inverse of a function.
I	A counter which iterates through the indices of an argument.
J	A counter which iterates through the indices of an argument.
K	The selected coordinate of a function or operator.
L	A function's left argument (the function may be a derived function).
R	A function's right argument (the function may be a derived function).
Z	A result.

The examples shown below illustrate the use of these syntax conventions.

Z←∪*R*	The monadic unique function.
Z←*L*≡*R*	The equivalent function.
Z←*L* f/ [*K*] *R*	Dyadic reduction with explicitly selected coordinate.
Z←(f⍣*B*) *R*	The power operator.
Z←*L* f⍢g *R*	The dual operator.

Definitions are given either in textual form or in an equivalent expression. A symbol common to the syntax and definition represents the same argument. For example, the syntax, *Z*←f\R for scan could be defined as *Z*[*I*]←f/*I*↑*R*.

In some definitions, such as the above definition of scan, argument and result ranks are assumed even though the function or operator extends to arrays of other ranks. The assumed argument rank is the lowest rank which displays the salient characteristics of the function or operator; extensions to higher ranks are noted.

The double-headed arrow (↔) indicates that two expressions have the same meaning or return identical results (for instance, ⍴*L* ∘.f *R* ↔ (⍴*L*),⍴*R*).

Definitions and examples are given in origin 1.

WS FULL errors are not listed. Also not listed is a ⎕*IO IMPLICIT ERROR* signalled because ⎕*IO* is localized and not defined and an expression such as +/[1]*A* has been executed. In such a situation the error is in the axis selection rather than in the plus-reduction (the expression +/*A* executes without error).

CHAPTER 2

SET FUNCTIONS

The group of functions called set functions manipulate vectors based on a comparison of their items. None are sensitive to ⎕*CT*. Consequently, the arrangement of the result is not affected by ⎕*CT*.

Each of the set functions deals with ordered sets; that is, the order of the arguments is reflected in the order of the result.

The set functions are

- Unique, which returns the unique items of its argument.
- Set difference, which returns the items in its left argument not found in its right argument.
- Union, which returns the items in its left argument followed by the items in its right argument that are not found in its left argument.
- Intersection, which returns the items common to both arguments in the order they appear in the left argument.

2.1 Unique

Syntax

```
Z←∪R
```

Definition

```
Z←((R⍳R)=⍳⍴R)/R
```

Discussion

The unique function returns the unique items of its argument.

Argument and Result

The argument is a vector; a scalar is coerced into a one-item vector. The result is a vector with as many items as there are unique items in the argument.

Examples

```
      ∪'OH MY MY'
OH MY
      ∪'FIDDLE' 'DEE' 'DEE' 'FIDDLE' 'DEE' 'DUM'
(FIDDLE) (DEE) (DUM)
```

Errors

RANK ERROR	The argument is of rank greater than one.

2.2 Set Difference

Syntax

```
      Z←L~R
```

Definition

```
      Z←(~L∊R)/L
```

Discussion

Set difference returns items of its left argument that are not found in its right argument.

Arguments and Result

Both arguments are vectors; scalars are coerced into one-item vectors. The result is a vector with as many items as there are in the left argument that are not found in the right argument.

Examples

```
      'FLING'~'NOEL'
FIG
      'S Q U E E Z E'~' '
SQUEEZE
      'FIDDLE' 'DEE' 'DEE' 'FIDDLE' 'DEE' 'DUM' ~ ⊂'DEE'
(FIDDLE) (FIDDLE) (DUM)
```

Errors

RANK ERROR	One or both arguments have rank greater than one.

2.3 Union

Syntax

```
      Z←L∪R
```

Definition

```
      Z←L,R~L
```

Discussion

Union returns its left argument followed by those items of its right argument that are not found in its left argument.

Arguments and Result

Both arguments are vectors; scalars are coerced into one-item vectors. The result is a vector with as many items as there are in the left argument plus the number of items in the right argument that are not found in the left argument.

Example

```
      'BUFF'∪'BOON'
BUFFOON
```

Errors

RANK ERROR	One or both arguments have rank greater than one.

2.4 Intersection

Syntax

```
      Z←L∩R
```

Definition

```
      Z←(L∊R)/L
```

Discussion

Intersection returns those items that are common to its arguments in the order in which they appear in its left argument.

Arguments and Result

Both arguments are vectors; scalar arguments are coerced into one-item vectors. The result is a vector with as many items as the arguments have in common.

Example

```
      'HAIR'∩'RAID'
AIR
```

Errors

RANK ERROR	One or both arguments have rank greater than one.

CHAPTER 3

MISCELLANEOUS FEATURES

The Nested Arrays System provides many new features. Among the ones described in this chapter are

- Heterogeneous arrays -- arrays that can be composed of both numeric and character data.
- Ambivalant user-defined functions -- user-defined functions that can be called either monadically or dyadically.
- Expanded display potential -- allows suppression of the results produced by executing functions.
- Strand notation -- a means of representing both simple and nested vectors.

3.1 Zilde

Syntax

```
      ⍬                                      Overstrike:  0~
```

Definition

```
      ⍳0
```

Discussion

Zilde is the constant empty numeric vector.

Examples

```
      +/⍬                 Gives the identity element of addition.
0
      Z←M[⍬⍴A;]           Reshapes A into a scalar so that a row
                          of the matrix M will be selected
                          as a vector.
      ⍬ ≡ ⍴A              Is A a scalar?
0
```

3.2 Catenation along the First Dimension

Syntax

```
      Z←L⍪R                                          Overstrike:  ,-
```

Definition

```
      Z←L,[⎕IO] R
```

Discussion

The behavior of catenation is unchanged; ⍪ is an abbreviation for specifying catenation along the first dimension, just as ⌿ and ⊖ are abbreviations for specifying reduction and rotation along the first dimension.

Example

```
      'GOB'⍪ 2 3⍴'IRENET'
GOB
IRE
NET
```

Errors

Errors that are possible with catenation are also possible with catenation along the first dimension.

3.3 Heterogeneous Arrays

Definition

Heterogeneous arrays are arrays that can be composed of numeric data intermixed with character data.

Discussion

The concept of heterogeneous arrays permits *APL* arrays to be partially numeric and partially character in any desired pattern. This opens up a number of new possibilities for arranging data.

The concept of heterogeneous arrays should not be confused with the concept of nested arrays. An array can be simple but heterogeneous.

Examples

```
      'SMTWTFS';' ';5 7⍴'    ',⍳30
  S   M   T   W   T   F   S

              1   2   3   4
  5   6   7   8   9  10  11
 12  13  14  15  16  17  18
 19  20  21  22  23  24  25
 26  27  28  29  30
```

Student numbers and corresponding grades may be stored as follows:

```
      A←1442 3146 9931 8674,4 2⍴'A C-B+F '
      A
 1442 A
 3146 C-
 9931 B+
 8674 F
```

3.4 Type

Syntax

```
      Z←⊤R
```

Definition

Type replaces characters in its argument with blanks and numbers with zeros.

Argument and Result

The argument is any array. The result has the same structure as the argument.

Examples

```
      ⎕←A←?2 3ρ10
 2  8  5
 6  3  1
      ⊤A
 0  0  0
 0  0  0
      ' '=⊤'BLIMP'
1  1  1  1  1
      A←⊤'ONLY' , 4 , 'YOU'
      A=' '
1  1  1  1  0  1  1  1
      A=0
0  0  0  0  1  0  0  0
```

3.5 Sink

Syntax

```
      ←R
```

Definition

Sink absorbs its argument without displaying it. If sink is used in a traced function line, the value is displayed.

Discussion

Sink allows you to discard an unwanted value without resorting to reshaping the value into an empty matrix or assigning it to a system variable.

Example

```
      ←⎕DL 3×60 ◇ 'THE EGGS ARE READY!'
THE EGGS ARE READY!
```

3.6 Ambivalent User-Defined Functions

Syntax

```
      ∇ {LEFT} FUNCTION RIGHT
```

Definition

An ambivalent user-defined function has braces around the left argument name in its header. It can be called monadically or dyadically. When called monadically, its left argument is not defined; a *VALUE ERROR* is signalled within execution of the user-defined function if the left argument is referenced prior to being set.

Discussion

This feature permits creation of user-defined functions which can be called monadically or dyadically. When called monadically the function can either assume a default left argument or exhibit different behavior. Testing to see if the left argument was provided can be done by using the system function ⎕*NC* (name class).

Examples

```
     ∇ Z←{L} DIV R
[1]    ⍎(0=⎕NC 'L')/'L←1' ◇ Z←L÷R
     ∇

       3 DIV 4
0.75
       DIV 4
0.25

     ∇ {FNS} FNREPL REPL;...
[1]    →(0≠⎕NC 'FNS')⍴L1 ◇ FNS←ALLFNS
[2]   L1: ...
```

3.7 Enhanced Display Potential

A function that has braces around the result name in its header returns a result with display potential off. For example,

```
    ∇ {RESULT}←FUNCTION
```

This result is not displayed automatically in an executing function. It is displayed if returned by a function line which is traced.

This feature can be used when a function returns a useful result but when it would not be useful for this result to be displayed by default. For example,

```
    ∇ {Z}←L FAPPENDR R
[1]   Z←2⊃⎕FSIZE R ◇ L ⎕FAPPEND R
    ∇

      'GNU' ⎕FCREATE 9
      'PIRLJ' FAPPENDR 9
1
    ∇ FOO
[1]   'LET IT BE' FAPPENDR 9
    ∇

      FOO
      2⊃⎕FSIZE 9
3
```

In addition, certain system functions, when called from an executing function, return results that are not displayed automatically. The functions are ⎕DEF, ⎕DEFL, ⎕DL, ⎕ERASE, ⎕EX, ⎕FX, ⎕LOCK, ⎕SIGNAL, ⎕SVC (in its dyadic form), ⎕SVO, ⎕SVR, and ⎕WAIT.

The main purpose of these system functions is to produce a side effect rather than a result. Results of these system functions are displayed if the statements that call them are traced or if the functions are executed in immediate execution mode. For example,

```
    ∇ G
[1]   ⎕EX 'A'
    ∇
      A←5
      ⎕NC 'A'
2
      ⎕EX 'A'
1
      ⎕NC 'A'
0
      A←5
      G
      ⎕NC 'A'
0
```

3.8 Reshape Extended

Syntax

```
        Z←LρR
```

Definition

When the right argument of reshape is an empty array, the fill item of the right argument is supplied in each position of the result.

Discussion

When the right argument is empty, reshape returns fill items in the shape specified by the left argument. This makes it possible to define the fill item for an array as ⊤θρR.

Examples

```
        θρθ
0
        ' '=2 6ρ''
 1 1 1 1 1 1
 1 1 1 1 1 1
```

3.9 Equivalent

Syntax

Z←L≡R　　　　　　　　　　　　　　Overstrike: =_

Definition

Equivalent returns 1 if the left and right arguments are identical in rank, shape, items, and (if empty) prototypes. It returns 0 otherwise. Equivalent is sensitive to ⎕*CT*.

Discussion

Equivalent compares two arrays and determines whether they are identical in all respects.

Arguments and Result

The arguments can be of any rank, shape, and type. The result is a scalar 1 or 0. Equivalent is sensitive to ⎕*CT*.

Examples

```
      ⍬≡⍳0
1
      ⍬≡''
0
      0≡,0
0
      A←2 3⍴'CATDOG'
      A≡A
1
      A≡⍳4
0
      (2 3⍴' ')≡⍕A
1
```

The last example shows a useful technique for verifying that the structure and type of a value (perhaps an argument to a function) are what was expected.

Errors

⎕*CT IMPLICIT ERROR*　　　⎕*CT* is localized but not defined.

3.10 Inequivalent

Syntax

```
        Z←L≢R                                                Overstrike: ≠_
```

Definition

```
        Z←~L≡R
```

Inequivalent is sensitive to ⎕CT.

Discussion

Inequivalent returns 1 if its arguments are different in any respect and 0 if they are identical.

Arguments and Result

The arguments can be of any rank, shape, and type. The result is a scalar 1 or 0. Inequivalent is sensitive to ⎕CT.

Examples

```
        ⍬≢⍳0
0
        2 3 4 1 ≢ 2 3⍴'CATDOG'
1
        5≢'V'
1
```

Errors

⎕CT IMPLICIT ERROR	⎕CT is localized but not defined.

3.11 Strand Notation

Definition

Strand notation is a means of representing vectors, either simple or nested. Three kinds of constructs appear in strand notation: numeric values such as 12, constant character vectors such as '*HIERONYMUS BOSCH*', and expressions such as (*PICKLE*×*JUICE*). When two or more of any of these are adjacent, each is interpreted to be an item. Constructs which evaluate to simple scalars remain simple.

Discussion

Strand notation is an extension of the familiar notation used to enter a constant numeric vector. A position can now consist of a number (as before), a character scalar or vector, or an expression. An expression may need to be enclosed in parentheses to limit its scope.

Note that nesting occurs only when two or more constructs are adjacent.

Examples

All of the following examples (excluding the initial assignment) return three-item vectors.

```
      A←1 ◇ B←2 ◇ C←3 ◇ D←1 2 3
      A B C
1 2 3
      A B D
1 2 (1 2 3)
      A B C×2
2 4 6
      A B D+10
11 12 (11 12 13)
      A B (D+10)
1 2 (11 12 13)
      (1 9 4 1) 4 'YOU'
(1 9 4 1) 4 (YOU)
      A 'SNARK' 3.14
1 (SNARK) 3.14
      (2 3) 4 5
(2 3) 4 5
      5 '=' 'V'
5 =V
```

3.12 Strand Notation Assignment

Syntax

```
      C D E←R
```

Definition

Strand notation assignment allows more than one variable to be assigned in one operation. Each variable to the left of the assignment arrow receives the corresponding item of the vector to the right. A single-item right argument is coerced into a vector with one item for each variable name on the left.

Arguments

The right argument is a vector with as many items as there are names to the left of the assignment arrow.

Examples

```
      A B C←1 2 3
      A ◇ B ◇ C
1
2
3
      A B (C←('YOUR APPEARANCE')('IS')('OUR BUSINESS')
      A B C
(YOUR APPEARANCE) (IS) (OUR BUSINESS)
      A C←C A                          Exchange the values of A and C.
      A B C
(OUR BUSINESS) (IS) (YOUR APPEARANCE)

    ∇ {FNS} FNREPL REPL;...
[1]   ...
[2]   L1:OLD NEW←REPL ◇ ...
```

Errors

LENGTH ERROR	The right argument does not have the same number of items as there are names to the left.
RANK ERROR	The right argument is of rank greater than one.

CHAPTER 4

NEW OPERATORS

Operators are *APL* language constructs which operate on one or two arguments and produce derived monadic or dyadic functions. Derived functions usually provide different capabilities than the function or functions which were operated on.

A familiar example is +/*R*. The reduction operator takes the plus function as its argument and produces a monadic function which then operates on its argument *R*. In this example, the operator's left argument is addition and the (derived) function's right argument is *R*.

The operators that follow are new rules for deriving functions.

4.1 Behavior of Operators

Sections 4.1.1 through 4.1.3 outline some of the enhancements in operator behavior in the Nested Arrays System.

4.1.1 Accepting User-Defined Functions

User-defined functions can now be called by operators. This opens up many new possibilities for using operators as array-oriented control structures for executing defined functions.

For example, the function *EDIT* is called once for each item of *TEXT*.

```
      EDIT¨TEXT
```

4.1.2 Accepting Any Primitive Function

Formerly, operators could only accept scalar functions as their arguments. Removing this restriction is possible because existing operators have been redefined in such a way that they can return nested results. See Chapter 6 for details.

4.1.3 Operator Sequences

Operators have long left scope and short right scope. An operator takes as its left argument the function or derived function to the left. Parentheses may be used to limit the scope of an operator by interrupting the sequence. An operator takes as its right argument only the first function to its right. Parentheses may be necessary to lengthen an operator's right argument. Constrast this with functions which have long right scope and short left scope.

In the following example the each operator (see Section 4.10) takes as its left argument the derived function of reduction acting on addition.

```
      +/¨ (1 2) (3 4) (5 6)
3 7 11
```

In the next example the right argument of the dual operator is 100 composed with the multiplication function. (The dual operator is discussed in Section 4.15, and composition is discussed in Section 4.14.) The parentheses are necessary to make dual the primary operator and the function derived from composition as its right argument. The effect of the expression is to truncate digits to the right of the hundreth place.

```
      ⌊⍢(100∘×) 3.14159
3.14
```

The short right scope of the composition operator makes the expression +∘3×2 equivalent to (+∘3)×2, which is equivalent to (×2)+3.

```
      +∘3×2
4
      L+.×.÷R ←→ L(+.×).÷R
      +.×/R ←→ (+.×)/R
```

4.2 Dyadic Reduction

Syntax

```
      Z←L f/ R
      Z←L f⌿ R
      Z←L f/ [K] R
```

Definition

For (L≥0)∧L≤ρZ:

```
      Z[I]← f/ R[(I-1)+⍳L]
```

For (L<0)∧(|L)≤ρZ:

```
      Z[I]← f/ ⌽R[(I-1)+⍳-L]
```

When the magnitude of the function's left argument is greater than the length of the selected coordinate of the right argument, an empty array results.

Dyadic reduction extends to arrays of any rank by selecting the direction of the vectors to be reduced via the axis operator.

Discussion

Dyadic reduction performs reductions on the items in a "window", whose length is specified by the function's left argument. The window moves across the items in the right argument, one position at a time. For example 3 +/ 1 2 3 4 5 6 performs four plus-reductions, each on windows of width three and returns (1+2+3),(2+3+4),(3+4+5),4+5+6.

If the function's left argument is negative, the items in each window are reversed before the reduction is done.

Using 2 or ¯2 as the function's left argument is an especially useful case which applies a function between consecutive pairs of items in its argument. For instance, the expression ∧/ 2 </ V returns 1 if all the items in V are in ascending order and ∧/ 2 =/ V returns 1 if all the items in V are equal.

Arguments and Result

The function's left argument is an integer-valued scalar. The operator's left argument is a dyadic function. The function's right argument is an array.

The shape of the result is the shape of the function's right argument in all coordinates except the one reduced. The length of that coordinate (the Kth coordinate) is given by 0⌈(ρR)[K]+1-|L.

Examples

```
      0 +/ 1 1 1 1
0 0 0 0 0
      1 +/ 1 1 1 1
1 1 1 1
      2 +/ 1 1 1 1
2 2 2
      3 +/ 1 1 1 1
3 3
      4 +/ 1 1 1 1
4
      ρ5 +/ 1 1 1 1
0
      ¯2>/1 3 4 4 5 6 8 2
1 1 0 1 1 1 0
      Z←(N+/V)÷N              Unweighted moving average.
```

Errors

DOMAIN ERROR	The left argument is not simple and integer-valued.
INDEX ERROR	The selected coordinate does not exist in the function's right argument.
SYNTAX ERROR	The operator's left argument is not a dyadic function.
RANK ERROR	The function's left argument is not scalar.

4.3 Dyadic Scan

Syntax

```
      Z←L f\ R
      Z←L f⍀ R
      Z←L f\ [K] R
```

Definition

```
      Z←X,L f/ R
```

Where `X[I]←(I××L) f/ I↑R;` for `I∊⍳(⍴R)⌊¯1+|L`.

Dyadic scan extends to arrays of any rank by selecting the direction of the vectors to be scanned via the axis operator.

Discussion

Dyadic scan is similar to dyadic reduction. With dyadic scan, however, leading items of the result are produced by reducing vectors shorter than the length specified by the function's left argument. The window starts with a single item and grows, one item at a time, until it reaches the length specified in the left argument. From that point on, the result is identical to the result produced by dyadic reduction.

If the function's left argument specifies a length greater than the selected dimension of the function's right argument, dyadic scan behaves as though the magnitude of the function's left argument is equal to the selected dimension of the function's right argument.

The following identities illustrate possible uses of dyadic scan.

`2 ≠\` is the inverse function of `≠\` for Boolean arguments.

`¯2 -\` is the inverse function of `+\`.

`¯2 ÷\` is the inverse function of `×\` for non-zero arguments.

Arguments and Result

The function's left argument is an integer-valued scalar. The function's right argument is an array. The result has the same shape as the function's right argument.

Examples

```
      1 +\ 1 1 1 1
1 1 1 1
      2 +\ 1 1 1 1
1 2 2 2
      3 +\ 1 1 1 1
1 2 3 3
      4 +\ 1 1 1 1
1 2 3 4
```

```
      5 +\ 1 1 1 1
1 2 3 4
```

Errors

DOMAIN ERROR	The left argument is zero.
	The left argument is not simple and integer-valued.
SYNTAX ERROR	The operator's left argument is not a dyadic function.
RANK ERROR	The left argument is not scalar.

4.4 Convolution

Syntax

```
        Z←L f⍡g R                                   Overstrike:  ⊤¨
```

Definition

```
        Z[I]← f/(⌽(0⌈I-⍴R)↓(I⌊⍴L)↑L)g(0⌈I-⍴L)↓(I⌊⍴R)↑R
```

The above expression describes convolution when the function's left and right arguments are vectors; the extension to arrays of higher rank is similar to that of inner product except that the two inner dimensions are merged.

Discussion

The convolution operator performs a moving window inner product with the selected items of the left argument reversed. When the left and right arguments are vectors, the first items of the result are

```
        Z[1]← f/ L[1] g R[1]
        Z[2]← f/ L[2 1] g R[1 2]
        Z[3]← f/ L[3 2 1] g R[1 2 3]
        .
        .
        .
```

Arguments and Result

The operator's left and right arguments are dyadic functions. The left and right arguments of the derived function are arrays of any rank. The shape of the result is given by (¯1↓⍴L),(¯1+(¯1↑⍴L)+1↑⍴R),1↓⍴R.

Examples

The following example performs polynomial multiplication. The left and right arguments are vectors of coefficients. The result is the coefficients of the product polynomial.

```
      1 2 3 +⍡× 4 5 6 7
4 13 28 34 32 21
```

The following example illustrates how the result above was computed. The operator's right argument is produced using direct definition.

```
      TIMES←'(3 0⍕ +/ α × ω),''  ←→  +/ '',(⍕α),'' × '',⍕ω ◇ α×ω'
      1 2 3 + ⍡ (TIMES∇∘) 4 5 6 7
  4  ←→  +/ 1 × 4
 13  ←→  +/ 2 1 × 4 5
 28  ←→  +/ 3 2 1 × 4 5 6
 34  ←→  +/ 3 2 1 × 5 6 7
 32  ←→  +/ 3 2 × 6 7
 21  ←→  +/ 3 × 7
4 13 28 34 32 21
```

Another way to interpret the expression *Z*←*L* +⍤̈× *R* is to consider the right argument to be a vector of weights. The result is a weighted moving average. Contrast this with the unweighted moving average given as an example in dyadic reduction (Section 4.2).

Convolution can also be used to define ⎕*SS* as follows:

```
 A ⎕SS B  ←→  (⍴A)↑(1-⍴B)↓(¯1+⍴B)↓ A∧⍤̈=⌽B
```

<u>Errors</u>

DOMAIN ERROR The inner dimension of each argument is zero.

4.5 Mask

Syntax

```
      Z←L (A∘/) R
      Z←L (A∘⌿) R
      Z←L (A∘/[K]) R
```

Definition

A negative value in the operator's left argument selects the corresponding item from the function's left argument; a positive value selects the corresponding item from the function's right argument. The magnitude of a value in the operator's left argument indicates how many times to repeat a selected item. A zero in the operator's left argument leaves out the corresponding item in the function's left and right arguments.

When the function's left and right arguments are of rank greater than one, the vector operator's left argument selects or removes slices of the arrays along the designated coordinate.

Discussion

Mask chooses between items of its function's left and right arguments based on values in its operator's left argument. Items can be repeated or left out entirely.

Arguments and Result

When the three arguments are vectors, they are all of equal length. The operator's left argument contains integers. A scalar argument is reshaped to match the shape of a nonscalar argument. The shape of the result (possibly after the operator's left argument has been extended) is given by +/|A.

When the function's left and right arguments have rank greater than one, their shapes must be identical. The operator's left argument is an integer-valued vector with the same length as the selected dimension. When the function's left or right argument is a scalar, it is reshaped to match the shape of the other argument. When the operator's left argument is a scalar, it is reshaped to be a vector the length of the selected dimension of the function's arguments. The shape of the result has the shape of the function's left and right arguments in all dimensions except the selected one, which has length +/|A.

Examples

```
      'ABCDEF' (¯1 2 0 ¯3 1 0∘/) '123456'
A22DDD5
      'GLAD' (1 ¯1 2 ¯1∘/) 'FROG'
FLOOD
```

```
      ⎕←A←4 3ρ'ABCDEFGHIJKL'
ABC
DEF
GHI
JKL
      ⎕←B←4 3ρ'0123456789'
012
345
678
901
      A (2 ¯1 0 ¯3∘≠) B
012
012
DEF
JKL
JKL
JKL
      A (¯3 0 1∘/) B
AAA2
DDD5
GGG8
JJJ1
```

Errors

DOMAIN ERROR	The operator's left argument is not simple and integer-valued.
INDEX ERROR	The selected coordinate does not exist in the function's left or right argument.
LENGTH ERROR	The three arguments are not conformable.
RANK ERROR	The operator's left argument is not a vector or scalar.

4.6 Replicate Extended

Syntax

```
      Z← L / R
      Z← L ⌿ R
      Z← L /[K] R
```

Definition

```
      Z←(⊂T⊃R) (L∘/) R
      Z←(⊂T⊃R) (L∘⌿) R
      Z←(⊂T⊃R) (L∘/[K]) R
```

Discussion

Replicate has been extended to allow negative values in the left argument to select the fill item of the right argument. The concept of fill item is discussed in Section 6.9.

Arguments and Result

The left argument to replicate is a simple integer-valued vector the same length as the selected dimension of the right argument. The right argument is an array. A scalar left or right argument will be reshaped to the length of the other argument. The result has the shape of the right argument in all dimensions except the selected one; the length of that dimension is given by +/|*L*.

Examples

```
      1 ¯2 1 / 2 3 1
2 0 0 1
      0 1 1 0 1 ¯1 1 1 0 1 / 'WONDERLAND'
ONE LAD
```

Errors

DOMAIN ERROR	The left argument is not simple and integer-valued.
INDEX ERROR	The selected coordinate does not exist in the right argument.
LENGTH ERROR	The left and right arguments are not conformable.
RANK ERROR	The left argument is not a vector or scalar.

4.7 Mesh

Syntax

```
      Z←L (A∘\) R
      Z←L (A∘⍀) R
      Z←L (A∘\[K]) R
```

Definition

The operator's left argument to mesh is a control vector which indicates how the function's left and right arguments are to be interleaved and where fill items of the function's right argument are to be inserted.

The *I*th positive value of the operator's left argument selects the *I*th item of the function's right argument. The *J*th negative value of the operator's left argument selects the *J*th item of the function's left argument. A zero in the operator's left argument indicates that a fill item from the function's right argument is to be inserted. Except for a zero, the magnitude of a value in the operator's left argument indicates how many times to repeat a selected item of the function's left or right argument.

Mesh extends to arrays of rank greater than one by merging together slices of the function's left and right arguments as directed by the selected coordinate and the operator's left argument.

Discussion

Mesh merges its function's left and right arguments, possibly repeating items and possibly inserting fill items of the function's right argument.

Arguments and Result

The operator's left argument is an integer-valued vector or scalar.

For vectors:

```
      ⍴L ←→ +/A<0
      ⍴R ←→ +/A>0
      ⍴Z ←→ +/1⌈|A
```

The function's left and right arguments must match in all dimensions except the selected one which follows the rules for vectors.

A scalar left or right argument is reshaped to the shape of the nonscalar argument in all dimensions except the selected one which is determined by the rules for vectors.

The result has the same shape as the function's left and right arguments except along the selected dimension which follows the rule for vectors.

Examples

```
      'I' (1 ¯1 2 ¯1 2 ¯1 2 ¯1∘\) 'MSSP'
MISSISSIPPI
      1 2 (¯2 ¯1 0 1 2 3∘\) 7 8 9
1 1 2 0 7 8 8 9 9 9
      'BIB' (¯1 ¯1 1 ¯1 1 0 1 1 2 1 1 1∘\) 'LOBAGINS'
BILBO BAGGINS
```

Errors

DOMAIN ERROR	The operator's left argument is not simple and integer-valued.
LENGTH ERROR	The argument shapes are not conformable.
RANK ERROR	The operator's left argument is not a vector.

possible

4.8 Expansion Extended

Syntax

```
      Z←L \ R
      Z←L ⍀ R
      Z←L \[K] R
```

Definition

```
      Z←(⊂⊤⊃R) (L∘\) R
      Z←(⊂⊤⊃R) (L∘⍀) R
      Z←(⊂⊤⊃R) (L∘\[K]) R
```

Discussion

Expansion has been extended to allow the left argument to be integer-valued. (In conventional *APL*, the left argument is restricted to Boolean values.) If a negative value occurs in the left argument, the fill item of the right argument is selected for the result; if a positive value occurs in the left argument, an item is selected from the right argument. The magnitude of values in the left argument indicates the number of times an item of the right argument or a fill item is to be included in the result except that 0 and ¯1 have the same effect which is to include one fill item.

Arguments and Result

The left argument is a simple integer-valued vector. The number of positive values must equal the selected dimension of the right argument. The right argument is an array. The result has the shape of the right argument in all dimensions except the selected one which is given by +/1⌈|L.

Examples

```
      1 1 2 1 1 ¯1 1 1 1 \ 'BUTERFLY'
BUTTER FLY
      2 0 1 ¯1 1 ¯2 1 \ 1 2 3 4
1 1 0 2 0 3 0 0 4
```

4.9 Commute

Syntax

```
      Z←L f⍨ R                                      Overstrike: ~¨
```

Definition

```
      Z←R f L
```

Discussion

Commute causes the left and right arguments to a function or derived function to be exchanged before being presented to the function. It is of use mainly in sequences of operators.

Argument and Result

The function or derived function argument of commute must be dyadic.

Examples

```
      3-⍨4
1

[5]   →L3 /⍨ ⍬≡⍴A               Read the /⍨ as "if" (branch
                               to L3 if A is scalar).

      -/ 1 2 3 4 5
3
      1-2-3-4-5
3
      -⍨ / 1 2 3 4 5
¯5

    ∇ Z←L CSUB R
[1]   Z←R-L
    ∇

      CSUB/ 1 2 3 4 5
¯5
      (((5-4)-3)-2)-1
¯5
```

Errors

SYNTAX ERROR	The left argument of commute is not a dyadic function.

✓

4.10 Each

Syntax

```
      Z← f¨ R
      Z← L f¨ R
```

Definition

```
      I⊃Z ←→ f I⊃R
      I⊃Z ←→ (I⊃L) f I⊃R
```

For *I*∊⍳⍴*R*.

Discussion

The each operator applies a function to its arguments item by item. This provides the power of scalar extension to any function. Note that the scalar extension applies only to the outermost level; each does not produce a pervasive derived function.

Arguments and Result

In the dyadic form, the function's left and right arguments must be scalar conformable; the shape of the result is determined by the same rules that apply to scalar functions.

Examples

```
      1 2 3 4 ↑¨ 5 6 7 8
(5) (6 0) (7 0 0) (8 0 0 0)
      A
((1 2 3) (4 5 6)) ((7 8 9) (10 11 12))
      ⌽A
((7 8 9) (10 11 12)) ((1 2 3) (4 5 6))
      ⌽¨A
((4 5 6) (1 2 3)) ((10 11 12) (7 8 9))
      ⌽¨¨A
((3 2 1) (6 5 4)) ((9 8 7) (12 11 10))
      ⌽¨¨¨A
((1 2 3) (4 5 6)) ((7 8 9) (10 11 12))

      1 2 3 4 ,¨ 5 6 7 8
(1 5) (2 6) (3 7) (4 8)
      2 ,¨ ⍳4
(2 1) (2 2) (2 3) (2 4)
```

In the following example, *TIENO* is catenated, item by item, to the items of ⍳10. The result is a ten-item vector, each of whose items is a two-item vector. Next, the function ⎕*FREAD* is applied to each two-item vector. The final result is a ten-item vector containing the first ten components of the file tied to *TIENO*.

```
      C←⎕FREAD¨TIENO,¨⍳10
```

Errors

LENGTH ERROR	The function's left and right arguments are not conformable.
RANK ERROR	The function's left and right arguments are not of compatible rank.

4.11 Power Operator

Syntax

```
      Z←(f⍣B) R                                        Overstrike: ⋆¨
```

Definition

```
      Z←f f f . . . f R
or
      Z←h h h . . . h R
```

The operator's left argument is a monadic function which is applied to the function's right argument the number of times specified by the right operator argument. If the operator's right argument is negative, the inverse of the function (represented by h above) is applied the number of times specified by the magnitude of the right operator argument. If the operator's right argument is zero, the function's right argument is returned unchanged.

Arguments

The operator's left argument is a monadic function. The operator's right argument is an integer-valued scalar. The function's right argument is an array.

Examples

```
      (+/⍣3) 2 2 2 2⍴1
8 8

     ∇ R←FIB A
[1]    ⍝ CALCULATE THE NEXT TERM IN THE FIBONACCI SERIES
[2]    R←A,+/¯2↑A
     ∇

      (FIB⍣15) 1 1                     Calculate the next 15 terms.
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

      ⎕←B←+\ 1 1 1 1 2
1 2 3 4 6
      (+\⍣¯1) B
1 1 1 1 2
```

The following expression conditionally reshapes *A* into a two by five matrix if *A* is scalar; otherwise *A* is returned unchanged.

```
      A←(2 5∘⍴⍣(⍬≡⍴A)) A
```

The parentheses around ⍬≡⍴*A* are necessary because without further direction the short right scope of the power operator would take only the zilde as the operator's right argument.

Errors

DOMAIN ERROR	The operator's right argument is not integer-valued.
	The operator's left argument is not an invertible function.
RANK ERROR	The operator's right argument is not a scalar.
SYNTAX ERROR	The operator's left argument is not a monadic function.

4.12 Power Limit

Syntax

```
      Z← f⍣∘ R                                    Overstrike: *¨
```

Definition

```
      Z← f f f . . . f R
```

The operator's left argument is a monadic function which is repeatedly applied to the function's right argument using the result of one iteration as the argument to the next until the result is identical twice in a row. This result is returned. The derived function produced by this operator is sensitive to ⎕CT.

Discussion

Power limit is useful for converging series calculations or other loop-until-done computations.

Arguments

The operator's left argument is a monadic function. The function's right argument is a value to be applied to the derived function.

Examples

```
      ? ⍣∘ 10⍴1000
1 1 1 1 1 1 1 1 1 1

    ∇ Z←SQRT A
[1]   ⍝ CALCULATE SQUARE ROOT USING NEWTON'S METHOD
[2]   Z←A∘('0.5×ω+α÷ω'∇∘)⍣∘A
    ∇

      SQRT 2
1.414213562
      2*0.5
1.414213562
```

Errors

SYNTAX ERROR	The operator's left argument is not a monadic function.
⎕CT IMPLICIT ERROR	⎕CT is localized but not defined.

4.13 Power Series

Syntax

```
     Z← f⍣g R                                        Overstrike:  *¨
```

Definition

```
     Z←I f R f (R f.g R) f (R f.g R f.g R) f . . .
```

I is a matrix the shape of the right function argument with g/⍬ on the diagonal and f/⍬ elsewhere. Inner products are repeatedly performed until the accumulation is identical twice in a row.

The derived function produced by this operator is sensitive to ⎕*CT*.

Discussion

The power series operator does a series of inner products between a square matrix and itself. The successive inner products are accumulated until the result is identical twice in a row.

This operator gets its name from series like

```
     1 + X + (X*2) + (X*3) + . . .
```

Arguments and Result

The operator's arguments must be dyadic functions with identity elements. The function's right argument must be a square matrix. The result has the same shape as the function's right argument.

Example

A network of nodes with one-way connections can be represented as a Boolean matrix with 1's marking the connections. For example, a 1 in the second row, third column, would indicate that there is a direct path from node two to node three. If the reverse path exists, it would be represented by a 1 in the third row, second column.

If this matrix is stored in the variable *NET*, the possible journeys of two jumps are computed by *NET*∨.∧*NET*. Journeys of three jumps are computed by *NET*∨.∧*NET*∨.∧*NET* and so forth. The expression ∨⍣∧*NET* computes all possible journeys of any length. For example,

```
      NET
 0 0 1 0
 0 0 0 1
 0 1 0 0
 0 0 0 0
```

The preceding matrix indicates that the following one-way moves are possible:

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

Here are the journeys of length two:

```
      NET∨.∧NET
0 1 0 0
0 0 0 0
0 0 0 1
0 0 0 0
```

And length three:

```
      NET∨.∧NET∨.∧NET
0 0 0 1
0 0 0 0
0 0 0 0
0 0 0 0
```

All possible journeys of any length (including zero) are accumulated into one matrix as follows:

```
      ∨⍣∧ NET
1 1 1 1
0 1 0 1
0 1 1 1
0 0 0 1
```

<u>Errors</u>

DOMAIN ERROR	There is no identity element for one or both of the operator's arguments.
RANK ERROR	The function's right argument is not a matrix.
SYNTAX ERROR	One or both of the operator's arguments are not dyadic functions.
⎕*CT IMPLICIT ERROR*	⎕*CT* is localized but not defined.

4.14 Composition

Syntax

```
      Z←  f∘g R
      Z←L f∘g R
      Z← (f∘B) R
      Z←  A∘g R
```

Definition

```
      Z←   f g R
      Z← L f g R
      Z← R f B
      Z← A g R
```

Discussion

Composition assembles functions and arguments to form derived functions. This provides the glue with which to put together functions and arguments so that operator's arguments can be more complex than single functions.

Examples

```
      (*∘2) 10            The square function.
100
      (*∘0.5) 100         The square root function.
10
      +/∘⍳ 4
10
```

In the following example, +/ and ⍳ are composed to form a monadic derived function which adds up iota vectors. This derived function is applied by the each operator, item by item, to the values in the right function argument.

```
      +/∘⍳¨1 2 4
1 3 10
      Z←⍋∘⍋¨ V            Produce a ranking vector for each item of V.
```

✓

4.15 Dual

Syntax

```
      Z←  f⍤̈g R                                         Overstrike:  ∇¨
      Z←L f∇̈g R
```

Definition

```
      Z← h f g R
      Z← h ( g L) f g R
```

The inverse of the function g is represented by h.

Discussion

Dual applies the operator's right argument which is a monadic function separately to each argument of the derived function, applies the operator's left argument to the result of that, then applies the inverse function of the operator's right argument to the result.

In a physical context this is comparable to opening a drawer, altering its contents, and then closing the drawer.

Arguments

The function which is the operator's right argument must have an inverse. Since inverses cannot be computed for user-defined functions, a user-defined function cannot be used as the right function to dual.

Examples

```
      ⍴A←'DELETE TRAILING BLANKS'
30
      ⎕←B←(∨\∇̈⌽ A≠' ')/A
DELETE TRAILING BLANKS
      ⍴B
22
      ⌊∇̈(100∘×) 3.14159
3.14
      +/∇̈(*∘2) 3 4        Pythagorean theorem.
5
```

Errors

DOMAIN ERROR	The operator's right argument is not an invertible function.
SYNTAX ERROR	The function which is the operator's right argument has the wrong valence.

Possible

4.16 Function Table

Syntax

```
      Z←∘.g R
```

Definition

```
      Z←⊃ ∘.g/ ι¨ R
or
      Z←(ι1⊃R) ∘.g (ι2⊃R) ∘.g . . .
```

The derived function produced by the function table operator is sensitive to ⎕*IO*.

Discussion

The function table operator applies a designated outer product between the iota vectors of each item of its argument.

Examples

```
       ∘.×3 6
  1  2  3  4  5  6
  2  4  6  8 10 12
  3  6  9 12 15 18
       ∘.=4 4
 1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1
       ∘.,2 5
 (1 1) (1 2) (1 3) (1 4) (1 5)
 (2 1) (2 2) (2 3) (2 4) (2 5)
```

Errors

SYNTAX ERROR	The operator's right argument is not a dyadic function.
⎕*IO IMPLICIT ERROR*	⎕*IO* is localized but not defined.

4.17 Direct Definition

Syntax

```
Z←  (∘∇B) R      Monadic
Z←L (A∇∘) R      Dyadic
Z←  (A∇B) R   ]  Ambivalent
Z←L (A∇B) R   ]
```

Definition

Direct definition produces a derived function from the expressions contained in the character vector argument or arguments.

A character vector to the left of the operator represents a dyadic definition; one to the right represents a monadic definition. If both appear, an ambivalent function is created. Otherwise, a jot replaces the unused argument and a derived function of fixed valence is created.

Each character vector argument contains one or more *APL* expressions. The symbols alpha (α), omega (ω), and underscore (_) represent respectively the left argument, the right argument and the function itself. Alpha and omega contain the arguments to the function and can be reassigned. Alpha cannot be used in monadic definitions. Underscore permits recursive definitions.

If colons appear in a character vector, they serve to group series of expressions. The result of the group of expressions before the first colon must be a simple integer scalar or vector containing no negative values. This determines the sequence in which the remaining groups of expressions are executed. The remaining groups are numbered beginning with zero.

The result of the function is the result of the last expression executed.

Discussion

Direct definition is a convenient means of creating temporary functions, especially for use as arguments to operators.

Arguments and Result

Each argument to the operator is a simple character vector or scalar; a scalar is coerced to a one-item vector.

The result of the function is the result of the last expression executed.

If colons appear in the definition, the result of the group of expressions before the first colon must be a simple integer scalar or vector which contains no negative values.

Examples

```
      (∘∇'(+/ω)÷ρω') 2 4 6            Monadic-only average function.
4
      (∘∇'(+/ω)÷ρω')¨ (2 4 6) (10 40 30 50) (1 2)
(4) (32.5) (1.5)

      D←'α÷ω' ◇ M←'1÷ω'
      3 (D∇M) 4                       Ambivalent function equivalent to the
0.75                                  ÷ symbol.

      (D∇M) 4
0.25

      MIN←'α>ω : α : ω'
      3 (MIN∇∘) 4                     Dyadic-only minimum function.
3
      4 (MIN∇∘) 3
3

      (∘∇'ω=0 : ω×_ω-1 : 1')¨ 0 1 2 3 4 5 6          Factorial.
1 1 2 6 24 120 720

      A←(∘∇'0 ◇ 2 0 0 : 1 ◇ 2 : 3 ◇ 4 : 5 ◇ 6') 256
0
5
6
1
2
1
      A
2
```

Errors

DOMAIN ERROR	An argument to the operator is not simple and character.
	The result of the expressions to the left of the first colon is not simple and integer-valued.
INDEX ERROR	The result of the expressions to the left of the first colon did not select an existing group of expressions.
RANK ERROR	An argument to the operator is not a scalar or vector.
	The result of the expressions to the left of the first colon is not a scalar or a vector.
SYNTAX ERROR	The derived function was called with a valence for which it was not defined.
	The α symbol appears in the monadic definition.
VALUE ERROR	The last expression executed did not return a result.

CHAPTER 5

NESTED ARRAY FUNCTIONS

The functions described in this chapter manipulate nested arrays in various ways. Among the functions included in this chapter are

- Enclose, which creates nested scalars.
- Disclose, which removes a layer of nesting.
- Pick, which selects objects from nested arrays.
- Split, which lowers the rank of an array by enclosing a coordinate.
- Mix, which raises the rank of an array by disclosing and inserting a coordinate.
- Choose, which can index scattered points of any rank array.

5.1 Enclose

Syntax

```
      Z←⊂R
```

Definition

Enclose turns any array except a simple scalar into a nested scalar. A simple scalar is returned unchanged. For any other array, a layer of nesting is applied and a nonsimple scalar is returned.

Discussion

Enclose and disclose are the fundamental operations that create and dissolve nested arrays. Enclose can be thought of as putting a scalar wrapping around an array. Enclose returns a simple scalar argument unchanged; a nonsimple scalar or an array of rank greater than zero is enclosed into a nonsimple scalar.

Argument and Result

The right argument can be any array. The result is a scalar.

Examples

```
      ⊂5
5
      ⊂1  2  3
(1  2  3)
      ρρ⊂2  3ρ ι6
0
```

5.2 Disclose

Syntax

```
      Z←⊃R
```

Definition

Disclose returns the first item of an array or the fill item if the array is empty.

Discussion

Disclose is the inverse function of enclose; that is, *R* ←→ ⊃⊂*R*. It removes the scalar wrapping from a nested scalar, revealing the item within.

When the argument to disclose is an empty array, disclose returns the prototype or fill item (see Section 6.9) for the array. Disclose applied to an empty array is also called extract.

When the argument to disclose is not an empty array, disclose operates only on the first item of the array; that is, ⊃*A* ←→ ⊃⍬⍴*A*. Disclose applied to an array which has more than one item is also called first.

When the argument to disclose is simple, disclose returns the first item or the fill item as a scalar.

Argument and Result

The argument is any array. The result is the first item within the array.

Examples

```
      ⊃4 5 6
4
      A
(9 7 5) (2 3 4)
      ⊃A
9 7 5
      ⊃0⍴⊂2 3⍴⍳6
 0 0 0
 0 0 0
```

5.3 Partitioned Enclose

Syntax

```
      Z←L⊂R
      Z←L⊂[K] R
```

Definition

```
      I⊃Z  ←→  (I=+\L)/[K] R
```

Discussion

Partitioned enclose builds a nonsimple vector out of selected portions of an array. The leading item of the left argument does not have to be 1; the first selected portion begins where the first 1 appears.

Arguments and Result

The left argument is a Boolean vector with the same length as the selected coordinate of the right argument. A scalar left argument is reshaped to match the selected dimension of the right argument. The right argument is an array of any rank. The selected dimension of the right argument must equal the length of the left argument. The result is a vector of length *+/L* containing the selected portions of the right argument.

Examples

```
      ⎕←A←0  0  1  0  1  1  0  0  ⊂  1  2  3  4  5  6  7  8
(3  4)  (5)  (6  7  8)
      ρA
3
      1⊂1  2  3
(1)  (2)  (3)
      1  0  1  ⊂[1]  3  4ρ ι12
(  1  2  3  4    (  9  10  11  12)
   5  6  7  8)
```

Errors

DOMAIN ERROR	The left argument is not simple and Boolean.
LENGTH ERROR	The length of the left argument does not match the selected dimension of the right argument.
RANK ERROR	The left argument is not a vector.

5.4 Pick

Syntax

```
      Z←L⊃R
```

Definition

```
      ⍬⊃R ←→ R
      L⊃R ←→ (1↓L)⊃⊃R[⍬⍴L]
```

This function is sensitive to ⎕IO.

Discussion

Pick indexes portions of an array through any number of layers of nesting. The right argument is an array to be "picked through". The left argument is a vector, each item of which indexes through a layer of nesting in the right argument.

For example, in the expression 2 3⊃(1 2 3) (4 5 6), the 2 selects the second item, which is the vector 4 5 6, and the 3 selects the third item of that vector; the result is the simple scalar 6.

The shape of each item in the left argument must equal the rank of the array which has been picked up to this point, except that a scalar item is coerced into a one item vector which can pick from a vector. In particular, only an enclosed empty vector can pick a scalar.

Arguments and Result

The right argument is any array. The left argument is an integer-valued vector which may be nonsimple. The result is the portion of the right argument which is selected by the left argument. Pick is sensitive to ⎕IO.

Examples

```
      A
   ((1 2) 3 4)     ((2 4) 6 8)     ((3 6) 9 12)
 ((4 8) 12 16) ((5 10) 15 20) ((6 12) 18 24)
      A≡⍬⊃A
1
      (⊂2 1)⊃A
(4 8) 12 16
      ((⊂2 1),1)⊃A
4 8
      ((⊂2 1),1 2)⊃A
8
      3≡(⊂⍬)⊃3
1
```

Errors

DOMAIN ERROR	The left argument is not integer-valued.
INDEX ERROR	The left argument selects nonexistent positions of the right argument.
LENGTH ERROR	The shape of an item of the left argument does not match the rank of the selected portion of the right argument.
RANK ERROR	The left argument or an item of the left argument is not a scalar or vector.
⎕*IO IMPLICIT ERROR*	⎕*IO* is localized but not defined.

5.5 Simple

Syntax

```
      Z←=̲R                                          Overstrike:  =_
```

Definition

```
      Z←R≡⊂¨R
```

Discussion

Simple returns a scalar 1 if the argument is simple and a scalar 0 otherwise.

Argument and Result

The argument is any array; the result is a Boolean scalar.

Examples

```
      =̲⍳3
1
      =̲⊂⍳3
0
```

5.6 Not-Simple

Syntax

```
      Z←≠̲R                                    Overstrike:  ≠_
```

Definition

```
      Z←R≠̲⊂¨R
```

Discussion

Not-simple returns a scalar 1 if the argument is not simple and a scalar 0 otherwise.

Examples

```
      ≠̲⍳3
0
      ≠̲⊂⍳3
1
```

5.7 Split

Syntax

```
      Z←↓R
      Z←↓[K] R
```

Definition

Split encloses a dimension of an array. The rank of the result is one less than that of the argument. Each item of the result is a vector with the length of the enclosed dimension.

A simple scalar is returned unchanged; a nested scalar is enclosed once more and its rank remains 0.

Discussion

The split function segments an array along a coordinate into a nested array.

Argument and Result

```
      ρZ ←→ (K≠ιρρR)/ρR

      ρ⊃Z ←→ (ρR)[K]
```

Examples

```
      ↓3 4ρι12
(1 2 3 4) (5 6 7 8) (9 10 11 12)
      ρA←↓2 3 4ρι24
2 3
      ρ⊃A
4
      ρA←↓[2] 2 3 4ρι12
2 4
      ρ⊃A
3
```

Errors

INDEX ERROR	The selected dimension does not exist in the argument.

5.8 Mix

Syntax

```
      Z←↑R
      Z←↑[K] R
```

Definition

Mix assembles the arrays which are the items of its argument. These arrays have identical shape. A fractional number within brackets specifies the two dimensions between which the additional coordinates are to be inserted. The result is an array with additional dimensions inserted between the specified ones. The added dimensions have the shape of the items of the right argument.

Discussion

Mix assembles the items of an array into a higher-dimensional array with one less level of nesting.

Argument and Result

The right argument is an array consisting of identically shaped items. The shape of the result is the shape of the argument with the shape of the items inserted between the specified dimensions.

Examples

```
      ↑(1 2 3 4) (5 6 7 8) (9 10 11 12)
  1  2  3  4
  5  6  7  8
  9 10 11 12
      ρ↑↓ 3 0ρ0
3 0
      ρ↑ 1 2ρ⊂3 4ρ0
1 2 3 4
      ρ↑[1.5] 1 2ρ⊂3 4ρ0
1 3 4 2
```

Errors

INDEX ERROR	The selected dimensions do not exist in the argument.
LENGTH ERROR	The items in the argument are not conformable.

5.9 Choose (Indexing Extended)

Syntax

```
      Z←L[I]
      L[I]←Z
```

Definition

When indices within brackets are not simple, each item is a vector which forms an index to the array being indexed.

No semicolons are permitted within the brackets when the indices are not simple.

Choose is sensitive to ⎕*IO*.

Discussion

Each item of the array within brackets is a vector the length of the rank of the array being indexed. Scattered points of any array can be indexed by this function. In particular, scalars can be indexed by enclosed empty vectors.

Arguments and Result

The argument to the left of the brackets is an array of any rank. The argument within brackets is an array, each item of which is a vector of the same length as the rank of the indexed array. Expressed differently, (∧/2≡/ρ¨,I)∧(ρρL)≡ρ⊃I. The result or the value being assigned into the array has the same shape as the indices unless a scalar is being assigned. In this case, the scalar is reshaped to match the shape of the indices. Each item within brackets must form a valid index to the left argument array. Choose is sensitive to ⎕*IO*.

Examples

```
      5[2 4ρ⊂⍬]
 5 5 5 5
 5 5 5 5
      ⎕←A←3 4ρ⍳12
  1  2  3  4
  5  6  7  8
  9 10 11 12
      A[(3 2) (1 4) (2 3)]
10 4 7
      A[2ρ¨⍳3]←0
      A
  0  2  3  4
  5  0  7  8
  9 10  0 12
      A ≡ A[⍳ρA]
1
```

Errors

DOMAIN ERROR	The argument within brackets is not integer-valued.
INDEX ERROR	An item of the argument within brackets does not form a valid index to the left argument.
LENGTH ERROR	The length of an item of the argument within brackets is not equal to the rank of the left argument.
	The shape of the argument within brackets does not match the shape of the array being assigned.
RANK ERROR	The rank of the array within brackets does not match the rank of the array being assigned.
⎕*IO IMPLICIT ERROR*	⎕*IO* is localized but not defined.

5.10 Index Generator Extended

Syntax

```
      Z←⍳R
```

Definition

```
      Z←∘.,R
or
      Z←⊃ ∘.,/ ⍳¨ R
```

The index generator is sensitive to *⎕IO*.

Discussion

The argument to monadic iota can now be a vector. Iota returns a result such that *A* ←→ *A*[⍳⍴*A*] where *A* can be of any rank.

Argument and Result

The argument to the index generator must be a simple integer-valued vector or scalar. A scalar argument is coerced into a one-item vector. The value of the argument determines the shape of the result; that is, ,*R* ←→ ⍴⍳*R*.

Example

```
      ⍳3 4
 (1 1) (1 2) (1 3) (1 4)
 (2 1) (2 2) (2 3) (2 4)
 (3 1) (3 2) (3 3) (3 4)
```

Errors

DOMAIN ERROR	The argument is not simple and integer-valued.
RANK ERROR	The argument has rank greater than one.
⎕IO IMPLICIT ERROR	*⎕IO* is localized but not defined.

CHAPTER 6

EXISTING FUNCTIONS AND OPERATORS ON NESTED ARRAYS

This chapter describes how the functions and operators in conventional *APL* implementations operate in the Nested Arrays System. On simple arguments, functions and operators behave no differently than the primitives on the *APL*PLUS* System. Operators acting on scalar functions and simple arguments also are not changed.

6.1 Scalar Functions

Syntax

```
      Z←L f R
      Z←  f R
```

Definition

```
      I⊃Z ←→  (I⊃L) f I⊃R
      I⊃Z ←→        f I⊃R
```

Discussion

Scalar functions are pervasive through nested arrays; that is, they apply to the data through every level of nesting.

Arguments and Result

At each level of nesting, scalar conformability rules apply and determine result shapes.

Examples

```
      (2 4 1) (3 6 5) ⌊ (8 1 2) (6 5 4)
(2 1 1) (3 5 4)
      (1 3 2) 4 + 3 (5 3 2)
(4 6 5) (9 7 6)
      A
( 1 2  (5 6) 7 (8 (9 (10 11)))
  3 4)
      A×2
( 2 4  (10 12) 14 (16 (18 (20 22)))
  6 8)
```

Errors

LENGTH ERROR	The arguments do not follow scalar conformability rules.
RANK ERROR	The ranks of the arguments do not conform.

6.2 Reduction

Syntax

```
      Z←f/R
      Z←f⌿R
      Z←f/[K]R
```

Definition

For a one-item vector:

```
         Z←⍬⍴R
```

Otherwise:

```
         Z←⊂(⊃R)f⊃f/1↓R
```

Discussion

The specified function operates between the items of the array. Another way to represent the behavior of reduction is

```
      Z←⊂(1⊃R) f (2⊃R) f . . . f (⍴R)⊃R
```

Arguments and Result

The operator argument is no longer required to be a scalar function because enclosing the result permits reduction's result shape rules to be followed. The right argument is not necessarily simple.

Examples

```
      +/(1 2 3) (4 5 6) (7 8 9)
(12 15 18)
      ∊/'LAZARUS LONG' 'ROBERT HEINLEIN'
(1 0 0 0 1 0 0 1 1 1 1 0)
```

Errors

DOMAIN ERROR	The function has no identity element and reduction along a dimension of length zero was attempted.
SYNTAX ERROR	The function specified is not dyadic.

6.3 Scan

Syntax

```
      Z←f\R
      Z←f⍀R
      Z←f\[K]R
```

Definition

```
      Z[I]←f/I↑R
```

Discussion

The redefinition of reduction affects the behavior of scan. The operator argument is not limited to scalar functions since the function is applied between items of the argument and the result is enclosed.

Arguments and Result

The operator's argument can be any dyadic function. The right argument need not be simple. The result has the shape of the argument and is not necessarily simple.

Example

```
      +\(1 2 3) (4 5 6) (7 8 9)
(1 2 3) (5 7 9) (12 15 18)
```

Errors

SYNTAX ERROR	The operator argument is not dyadic.

6.4 Inner Product

Syntax

```
      Z←L f.g R
```

Definition

```
      Z←f/ L g¨ R
```

Discussion

The redefinition of reduction, scalar, and other functions also affects inner product.

Arguments and Result

The operator's arguments can be any dyadic functions. The arguments follow the usual conformability rules but need not be simple. The result follows inner product's usual rules and need not be simple.

Example

```
      1 2 3 ,., 4 5 6
(1 4 2 5 3 6)
```

Errors

DOMAIN ERROR	A reduction of an empty argument was attempted for a function which has no identity element.
SYNTAX ERROR	One or both operator arguments are not dyadic functions.

6.5 Outer Product

Syntax

```
      Z←L ∘.f R
```

Definition

```
      Z[I;J]←⊂(I⊃L) f J⊃R
```

Discussion

Outer product applies a function which is the operator's right argument between the items of the arguments to produce the corresponding item of the result.

Arguments and Result

The operator's right argument must be a dyadic function. The shape of the result is (as before) the shape of the left argument catenated to the shape of the right argument.

Examples

```
      (1 2) (3 4) ∘.+ (5 6) (7 8) (9 10)
  (6 8)  (8 10) (10 12)
 (8 10) (10 12) (12 14)
      1 2 3∘.,4 5 6 7
 (1 4) (1 5) (1 6) (1 7)
 (2 4) (2 5) (2 6) (2 7)
 (3 4) (3 5) (3 6) (3 7)
```

Errors

SYNTAX ERROR	The operator's right argument is not a dyadic function.

6.6 Catenation and Lamination

Syntax

```
      Z←L,R
      Z←L⍪R
      Z←L,[K]R
```

Definition

Either or both arguments may be nonsimple and heterogeneous.

Example

```
      (1 2 3) (4 5),(1 2 3) 'Y'
(1 2 3) (4 5) (1 2 3) Y
```

6.7 Dyadic Iota and Epsilon

Syntax

```
      Z←L⍳R
      Z←L∊R
```

Definition

The definitions of dyadic iota and epsilon are changed by replacing = with $\underline{=}$.

Discussion

Dyadic iota and epsilon compare their arguments at all levels of nesting. The result of each function is simple. The shape of the result is based on the shapes of the argument.

Examples

```
      A
(BARBARA) (BOB) (CARL) (GRAHAM) (JESSIE) (JOE) (JOHN) (LAURIE) (PAUL)
      (SCOTT)
      B
(JESSIE) (SCOTT) (FLOYD) (JUDY) (BOB) (JOE)
      A∊B
0 1 0 0 1 1 0 0 0 1
      A⍳B
5 10 11 11 2 6

     ∇ FUNCTION;⎕ELX;ERRORS;HANDLERS
[1]    ERRORS←'DOMAIN ERROR' 'LENGTH ERROR' 'RANK ERROR'
[2]    HANDLERS←DE LE RE OTHER ⍝ LABELS TO HANDLE ERRORS
[3]    ⎕ELX←' →HANDLERS[ERRORS⍳⊂(∧\⎕DM≠⎕TCNL)/⎕DM]'
.
.
.
```

6.8 Structural Functions

Syntax

```
LρR
ρR
L⍉R
⍉R
L/R
L\R
LΦR
ΦR
L↑R
L↓R
,R
```

Definition

The left argument must be simple; the right argument need not be. Each function applies at the outermost level only. Fill items for *LρR* *L/R* *L\R* and *L↑R* are given by ⊂⊤⊃*R*. (See Section 6.9 for more information on fill items.)

Examples

```
      2 3ρ(1 2) (3 4)
 (1 2) (3 4) (1 2)
 (3 4) (1 2) (3 4)
      1 0 1\(1 2) (3 4)
(1 2) (0 0) (3 4)
```

6.9 Fill Items

Definition

```
      ⊂⊤⊃R
or
      ⊤⍉ρR
```

Discussion

The functions which may require fill items are *L↑R* *⊃R* *L/R* *L\R* *L(A∘\)R* and *LρR*. The fill item is determined by the type, rank, and shape of the first item of the right argument.

For a nested right argument, the fill item can be arbitrarily complicated. For example,

```
      A
( (1  1  1) (2  2  2)
  (3  3  3) (4  4  4))

      3↑A
( (1  1  1) (2  2  2)  ( (0  0  0) (0  0  0)  ( (0  0  0) (0  0  0)
  (3  3  3) (4  4  4))   (0  0  0) (0  0  0))   (0  0  0) (0  0  0))
```

Even when an array with a nested first item has been emptied, the prototype information is kept and can be extracted with any of the functions listed above.

6.10 Functions Not Defined on Nested Arrays

Syntax

```
        L⊥R
        L⊤R
        L⌹R
        ⌹R
        ⍋R
        ⍒R
        L⍕R
        ⍕R
        L?R
```

All system functions.

Definition

These functions will not accept nested arrays as arguments.

Discussion

When called by operators, these functions will perform on simple items.

Example

```
      ⍋¨(1 3 4) (6 2 8) (6 6 6) (6 5 2)
(1 2 3) (2 1 3) (1 2 3) (3 2 1)
```

Errors

DOMAIN ERROR	An argument is not simple.

GLOSSARY

coercion

Relaxation of strict rules about the shapes of function arguments. This is done to simplify the job of the programmer. For instance, strictly speaking the left argument of reshape should always be a vector. It simpifies expressions greatly (with no loss in clarity) to allow a scalar left argument to be treated as a one-item vector.

derived function

The function produced by an operator sequence.

display potential

The attribute of a result which indicates whether or not it will be displayed automatically. For example, assignment returns a result with display potential off; the result is suitable for further computation but is not displayed.

fill items

The items inserted into arrays by the execution of certain functions such as expansion.

function argument

An array which is positioned so as to be the left or right argument of a function or derived function. For instance, in the expression +/*R*, the variable *R* is the function argument. Contrast with "operator argument".

heterogeneous

A simple array containing both numbers and characters.

item

A fundamental term which refers to an object of an array. This object may be nonscalar, nested, or both. The items of a simple array are all simple scalars.

nested

A term indicating that one or more items of an array or its prototype contain enclosed arrays rather than being composed entirely of simple scalars.

operator argument

An array or function which is positioned so as to be the left or right argument of an operator. For example, in the expression +/*R*, addition is the operator argument. Contrast with "function argument".

operator sequence

An operator and its arguments. For example plus-reduction is the derived function produced by the operator sequence +/.

prototype

The fill item for an array.

simple

A term indicating that all positions of an array or the prototype of an array contain only scalars that are not nested. An array can be simple but heterogeneous.

structure

The form of an array determined by the shape at all levels of nesting and the pattern of nesting.

valence

The number of arguments that a function or operator can accept. For example, a function with valence one is monadic.

REFERENCES

1. A Generalization of APL, J. A. Brown, Ph.D. Dissertation, 1971, School Computer and Information Science, Syracuse University, New York, U. S. Commerce Department Clearinghouse 74H004942 AD-770488./5.

2. "Evaluating Extensions to APL", J. A. Brown, APL79 Conference Proceedings, 1979.

3. Mask and Mesh, E. E. McDonnell, APL Workshop, 1980, Minnowbrook, New York.

4. "Nested Arrays: The Tool for the Future", Bob Smith, APL In Practice, Wiley, 1980.

5. "Nested Rectangular Arrays for Measures, Addresses, and Paths", Trenchard More, APL79 Conference Proceedings, 1979.

6. Operators and Functions, Kenneth E. Iverson, IBM Research Report No. 7091, Yorktown Heights, New York.

7. "The Role of Operators in APL", Kenneth E. Iverson, APL79 Conference Proceedings, 1979.

NESTED ARRAYS
REFERENCE MANUAL

We write our publications to help you. You can help us by filling out and returning FEEDBACK. Your reply will be forwarded to the person who can best benefit from your comments.

1. Is the writing straightforward and clear?	Yes ☐	No ☐
2. Is the material organized well?	Yes ☐	No ☐
3. Are the examples clear and well chosen?	Yes ☐	No ☐
4. Do the illustrations (figures and tables) clarify the material?	Yes ☐	No ☐
5. Do you find the size of the publication and the way it is bound convenient?	Yes ☐	No ☐
6. Are there errors, or misleading information, in the publication? If so, where do they occur?	Yes ☐	No ☐
7. Did you find any parts of the publication unduly hard to understand? If so, which are they?	Yes ☐	No ☐

How can we change this publication to make it more useful to you? (If you need more room, please write on the back.)

Thank you for your comments. No postage is necessary if mailed in the U.S.A.

CUT HERE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #10586 WASHINGTON, D.C.

Postage Will Be Paid By Addressee

STSC,Inc.
7316 Wisconsin Avenue
Bethesda, Maryland 20014

CUT HERE

FOLD

FOLD, TAPE, AND MAIL (DO NOT STAPLE).