APL2 Programming:

# Language Reference

SH21-1061-01

APL2 Programming:

# Language Reference

SH21-1061-01

**Note!**

Before using this information and the product it supports, be sure to read the general information under “Notices” on page x.

## Second Edition (February 1994)

This edition replaces and makes obsolete the previous edition, SH21-1061-0. The technical changes for this edition are summarized under “Summary of Changes,” and are indicated by a vertical bar to the left of a change.

This edition applies to:

- Release 2 of APL2/370 Version 2, Program Number 5688-228
- Release 2 of APL2/6000 Version 1, Program Number 5765-012
- Release 2 of APL2/PC Version 1, Program Numbers 5604-260 (EMEA) and 5799-PGG (USA)
- Release 1 of APL2 for Sun Solaris Version 1, Program Number 5648-065
- Release 1 of APL2/2 Advanced, Version 1.0, Part Number 89G1697
- Release 1 of APL2/2 Entry, Version 1.0, Part Number 89G1556

and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest edition of the applicable IBM system bibliography for current information on this product.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Department J58, P. O. Box 49023, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

# Contents

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Corporation, IBM Director of Licensing, 208 Harbor Drive, Stamford, Connecticut, United States 06904.

## Programming Interface Information

This reference is intended to help programmers code APL2 applications. This reference documents General-Use Programming Interface and Associated Guidance Information provided by APL2.

General-use programming interfaces allow the customer to write programs that obtain the services of APL2.

## Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
APL2
APL2/6000
AIX/6000
IBM
OS/2
System/370
System/390

The following terms, denoted by a double asterisk (**) in this publication, are trademarks of other companies:

| | |
|---|---|
| Solaris | Sun Microsystems, Inc. |
| Sun | Sun Microsystems, Inc. |

# About This Book

This book defines the IBM* APL2* nested array version of the APL language as supported on OS/2*, Sun** Solaris**, AIX/6000*, DOS, VM/CMS, and MVS/TSO. Deviations from the language, as defined in this book, are documented in the separate user's guides.

APL2 is used in such diverse applications as commercial data processing, system design, scientific computation, modeling, and the teaching of mathematics and other subjects. It has been particularly useful in database applications, where its computational power and communication facilities combine to enhance the productivity of both application programmers and other users.

For more information about APL2 and its history, see the *APL2 Programming: An Introduction to APL2*.

## Who Should Read This Book

This book can be used by all APL2 users, though some chapters do assume that the user has some familiarity with APL or APL2.

## APL2 Publications

Figure 1 lists the books in the APL2 library. This table shows the books and how they can help you with specific tasks.

*Figure 1 (Page 1 of 2). APL2 Publications*

| Information | Book | Publication Number |
|---|---|---|
| General product | *APL2 Fact Sheet* | GH21-1090 |
| Warranty | *APL2/370 Application Environment Licensed Program Specifications* | GH21-1063 |
| | *APL2/370 Licensed Program Specifications* | GH21-1070 |
| | *APL2 for AIX/6000 Licensed Program Specifications* | GC23-3058 |
| | *APL2 for Sun Solaris Licensed Program Specifications* | GC26-3359 |
| Introductory language material | *APL2 Programming: An Introduction to APL2* | SH21-1073 |
| Common reference material | *APL2 Programming: Language Reference* | SH21-1061 |
| | *APL2 Reference Summary* | SX26-3999 |

*Figure 1 (Page 2 of 2). APL2 Publications*

| Information | Book | Publication Number |
|---|---|---|
| System interface | *APL2/370 Programming: System Services Reference* | SH21-1056 |
| | *APL2/370 Programming: Using the Supplied Routines* | SH21-1054 |
| | *APL2/370 Programming: Processor Interface Reference* | SH21-1058 |
| | *APL2 for OS/2: User's Guide* | SH21-1091 |
| | *APL2 for Sun Solaris: User's Guide* | SH21-1092 |
| | *APL2 for AIX/6000: User's Guide* | SC23-3051 |
| | *APL2 GRAPHPAK: User's Guide and Reference* | SH21-1074 |
| | *APL2 Programming: Using Structured Query Language* | SH21-1057 |
| | *APL2 Migration Guide* | SH21-1069 |
| Mainframe system programming | *APL2/370 Installation and Customization under CMS* | SH21-1062 |
| | *APL2/370 Installation and Customization under TSO* | SH21-1055 |
| | *APL2/370 Messages and Codes* | SH21-1059 |
| | *APL2/370 Diagnosis Guide* | LY27-9601 |

For the titles and order numbers of other related publications, see the “Bibliography” on page 490.

## Conventions Used in This Book

As you use this publication, be aware of the following:

- Alphabetic APL2 characters are printed in capital italic letters.
- The symbol ⇔ or ↔ is used to mean “is equivalent to.” It is not an APL2 operation. The equal sign (=) is used to mean the APL2 equal function.
- In illustrations of syntax, the following arbitrary names are used:

| | |
|---|---|
| *L* | Left argument |
| *R* | Right argument |
| *F* | Function |
| *LO* | Left operand |
| *RO* | Right operand |
| *MOP* | Monadic operator |
| *DOP* | Dyadic operator |

- Unless explicitly stated, the default APL2 environment is assumed. The index origin (⎕*IO*) is 1; the printing precision (⎕*PP*) is 10; and the print width (⎕*PW*) is 79.
- In examples, user input is indented six spaces to simulate the APL2 six-blank prompt.
- To conserve space and make it easier to contrast examples, the examples are presented in two or three columns whenever possible. Read the first column of examples first, and then the second and third.
- The term *workstation* refers to all platforms where APL2 is implemented except those based on System/370* and System/390* architecture.

- APL2 implemented on System/370-based and System/390-based architecture is referred to as APL2/370.

# Summary of Changes

## Products

APL2/2, Version 1 Release 1
APL2 for Sun Solaris, Version 1 Release 1
APL2/6000*, Version 1 Release 2
APL2/370, Version 2 Release 2
APL2/PC, Version 1 Release 2

***Date of Publication:*** January 1994

***Form of Publication:*** Revision, SH21-1061-01

### Document Changes

- Added references to the workstation products
- Updated information on the display of characters
- Added mention of distinguished names
- Added diamond information
- Updated selective specification information
- Updated figure and list of axis specification conditions
- Updated precision section
- Added system tolerances for the workstations
- Updated binomial section
- Updated compress (from slash) section
- Updated compress with axis (from slash) section
- Updated drop example
- Updated matrix inverse example
- Corrected binomial identity value in reduce (from slash)
- Updated reverse example
- Updated table of system functions and variables
- Updated ⎕AV section
- Updated event type codes
- Updated ⎕EX syntax information
- Updated ⎕LX examples
- Updated ⎕NA examples
- Updated ⎕SVC example
- Added posting rules to ⎕SVC
- Updated ⎕SVO (inquire) example
- Updated ⎕SVO (set) examples
- Updated ⎕SVR example
- Added ⎕UCS universal character set
- Added shared variables chapter
- Updated information for APL2 editors
- Updated table of APL2 system commands
- Added )CHECK
- Updated )DROP information
- Updated )LOAD examples
- Updated character set figures
- Added table listing ASCII, EBCDIC, Unicode, and symbol equivalents
- Updated system limitations appendix

# Chapter 1. APL2 in Action

APL2 structures data into *arrays*. These data can include a mix of characters and numbers. By means of the *specification arrow* (←), an array can be associated with a name and the resulting *variable* can then be used in place of the array in computations.

Whereas arrays contain data, *functions* manipulate the structure of arrays or perform calculations on their data. Every *primitive function* name is a symbol. For example, ÷ is the name of the primitive function *divide*. *Operators* apply to functions or arrays, and produce functions called *derived functions*. Every *primitive operator* name is a symbol. For example, ¨ is the name of the primitive operator *each*.

You can write your own programs or subroutines (called *defined functions* and *defined operators*), tailoring APL2 to the specific needs of your application. You name defined functions and operators when you define them, using one or more alphanumeric characters.

Collectively, functions and operators are known as *operations*.

*System functions* and *system variables* provide information about, and permit interaction with, the APL2 system. Each system function and system variable is represented by a distinguished name that begins with the quad symbol (⎕).

Arrays, functions, and operators are the *objects* of APL2.

APL2 also provides a facility for using system services and other program products through auxiliary processors. These services are accessed through *shared variables* and can be under the control of an APL2 defined function. A shared variable, the interface between processors, is used to pass information between them. Any variable can be offered as a shared variable. “Shared Variables” on page 60 describes the system functions and variables used for sharing, and Chapter 8, “Shared Variables” on page 364 contains additional details on sharing. The APL2 auxiliary processors available are detailed in the workstation user's guides and *APL2/370 Programming*: *System Services Reference*.

## Interaction

During an APL2 session, you enter expressions for evaluation, run programs (defined functions and operators), enter system commands, and define functions and operators.

The form of your interaction with APL2 is a dialog. You make an entry, APL2 processes the entry and returns a response. Most of the time the cursor waits in the seventh column for input. Displayed output usually begins in the first column. Throughout this manual, examples follow this convention (unless otherwise noted), as shown in Figure 2.

```
      4+5 8 3 2           Input
9 12 7 6                  Output
```

*Figure 2. Input and Output of APL2 Expressions*

The visual distinction between input and output is useful when you study the results of your APL2 work.

Except when you use one of the APL2 editors to define a function or operator, your dialog takes place in *immediate execution* (or calculator) mode. In *definition mode*, you use one of the APL2 editors to enter programs built of APL2 statements. These programs can be stored for later execution.

## Workspaces

The common organizational unit of the APL2 system is the *workspace*. Part of each workspace is set aside to serve the internal workings of the system, and the remainder is used, as required, to store programs and pieces of (transient and permanent) information. When in use, a workspace is called *active*.

Only one workspace is active at a time. A copy of an inactive workspace can be made active, or selected information can be copied from one or more inactive workspaces into the active workspace. Inactive workspaces are stored in libraries.

*System commands* provide information about and manage data for workspaces and libraries. They are entered separately rather than as part of APL2 expressions. System commands begin with a right parenthesis.

Chapter 10, “System Commands” on page 413 contains more information about workspaces and how to manipulate them.

## Sample Use of APL2

The annotated examples shown in Figure 3 and Figure 4 on page 4 illustrate aspects of APL2 that are described in the remainder of this publication. The comments to the right of the APL2 expressions name the operation or facility being demonstrated and the page number of its description. Comments and page references refer to the first use of the operation or facility, not to each occurrence.

These examples assume that a shoe distributor has some basic inventory questions. Figure 3 answers questions about the quantities of shoes in stock. Figure 4 on page 4 answers questions concerning the cost of the shoes.

To simplify the example, only a few styles of shoes for men, women, and children are used; namely, oxfords, loafers, sneakers, sandals, and pumps. However, the expressions shown are applicable to larger quantities of data; for instance, styles can be kept by style number. Also, the examples show expressions only in immediate execution mode. In practice, most of the expressions would be incorporated into more generalized defined functions. For larger volumes of data, input and file read/write functions can be used.

*Figure 3. Expressions for Maintaining and Reporting Inventory Quantities*

**APL2 Expression** — **Comment**

For each group (men, women, children), enter the number of shoes of different styles in stock:

```
       MEN←45 75 15
       WOMEN←35 75 15 45 95
       CHILDREN←35 0 55 15
```

Specification of variables (39) and use of arrays (5).

Determine the total number of men's shoes:

```
       +/MEN
135
```

Slash (/) operator (209) with the add (+) function (65) as operand.

Determine totals for each group:

```
       +/¨ MEN WOMEN CHILDREN
135 265 105
```

Vector notation (14) and each (¨) operator (107) with derived function summation (+/) as operand.

Determine the total number of shoes in stock:

```
       +/∊MEN WOMEN CHILDREN
505
```

Enlist (∊) function (118).

Represent stock as a single variable:

```
       STOCK←MEN WOMEN CHILDREN
       STOCK
45 75 15  35 75 15 45 95  35 0 55 15
       +/¨STOCK
135 265 105
       +/∊STOCK
505
```

Nested array (8) and its display (19).

Display the inventory information as a table:

```
       ⊃[1]STOCK
45 35 35
75 75  0
15 15 55
 0 45 15
 0 95  0
```

Disclose with axis (⊃[ ]) function (96), which fills with zeros where data was not provided.

Describe what the numbers represent:

```
       STYLES←'OXFORDS' 'LOAFERS' 'SNEAKERS'
       STYLES←STYLES,'SANDALS' 'PUMPS'
       GROUPS←'MENS' 'WOMENS' 'CHILDRENS'
```

Catenate (,) function (74) and character data (13).

Add row and column headings to the table:

```
       (' ',GROUPS),[1]STYLES,⊃[1]STOCK
           MENS WOMENS CHILDRENS
OXFORDS     45     35        35
LOAFERS     75     75         0
SNEAKERS    15     15        55
SANDALS      0     45        15
PUMPS        0     95         0
```

Catenate with axis (,[ ]) function (77) and intermixed character and numeric data.

*Figure 4. Expressions for Maintaining and Reporting Inventory Costs*

**APL2 Expression** — **Comment**

Maintain costs for each style in each group:

```
      COSTS←(39 19 29) (35 15 29 18 45)
      COSTS←COSTS,⊂25 16 21 12.5
      COSTS
 39 19 29  35 15 29 18 45  25 16 21 12.5
```

Parentheses (36) and Enclose (⊂) function (111).

Note that the example demonstrates the use of enclose to catenate a single nested item to a nested vector. It is necessary here because the line width of the figure is not large enough to accommodate the entire specification of *COSTS* on one line.

What is the cost of men's loafers?

```
      1 2⊃COSTS
 19
```

Pick (⊃) function (195).

Change the cost of men's loafers:

```
      (1 2⊃COSTS)←20
      COSTS
 39 20 29  35 15 29 18 45  25 16 21 12.5
```

Selective specification (40).

Determine retail costs if markups from wholesale costs for men's, women's, and children's shoes are 60, 70, and 80 percent, respectively.

```
      PRICES←COSTS×1+.6 .7 .8
      ,[⍳0]PRICES
 62.4 32 46.4
 59.5 25.5 49.3 30.6 76.5
 45 28.8 37.8 22.5
```

Multiply (×) function (183), application of scalar function (51), and ravel with axis (202) to display the prices for each group on a separate line.

Identify the stock investment for each group:

```
      GROUPS,¨COSTS+.×¨STOCK
 MENS 3690  WOMENS 7870  CHILDRENS 2217.5
```

Array product (.) operator (165) with functions add and multiply as operands.

Determine the resulting net profit (total sales value minus total cost) for each line in stock:

```
      NET←STOCK×PRICES-COSTS
      ,[⍳0]NET
 1053 900 261
 857.5 787.5 304.5 567 2992.5
 700 0 924 150
```

Subtract (-) function (243).

Determine the net profit by group and the total net profit:

```
      +/¨NET
2214 5509 1774
      +/+/¨NET
9497
```

Identify the group and style that has the largest net profit:

```
      GROUP_STYLES←,(GROUPS,¨' ')∘.,STYLES
      (,⊃NET=⌈/∊NET)/GROUP_STYLES
 WOMENS PUMPS
```

Array product operator (deriving outer product) (186) and ravel (,) function (202).

Maximum (⌈) function (180) as operand to slash operator, slash operator (deriving replicate (220), equal (=) function (219), and disclose (⊃) function (94).

# Chapter 2. Arrays

APL2 manipulates collections of numbers, characters, or both as single objects. These collections are called *arrays*. Arrays have two properties: *structure* and *data*. The following sections:

- Explain and illustrate the structural properties
- Describe the types of data items
- Explain the construction of arrays
- Detail the display of arrays

## Structure

APL2 arrays are ordered rectangular collections of data *items*. There are three measures of an array's structure:

- Rank
- Shape
- Depth

### Rank

An array can have zero or more dimensions or *axes*. The number of axes that an array has is called its *rank*. Arrays can be called one-dimensional, two-dimensional, three-dimensional, and so forth, according to their rank. Figure 5 summarizes array structure by rank and gives sample arrays of various ranks. As the figure shows, arrays of rank 0, 1, and 2 have special names. Any array with a rank of two or greater is sometimes called a *multidimensional* array.

*Figure 5. Summary of Array Structures*

| **Rank** | **Name of Array** | **Description of Array** | **Example** |
|---|---|---|---|
| 0 | Scalar | One item arranged along no axes. | `4` |
| 1 | Vector | Zero or more items arranged along one axis. | `12 6 N 5` |
| 2 | Matrix | Zero or more items arranged along two axes. | `6 8 3 1`<br>`4 A 5 9`<br>`W X Y Z` |
| 3 or more (as many as the system limit) | *no special name* | Zero or more items arranged along *n* axes. | `9 2 3`<br>`G 7 Q`<br><br>`5 8 1`<br>`4 5 T` |

***Axes:*** The last axis of an array of rank 2 or greater is called the *column* axis. The next-to-last axis of an array of rank 2 or greater is called the *row* axis. There are no established terms for the axes of arrays of rank 3 or more, although sometimes the first axis of a three-dimensional array is called a *page* or a *plane*.

## Shape

Each axis of an array contains zero or more items. The vector containing the number of items along each axis is called the *shape vector* of the array. For example, the shape vector of a 3-row by 4-column matrix `M` is `3 4`. The typical way of expressing this is to say that the shape of `M` is `3 4`.

The first item of the shape vector is the *length* or *size* of the first axis, the second item of the shape vector is the length of the second axis, and so forth. The number of items in an array is the product of the lengths of the axes. Thus, a 3-row, 4-column matrix contains 12 items (`3×4`). And a two-page, two-row, three-column array also contains 12 items (`2×2×3`). Figure 5 on page 5 shows examples of these two arrays.

The shape function (`ρ`), discussed on page 241, can be used to find the shape and rank of an array.

***Empty Arrays:*** If the length along one or more axes is 0, the array is *empty* and the number of items in the array is 0. An empty array has a rank of 1 or greater, because a scalar has no axes and therefore cannot have an axis of length 0. Chapter 4, “General Information” on page 46 describes the effects of applying operations to empty arrays and Chapter 7, “Defined Functions and Operators” on page 345 explains further uses of empty arrays.

### Rectangularity

All APL2 arrays are rectangular—even scalars and vectors. *Rectangularity* in APL2 arrays means that the position of an item along any axis is independent of its position along the other axes. Thus, in a matrix, for example, every row has the same length.

An item in an array is located by naming its position along each axis. For example, in the 3 by 4 matrix `MAT`, shown below, each item is located by naming first its position along the rows and then its position along the columns. In the example, the positions appear as subscripts on each item of `MAT`.

$$\begin{array}{llll} A_{1,1} & B_{1,2} & C_{1,3} & D_{1,4} \\ E_{2,1} & F_{2,2} & G_{2,3} & H_{2,4} \\ I_{3,1} & J_{3,2} & K_{3,3} & L_{3,4} \end{array}$$

Positional notation of an item in an array is called the *index* of the item. The index consists of an ordered set of integers, each of which describes the position of the item along the corresponding axis. An index composed of subscripts {*2,4*}, for example, locates the item in the second row, fourth column of a matrix. (In the matrix `MAT`, this is the item `H`.)

In APL2, the index of an item can be denoted with square brackets surrounding the index value. Semicolons separate the positions along each axis. For example, the item `H` in the matrix `MAT` is selected by index as `MAT[2;4]`. (Bracket index is fully described in “`[ ]` Bracket Index” on page 70.)

**Row-Major Order:** Selecting items from an array in *row-major* order means selecting them row by row and from left to right within the row. For example, the ravel function (`,`) makes any array a vector by selecting its items in row-major order and structuring them as a vector.

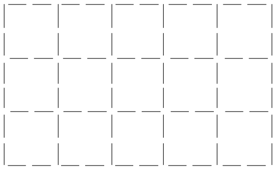
```
      M←3 4⍴32 14 6 21 8 ¯7 12 3 ¯9 42 27 18
      M
32 14  6 21
 8 ¯7 12  3
¯9 42 27 18

      ,M
32 14 6 21 8 ¯7 12 3 ¯9 42 27 18
```

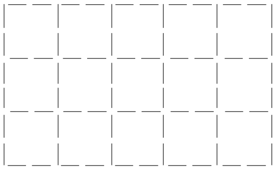
## Subarrays

An array with each of its items contained in another array is a *subarray* of that array. For example, the matrix *A* is a subarray of the matrix *B*. The shaded items of *B* are found in *A*, and no item of *A* is not in *B*.

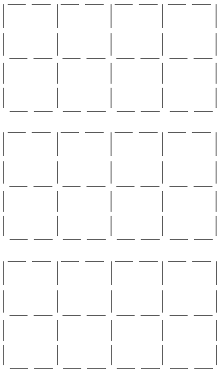
| A | B |
|---|---|
| ` 2  4`<br>`10 12` | `1  2  3  4`<br>`5  6  7  8`<br>`9 10 11 12` |

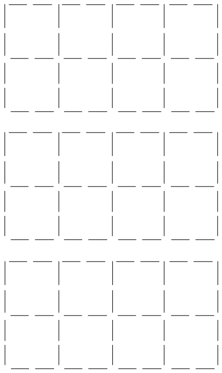
If the subarray includes all items along one or more axes of an array, it is a *contiguous subarray* of that array. For example, the shaded portion of the 3 by 5 array shown on the left below is a contiguous column subarray because it contains all the row positions. The shaded portion of the array on the right is a contiguous row subarray.





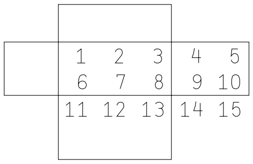
In the 3 by 2 by 4 array on the left below, the shaded portion is a contiguous page subarray because it includes all row and column positions. In the 3 by 2 by 4 array on the right, the shaded portion is a contiguous row subarray because it includes all page and column positions.





The concept of contiguous subarrays is important in understanding the application of such functions as take, drop, grade up, and reverse. For example, take (↑) yields the intersection of contiguous subarrays selected along each axis of the right argument. The left argument defines the number of subarrays to select along each axis.

```
      M← 3 5⍴ ⍳15
      M
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15

      2 3↑M
 1  2  3
 6  7  8
```



**Note:** *M* could have been made up of characters or other arrays. However, the result would still be the first three columns of the first two rows of *M*.

## Depth

An item of an array is itself an array. If every item in the array is a simple scalar (a single number or a single character), the array is called a *simple* array. If one or more items in the array is not a simple scalar, the array is called a *nested* array. In the examples below, for instance, the vector *S* on the left is a simple vector with three items, each of which is a single number. The vector *T* on the right is a three-item vector with two vector items and one item (`4`) that is a simple scalar.

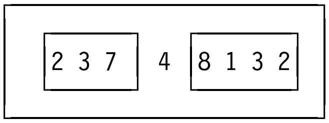
**Simple Vector**

```
     S←2 3 7
     S
2 3 7
```

**Nested Vector**

```
     T←(2 3 7) 4 (8 1 3 2)
     T
 2 3 7  4  8 1 3 2
```

In the nested vector example, the first item of *T* is the vector `2 3 7`, the second item of *T* is the scalar `4`, and the third item of *T* is the vector `8 1 3 2`. The illustration below shows each vector item of *T* enclosed within a box. The outer box represents the vector *T* in its entirety.



The degree of nesting of an array is called *depth*. A simple scalar has a depth of `0`. The simple vector *S* has a depth of 1. This means that all its items are simple scalars, that is, either single numbers or single characters. The depth of *T* is 2. A depth of `2` means that at least one of its items has a depth of 1.

The following indicates the depths an array can have and gives the meaning of each depth:

| Depth | Meaning |
|---|---|
| 0 | Simple scalar. |
| 1 | Simple, nonscalar array (vector, matrix, or n-dimensional array) containing only simple scalars as items. |
| 2 | Array that contains at least one array of depth 1. It contains no items with depths greater than 1. |
| *n* | Array that contains, as an item, at least one array of depth n-1. For example, an array of depth 6 contains at least one array of depth 5. It may contain other arrays of lesser depth as well. |

The depth function (≡) (discussed in "≡ Depth" on page 91) shows the depth of an array.

The matrix *M*, below, shows the use of nested arrays to add headings to a table and to substitute `'NONE'` for items whose value is 0. The matrix has five rows and three columns. Each item in the first row is a character vector, and each item in the first column is a character vector. *NONE* in the last row, last column, is also a character vector. The depth of *M* is `2`.

```
      M
FOOD  CALORIES PROTEIN
MILK       160       9
APPLE       60       1
BREAD       75       2
JELLY       50    NONE
```

## Picture of an Array's Structure

Functions that illustrate the structure of an array are contained in the *DISPLAY* workspace, one of the workspaces located in a public library and distributed with the APL2 Program Product. This workspace is described in the appropriate workstation user's guide or in *APL2/370 Programming*: *Using the Supplied Routines* for CMS and TSO.

The result of the functions is a series of boxes that surround the array and its items. To illustrate the array's structure, a simple scalar is shown as its value. The top and left borders of the box display symbols that indicate the rank of the array. If no symbol appears in either the top or left border, the array is a scalar. Information in the lower border indicates either the data type of the array or that the array is nested. The symbols are defined below:

| Symbol | Meaning |
|---|---|
| → | Vector (at least) |
| ↓ | Matrix or multidimensional array |
| ⊖ or ⌽ | Empty array along this axis |
| ~ | Numeric array |
| + | Mixed character and numeric data in array |
| *no symbol* | Character data |
| ∊ | Nested array |
| _ | Scalar blank (also appears under a simple scalar character when the array contains a nonscalar array) |

When a path is traced from outside the display to an item, the number of lines crossed is the depth of the item.

The examples below show the use of `DISPLAY`. The first example shows a nested array of depth 2, and the second example shows a nested array of depth 3.

```
      DISPLAY 1 2 'MORE' (3 'A') (2 2ρι4) 'B'
.→-------------------------.
|     .→---. .→--. .→--.   |
| 1 2 |MORE| |3 A| ↓1 2| B |
|     '----' '+--' |3 4| - |
|                  '~--'   |
'∊-------------------------'

      DISPLAY 1 2 'MORE' (3 'OR') (2 2ρι4) 'B'
.→------------------------------.
|     .→---. .→-------. .→--.   |
| 1 2 |MORE| |   .→-. | ↓1 2| B |
|     '----' | 3 |OR| | |3 4| - |
|            |   '--' | '~--'   |
|            '∊-------'         |
'∊------------------------------'
```

## Data

Data enters the active workspace by:

- Explicit entry at the display device
- Execution of APL2 functions and operators
- External functions
- Names associated with files
- Use of shared variables, system variables, system functions, and system commands

An array can be composed of numbers or characters or a mixture of numbers and characters. This section describes the characteristics and display of each type of data.

## Numeric Data

All numbers are entered and displayed in decimal representation (base-10). Numbers smaller or larger than the system limit cannot be used. (For the system limits, see Appendix C, “System Limitations for APL2” on page 489.)

Numeric data is complex—both real and nonreal numbers. Complex numbers are numbers of the form $a+bi$, where $i$ is the square root of `¯1`. A number is *real* if $b$ is 0. A number is *nonreal* if $b$ is not 0.

### Real Numbers

Real numbers have the following attributes:

| Attribute | Meaning |
|---|---|
| Boolean | Zero or one |
| Integer | Nonfractional numbers, including zero and one |
| Rational | Fractional numbers and integers |

The irrational numbers pi (π) and *e* are available through the functions pi times (see "`○` Pi Times" on page 194) and exponential (see "`*` Exponential" on page 127) as rational approximations to the extent of the numeric precision of the system. Other irrational numbers, such as the square root of 2, are also available as approximations through the application of certain computational functions.

Real numbers can be entered and displayed in either conventional form (including a decimal point, if appropriate) or scaled form. In conventional form, the number twenty-five, for example, is represented as `25` and the number four and three-tenths is represented as `4.3`. In scaled form, the number one million is represented as `1E6`.

***Scaled Form:*** The scaled form of a number, which is also sometimes called the exponential or scientific form, consists of three consecutive parts:

1. An integer or decimal fraction called the mantissa or multiplier.
2. The letter `E`, which can be read "times `10` to the power...."
3. An integer called the scale, which must not include a decimal point. The scale specifies the power of 10 by which the mantissa is multiplied.

For example:

```
      2.4578E6
2457800
```

```
      5.278912467E11
5.278912467E11
```

***Negative Numbers:*** Negative numbers are represented by an overbar (`¯`) immediately preceding the number. In scaled form, the multiplier and the scale may both be negative. For example:

```
      ¯253
¯253
```

```
      ¯5.1575E¯3
¯0.0051575
```

Note that the overbar (`¯`) used to start a negative numeric constant differs from the bar (`-`) that denotes the subtract and negative functions.

## Complex Numbers

Complex number constants can be represented in three forms, the last two of which are polar notations:

1. Real and imaginary part separated by the letter `J` and no spaces. The number is real if the imaginary part is `0`.
2. Magnitude and angle in degrees separated by the letter `D` and no spaces. The number is real if the angle is an integral multiple of 180.
3. Magnitude and angle in radians separated by the letter `R` and no spaces. The number is real if the angle is an integral multiple of pi (π).

APL2 displays complex numbers in *J* notation, even though they can be entered in any of the three forms. Defined functions *FMTPR* and *FMTPD* in the workspace *MATHFNS* distributed with APL2 are available to display complex numbers in *R* and *D* notation, respectively.

A nonreal number that has no real part is called an *imaginary number*. The imaginary number *i* (the square root of `¯1`) can be written as:

```
0J1
1D90
1R1.5707963267948965
```

Either or both parts of a complex number constant can be specified in scaled form. For example, `1.2E5J¯4E¯4` is the same as `120000J¯.0004`, and `8E3D1E2` is the same as `8000D100`.

***Display of Numbers:*** Numbers can be entered in any of the forms discussed above. The default display of numbers is governed by the printing precision (`⎕PP`, see "`⎕PP` Printing Precision" on page 315); possibly by the nature of other items in the same array column (see "Display of Arrays" on page 17); and for numbers with absolute values between 0 and 1, by the relationship of leading fractional zeros to the number of significant digits.

The format by specification (`⍕`) function, discussed on page 143, can be used to specify the form in which a numeric array is displayed.

***Leading and Trailing Zeros:*** Leading zeros to the left of a decimal point and trailing zeros to the right of a decimal point are not displayed. A single zero before a decimal point is not considered a leading zero.

```
      .5
0.5
      0004
4
```

```
      .50000
0.5
      4.560000
4.56
```

```
      000.2
0.2
      00.0123000
0.0123
```

***Display Precision:*** The system variable `⎕PP` (printing precision) controls the precision with which numbers are displayed. The default value is 10 digits.

```
      2.718
2.718
```

```
      3.141592653589793
3.141592654
```

The precision with which numbers are stored internally is always the maximum that the implementation permits. When `⎕PP` is set to its maximum, all available precision is displayed.

***Display in Scaled Form:*** A number is displayed in scaled form when:

- The number of leading fractional zeros is greater than `5`.
- The number of digits in the integer portion of a number exceeds `⎕PP`. However, if the number is stored internally as an integer (`(4 ⎕AT N)[2]` is less than `8`), `⎕PP` is ignored and the number is not displayed in scaled form.

```
      456789              │        ⎕PP←10
456789                    │        .0000005678
      ⎕PP←4               │  5.678E¯7
      456789              │
456789                    │
      456789.0            │
4.568E5                   │
```

In scaled form, except for `0`, the absolute value of the mantissa is greater than or equal to `1` but less than `10`. In scaled form, `0` is represented as `0E0`.

```
      467.34589E9         │        ¯456.179345E¯9
4.6734589E11              │  ¯4.56179345E¯7
```

***Display of Complex Numbers:*** Nonreal numbers are displayed in `J` notation regardless of the notation in which they are entered. Although real numbers can be entered in complex notation, they are always displayed in conventional form. For example:

```
      2J0                 │        2J3
2                         │  2J3
```

In `J` notation, the real or imaginary part is not displayed if it is less than the other by more than `⎕PP` orders of magnitude (unless `⎕PP` is at its maximum). For example:

```
      2J3E45              │        3E45J2
0J3E45                    │  3E45
```

## Character Data

Character constants are created by entering a character from the keyboard within a pair of single quotation marks. These surrounding quotation marks are not displayed on output. Their purpose is to identify an item as character data. For example:

```
      'A'      │       'TIME'  │       '⍒∆○'
A              │ TIME          │ ⍒∆○
```

The leftmost example shows a single character, even though three-print positions are necessary to create it. Likewise, `⍒` in the last example is one character, even though an overstrike combination (`∇`, backspace, `|`) is required to create it on some display devices.

Appendix A, “The APL2 Character Set” on page 470, discusses the APL2 character set.

The single quotation mark character itself must be entered as a pair of single quotation marks (without an intervening space) in a character constant:

```
      ''''
'
```

```
      'CAN''T'
CAN'T
```

### Display of Characters

Characters are displayed exactly as they are entered, but without the surrounding quotation marks and without double internal quotation marks. Blanks within quotation marks are retained, and each blank is an item.

```
      '3+4'
3+4
```

```
      'ONE TWO   THREE'
ONE TWO   THREE
```

**Note:** Some characters are control characters and can cause unpredictable results when displayed on certain devices.

**Note:** Characters that are not in `⎕AV` cannot be displayed on most devices and are shown as the omega (ω) character. For example:

```
      ⎕AF 257
ω
```

## Construction of Arrays

Creating scalars and vectors of two or more items requires entry of only the data that make up the items. Creating matrixes, arrays of higher rank, and zero- and one-item vectors requires the use of functions.

## Vector Notation

The juxtaposition of two or more arrays in an expression results in a vector whose items are the arrays. Representing a vector in this manner is called *vector notation*. Each of the following simple vectors is created by juxtaposing simple scalars:

```
      3 4 5 6
3 4 5 6
      'F' 'A' 'C' 'E'
FACE
```

```
      2 6 'D' 4 'W'
2 6 D 4 W
      X←6
      2 3 X 36
2 3 6 36
```

***Simple Vector:*** For a simple vector, either blanks or parentheses must separate the items, unless a character item is adjacent to a numeric item or a name. Although permitted, more than one consecutive blank or set of parentheses is redundant. The following numeric vectors are equivalent:

```
    1 2 ←→ 1(2) ←→ (1)2 ←→ (1)(2) ←→ 1     2
```

The following mixed vectors are equivalent.

```
2 'X' 8 ←→ 2'X'8 ←→ 2('X')8 ←→ (2('X')(8))
```

Characters in a vector consisting only of characters can be listed within one set of single quotation marks:

```
  'FACE' ←→ ('F' 'A' 'C' 'E')
```

**Note:** When quotation marks surround each character, a space must separate the characters.

```
      'F' 'A' 'C' 'E'
FACE
```

```
      'F''A''C''E'
F'A'C'E
```

Without the space, the inner quotation marks are interpreted as a quotation mark character rather than as a character delimiter.

***Nested Vector:*** Forming a nested vector with vector notation requires grouping the items of the vector and separating the groups by parentheses or quotation marks—or the names used must represent nonsimple scalars. Each of the following expressions yields a three-item nested vector:

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

      'RED' 'WHITE' 'BLUE'
 RED WHITE BLUE

      (9 7 4)'BOX'(7 'F' 9 'G')
 9 7 4  BOX  7 F 9 G

      ('UP' 'UP')'AND' 'AWAY'
  UP UP   AND AWAY

      V←3 5 6
      'O' V 'X'
 O  3 5 6  X
```

The fourth example has a nested first item–the vector `UP UP`. The last example is nested because the name `V` represents a vector. (See “Parentheses” on page 36 for more information about the use of parentheses in expressions.)

Vector notation cannot be used to construct a zero-item or one-item nested vector because no juxtaposition takes place, and parentheses or quotation marks, if used, do not both group and separate. You can construct a zero-item array by using reshape (ρ) or one-item nested array by using the function enclose (⊂), as discussed in the next section.

## Using Functions to Create Arrays

To create multidimensional arrays or to create vectors of zero or one item, you can use a function. Methods of creating them include:

- Reshaping another array
- Joining arrays
- Selecting from an array
- Using table operations

The following examples illustrate array creation using some of the functions described in Chapter 5, “Primitive Functions and Operators” on page 62.

### Reshaping Another Array

```
      2 3⍴2 6 1 5 8 7
2 6 1
5 8 7

      3 2 4⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
ABCD
EFGH

IJKL
MNOP

QRST
UVWX
```

### Joining Arrays

```
      2 4 6,[.5]8 10 12
2  4  6
8 10 12

      'ABCD',[1.1]'WXYZ'
AW
BX
CY
DZ
```

### Selecting from an Array

```
      V←'ABCDEF'
      V[2 2⍴⍳4]
AB
CD
```

### Using Table Operations

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

Although each of these examples shows the creation of a simple multidimensional array, the functions apply in the same way when the data items are not single numbers or characters. For example:

```
      2 3⍴'ONE' 'TWO'
 ONE TWO ONE
 TWO ONE TWO
```

You can apply the function enclose (⊂), page 111, to create a scalar from an array that is not a scalar. Using a nested scalar may be necessary in the construction of certain arrays. Compare:

```
      2 4⍴'APL2'
APL2
APL2
```

```
      2 4⍴⊂'APL2'
 APL2 APL2 APL2 APL2
 APL2 APL2 APL2 APL2
```

In the example on the left, `'APL2'` is a simple four-item vector. In the example on the right, `⊂'APL2'` is a scalar array. The results of the 2 4 reshape of these arguments are quite different.

## Display of Arrays

“Display of Numbers” on page 12 and “Display of Characters” on page 14 discuss the default display of numeric and character items. This section discusses the default display of arrays. It assumes that the printing width of the display device is wide enough to accommodate each line of data to be displayed. `⎕PW` (printing width), page 318, discusses the display of lines wider than the printing width.

### Simple Scalars and Vectors

Simple scalars and vectors are displayed in a single line. If an item in a simple vector is a number, it is separated from adjacent items by one blank. Simple characters are not separated from other simple characters. For example:

```
      4 6 23 7
4 6 23 7
```

```
      0 '*' 3 4 '∘' 'A' 9
0 * 3 4 ∘A 9
```

If the single line is wider than the printing width (`⎕PW`), the line is continued on the next physical line and is indented six spaces.

### Simple Matrixes and Other Multidimensional Arrays

The displays of simple arrays are not indented. A simple matrix is displayed in a rectangular plane. All items in a given column of a simple matrix are displayed in the same format, but the columns themselves can have different formats and different widths.

If a column in a simple matrix contains a number, that column is separated from adjacent columns by at least one blank. For example:

```
      2 5⍴'*' '⎕' '∆' 123 45 '○' '∇' 6 7 8
*⎕ ∆ 123 45
○∇ 6   7  8
```

Simple multidimensional arrays are displayed in rectangular planes. Planes of a three-dimensional array are displayed with an intervening blank line. For higher dimensional arrays, each successive plane is separated by an additional line. For example:

```
      2 2 2 3ρ1 2 3 4 5 6
1 2 3
4 5 6
         One line separates the planes
1 2 3
4 5 6
         Two lines separate successive planes

1 2 3
4 5 6

1 2 3
4 5 6
```

If a column in a simple multidimensional array contains a number, that column is separated from adjacent columns in all planes by one blank. All items in corresponding columns of the planes are displayed in the same format.

Simple matrixes and other multidimensional arrays containing numbers that require scaled form are displayed with all items in a column in the same format.

```
      ⎕PP←4
      2 2ρ23 .0056 34566.0 .00000056
2.300E1 5.6E¯3
3.457E4 5.6E¯7
```

Decimal points, the *E* in scaled notation, and the *J* for complex numbers align in columns. The columns are formatted independently. For example:

```
      2 4ρ1 12.3 345 6J7 .1 .12 1J2 16J6
1   12.3  345    6J7
0.1  0.12   1J2 16J6
```

Some simple arrays containing nonreal numbers may be displayed in a form not suitable for input. The separator *J* in each column is aligned at the possible cost of separating paired real and imaginary parts, as in the first three columns of the matrix shown below.

```
      4 4ρ0 1 2J3 4J5.6 7.8J9
0       1       2  J3 4J5.6
7.8J9   0       1     2J3
4  J5.6 7.8J9   0     1
2  J3   4  J5.6 7.8J9 0
```

## Nested Arrays

The displays of nested arrays (and nested items within an array) are indented one space, and they also include a trailing blank. In each of the examples below, the first display shows the array as it is displayed. In the second display, a caret indicates each displayed blank.

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

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

Character vectors and scalar items in a column that contains numeric items are right-justified.

```
      4 2⍴'ONE' 'TWO' 1111 22 ¯4 5 7'?'

  ONE TWO            │  ^^ONE^TWO^
 1111  22            │  ^1111^^22^
   ¯4    5           │  ^^^¯4^^^^5^
    7   ?            │  ^^^^7^^^?^
```

Character scalars or vectors in a column that contains no numbers are left-justified:

```
      3 3⍴'ONE' 1111 22 'TWO' ¯4 5 'THREE' 7 '?'


 ONE   1111 22       │  ^ONE^^^1111^22^
 TWO     ¯4  5       │  ^TWO^^^^^¯4^^5^
 THREE    7  ?       │  ^THREE^^^^7^^?^
```

Other nested arrays are presented in a display that contains embedded blanks according to the structures of the adjacent items. The number of embedded blanks is one fewer for character items than for other items.

```
      3 2⍴1 2 3,(⊂4 5 6),7,⊂⊂8 9


 1      2            │  ^1^^^^^^2^
 3  4 5 6            │  ^3^^4^5^6^
 7   8 9             │  ^7^^^8^9^^
```

The default format function (page 135) yields a simple character array whose appearance may be the same as the display of its argument. (If they are different, it is because of ⎕*PW*.)

You can use the functions in the *DISPLAY* workspace distributed with APL2 (as discussed in “Picture of an Array's Structure” on page 9) to illustrate an arrays structure.

# Chapter 3. Syntax and Expressions

The rules for combining arrays with functions and functions with operators define the *syntax* of APL2. This chapter contains:

- A summary of syntax and the evaluation of expressions.
- The details of APL2 syntax
- The syntactical evaluation of expressions

## Summary of Syntax and Evaluation of Expressions

The following figures summarize the rules discussed in this chapter.

**NAMES**

Names for variables and for defined functions and operators are character strings, consisting of one or more of the following:

| | |
|---|---|
| **First character** | *A*...*Z*, *a*...*z*, ⍙ or ∆ |
| **Other characters** | Same as first character *plus* 0...9, ¯, and _ |

Names cannot begin with *S*∆ or *T*∆, which are reserved for stop control and trace control.

Names for system functions and system variables are called *distinguished names*. Except for beginning with a ⎕, they follow the same rules as other names.

For more information, see “Names” on page 24.

**EVALUATION OF EXPRESSIONS**

All functions execute according to their position within an expression. The rightmost function whose arguments are available is evaluated first.

For more information, see “Evaluating Expressions” on page 32.

## NAME AND SYMBOL BINDING

Binding strengths of arguments, operands, and syntactic construction symbols supplement the function evaluation rule. Binding defines how names and symbols group for evaluation.

The hierarchy of binding strengths in descending order is:

| Binding Strength | What Is Bound |
|---|---|
| Brackets | Brackets to what is on their left |
| Specification left | Left arrow to what is on its left |
| Right operand | Dyadic operator to its right operand |
| Vector | Array to an array |
| Left operand | Operator to its left operand |
| Left argument | Function to its left argument |
| Right argument | Function to its right argument |
| Specification right | Left arrow to what is on its right |

For binding, the branch arrow behaves as a monadic function. Brackets and monadic operators have no binding strength on the right.

For more information, see “Name and Symbol Binding” on page 33.

## PARENTHESES

Parentheses are used for grouping and for changing the default binding. They are correct if properly paired, and if the content within evaluates to an array, a function, or an operator.

Parentheses are redundant when:

- They group a single name (primitive or constructed).
- They group an expression already within parentheses.
- In an array expression, they:
  - Do not both group and separate.
  - Group the right argument of a function.
  - Group the vector left argument (written in vector notation) of an expression.
- In a function expression, they:
  - Group the left operand of an operator.
  - Group the function expression, and a left parenthesis does not separate two arrays.

For more information, see “Parentheses” on page 36.

**VECTOR NOTATION**

A vector can be created by juxtaposing two or more arrays in an expression. The items of a vector are arrays. If all items are simple scalars, the vector is simple. If at least one item is not a simple scalar, the vector is nested. For example, the three-item vector `5 'H' 3` is simple, and the three-item vector `(2 5 1) 5 (2 8)` is nested.

For more information, see "Vector Notation" on page 14.

**SPACES**

Spaces or parentheses are needed to separate constructed names if not separating them produces a different name. Spaces or parentheses are also needed to separate constructed names from other symbols, if not separating them produces an invalid name. For example, `3 F 4` requires spaces because `3F` is an invalid name and `F4` is a different name.

Spaces are *not* needed to separate primitive operations from their arguments or operands, or to separate a primitive operation from a defined operation. Redundant spaces are permitted.

For more information, see "Syntactically Valid Expressions" on page 28.

**VALENCE**

All functions are ambi-valent (can be monadic or dyadic), and the definition used in any instance is determined only by context.

Operators are not ambi-valent. A given operator is either monadic or dyadic, determined by definition, not context.

For more information, see "Determining Function Valence" on page 33.

## APL2 Syntax

The discussion on APL2 syntax covers:

- Arrays
- Functions and their relationship to their arguments
- Operators and their relationship to their operands

## Functions

Functions apply to arrays and produce arrays as a result. The *arguments* of functions are the arrays that functions manipulate. A function may have one or two arguments. If the function is a defined function, it can have no arguments or can be defined to take either one or two arguments. The number of arguments that a function takes is called its *valence*. The following terms can be used to describe the valence of a function:

| **Term** | **Valence** |
|---|---|
| Niladic Function | No arguments |
| Monadic Function | One argument |
| Dyadic Function | Two arguments |

A function that can take either one or two arguments is *ambi-valent*. For example:

```
      -16 22
¯16 ¯22
```

```
      20-16 22
4 ¯2
```

If a function is monadic, the function name is placed to the *left* of the argument, as in the following examples for the shape (ρ), depth (≡), and factorial (!) functions:

```
      ρ3 5 7 9
      ≡'ONE' 'TWO'
      !5
```

If a function is dyadic, the function name is placed between the arguments, as in the following examples for the divide (÷), pick (⊃), and rotate (⌽) functions:

```
      4÷5
      3⊃'RED' 'WHITE' 'AND' 'BLUE'
      1⌽'SPIN'
```

### Axis Specification

Some primitive functions can be applied along an indicated axis of an argument array. This application is called *axis specification*. The indicated axis is enclosed in brackets and appears to the immediate right of the function symbol. The following examples of axis specification use the 3-row, 4-column matrix *M*:

```
      M←3 4ρι12
      M
1  2  3  4
5  6  7  8
9 10 11 12
```

To append another row:

```
      M,[1]13 14 15 16
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

To add .3 to the items in the first row, .2 to the items in the second row, and .1 to the items in the third row:

```
      .3 .2 .1+[1]M
1.3  2.3  3.3  4.3
5.2  6.2  7.2  8.2
9.1 10.1 11.1 12.1
```

See also “Conditions for Axis Specification” on page 45.

## Operators

Operators take functions or arrays as *operands* and produce functions (derived functions) as a result. An operator can be monadic or dyadic; that is, it can take one or two operands. Operators are never ambi-valent, but their derived functions are.

If an operator is monadic, the operator name is placed to the *right* of the operand, as in the following examples for the slash (`/`) and each (`¨`) operators:

```
         +/
         ρ¨
```

If an operator is dyadic, the operator name is placed between the operands, as in the following example for the array product (`.`) operator:

```
         +.×
         ∘.×
```

In this context, the jot symbol (`∘`) is treated syntactically as a function.

The derived function resulting from applying an operator can be monadic or dyadic. Its valence does not depend on the valence of the operator from which it is derived. For example, slash is a monadic operator. The derived function summation is monadic, and the derived function n-wise summation is dyadic:

| | |
|---|---|
| Summation | `+/1 2 3 4` |
| n-wise summation | `2+/1 2 3 4` |

The operators `/` and `\` can be applied along an indicated axis. The indicated axis is enclosed in brackets and appears to the immediate right of the operator. For example, for the 3-row, 4-column matrix *M* shown on page 23:

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

```
      1 0 2 1/[2]M
1  3  3  4
5  7  7  8
9 11 11 12
```

## Names

Names are symbolic representations of APL2 objects—arrays, functions, and operators. Some names are always associated with the same object. These are called *primitive names*. Other names may be associated with different objects at different times. These are called *constructed names*. The rules for writing expressions, detailed later in this section, explain how to combine names.

Names beginning with the character ⎕ are called distinguished names and are assigned fixed meanings. For more information about distinguished names, see Chapter 6, “System Functions and Variables” on page 259.

## Primitive Names

Primitive names are those that are defined as part of the definition of APL2. A given primitive name is always associated with the same object. The primitive names in APL2 are numbers, characters, and the primitive function and operator symbols. For example, each of the names listed below is a single primitive name, even though the last two occupy more than one print position.

| **Name** | **Meaning** |
|---|---|
| `⌽ ⊖` | function name for rotate and reverse |
| `¨` | operator name for each |
| `24.5` | number |
| `'R'` | character |

Note that APL2 also includes special syntactic symbols that are not names. These are `∘ ' [ ] ( ) ← → ; ◊ : ⍝`. Their uses are discussed in “Expressions” on page 27.

## Constructed Names

There are two types of constructed names:

- Names for arrays, defined functions, defined operators, and labels
- Distinguished names (prefixed by ⎕) for system functions and system variables

Constructed names receive values by being associated with APL2 objects. Using a valid name that is not associated with an object results in a *VALUE ERROR*.

## Rules for Constructed Names

Names are constructed of one or more characters with the following constraints:

- Initial (or only) character is from the set:

  `ABC...XYZ∆`

  `abc...xyz⍙`

- Remaining characters (if any) are from the set:

  `ABC...XYZ∆`

  `abc...xyz⍙`

  `0123456789_¯`

- Certain compatibility settings in APL2/370 permit, or default to, the use of underbarred uppercase letters instead of lowercase letters. For more information see *APL2/370 Programming: System Services Reference*.

- The combinations `S∆` and `T∆` cannot be used as the first two characters of a name. They are reserved for stop control and trace control, respectively.

Any name constructed according to these rules is valid. The following examples show some valid and invalid names:

**Valid Names**

M
STOCK
AVERAGE
LOW_BID
REGION2
R2D2

**Invalid Names**

| | |
|---|---|
| SET UP | Contains a space |
| 3PRIME | Begins with a number |
| ¯ABCDE | Begins with an overbar |

***Associating Names with Objects:*** A constructed name has no value (it is not associated with a defined function, a defined operator, an array, or a label) until some action is taken to specify the association.

Names become associated with arrays through the use of the specification arrow (←), through parameter substitution when a defined function or operator is invoked, or through the use of dyadic ⎕NA. A *variable* is a constructed name that is associated with an array.

*Labels* are used in defined functions or operators to identify the target of branching. A label is associated with the line number in the body of a defined function or operation in which it appears.

Defined functions and operators are associated with user names as an implicit result of the ⎕FX function, or through the use of dyadic ⎕NA. Functions may also be associated with user names through parameter substitution in a defined operator. Editors, system commands, and facilities outside the language can also associate names with objects.

## Distinguished Names

Distinguished names are character strings reserved for fixed uses in the language. Distinguished names follow the rules for names except that they begin with the character ⎕ (quad). Distinguished names associated with arrays are called *system variables*; those associated with functions are called *system functions*. System variables and system functions help manage the active workspace and APL2 facilities and environment; for example, ⎕IO (index origin) and ⎕PP (print precision).

Although any distinguished name constructed according to the rules for names is valid, only a few are associated with objects. Specifying a distinguished name that does not represent a system function or system variable generates a SYNTAX ERROR.

## Syntactic Construction Symbols

The syntactic construction symbols and their uses are listed below. The roles these symbols play in the evaluation of expressions are discussed later in the chapter.

**Brackets** [ ]
Positioned to the right of an array name, indicate indexing; to the right of a function or operator name, indicate axis specification.

**Branch or escape arrow** →
Followed by an expression, indicates the next line, if any, in a defined function or operator to be executed. Alone, clears the state indicator of a suspended operation and its entire calling sequence. (Branching is fully discussed in “Branching” on page 349.)

**Jot** ∘
Acts as a placeholder for the left operand of outer product.

**Parentheses** ( )
Used for grouping; expressions within parentheses are evaluated first.

**Quotation mark** '
Delimits a character string.

**Semicolon** ;
Within brackets, separates the indexes along each axis. In the header of a defined operation, separates list of local names from each other and from operation syntax.

**Diamond** ◊
Separates multiple expressions that appear in a single line.

**Specification or assignment arrow** ←
Associates a name with an array, or modifies the values of selected positions in an array already associated with a name.

## Expressions

An *expression* consists of primitive and/or constructed names and possibly one or more syntactic construction symbols. For example:

```
5÷7
LIST←'PETER' 'PAUL' 'MARY'
+/[1]3 2ρ8 9 3 2 5 7
```

When an expression containing a function or a derived function and its argument(s) is evaluated, the result is an array. Such expressions—and array primitives— are called *array expressions*. For example:

```
      15
15
      4ρ2
2 2 2 2
```

```
      ι10
1 2 3 4 5 6 7 8 9 10
      ×/2 7 5
70
```

An expression containing only a function or derived function and no arguments is called a *function expression*, and an expression containing only an operator and no operands is called an *operator expression*. Function and operator expressions can be evaluated only in the context of an array expression. Although valid, they generate a *SYNTAX ERROR* if entered independently. (Note that niladic functions are treated syntactically like arrays, not functions.)

## Statements

A *statement* is an executable unit of work. It is made up of three parts, as shown below. Any of the three parts can be omitted but, if included, they must appear in the order shown below.

label **:** expressions ⍝ comment

where:

| | |
|---|---|
| **:** | (colon) Separates a label from the rest of the line. |
| expressions | Can be**:**<br>• No expression<br>• One expression<br>• More than one expression separated by diamonds (◊) |
| ⍝ | (comment) Separates the expressions from explanatory information. Any spaces between the end of the expression and the ⍝ are retained. |

For example:

```
TEST:→(X<0)/L1   ⍝CHECK FOR NEGATIVE X

TRY:

⍝FUNCTION REPLACES 0'S WITH '-'

4+9×7

A←1 ◊ B←2
```

## Syntactically Valid Expressions

The rules for writing syntactically valid expressions are few. They concern vector notation, placement of operations, syntactic construction symbols, and spaces. An expression written following these rules does not generate a *SYNTAX ERROR*, although it can generate some other error. The quick reference section at the front of this manual summarizes the rules.

***Vector Notation:*** You can create a vector by writing its array items separated by spaces, parentheses, and/or quotation marks. (See also “Vector Notation” on page 14.)

***Placement of Operations:*** The following rules apply:

- A dyadic function or operator name is written between its arguments or operands.
- A monadic function name is written to the left of its argument.
- A monadic operator name is written to the right of its operand.

***Syntactic Construction Symbols:*** The following rules apply to the use of syntactic construction symbols.

- Parentheses, quotation marks, and brackets must be matched.

  Examples - Valid:
  ```
  (A B C)[2]
  (A≠B)\[1]MAT
  'DON''T'
  ```

  Examples - Invalid:

  | | |
  |---|---|
  | `(K<N/TRT` | Missing right parenthesis |
  | `'CAN'T'` | Missing single quotation mark |

- Parentheses are permitted around array, function, and operator expressions. They must not split a name or group functions or operators.

  Examples - Valid:
  ```
  (4-7)÷6
  (⌈/)C
  ```

  Examples - Invalid:

  | | |
  |---|---|
  | `Q-(.÷)R` | Splits derived function |
  | `NA(ME)` | Is not the same as `NAME` |
  | `A(⍳⍳)B` | Groups functions |
  | `,(¨/)1 2 3` | Groups operators |

  (See also “Parentheses” on page 36.)

- The expression to the right of a specification arrow must be an array. The syntactic object to the left of the specification arrow can be the name of an array, or a name not associated with an object, a list of names, or an expression that selects positions from an existing named array.

  Examples:
  ```
  D←⍳10
  M[2;3]←4
  (J K)←3 4
  (2⊃X)←⊂'NEW'
  ```

  (See also “Specification of Variables” on page 39.)

- A branch arrow either must be the leftmost symbol or must be to the immediate right of a label. Any expression to the right of the branch must be an array expression.

  Examples:
  ```
     →(M>0)/POSITIVE
     L:→(M>0)/POSITIVE
     →
  ```

- Semicolons are allowed only within brackets.

  Example - Valid:
  ```
  MAT[1 4;5 2 6]
  ```

  Example - Invalid:

  | | |
  |---|---|
  | `'TOTAL ';345` | Semicolon is not within brackets |

- Semicolons are used in the headers of defined operations but the header is not an APL expression.

- A colon is allowed only following a name (a label) that is the leftmost name on the line.

  Example:
  `POSITIVE:Z←!M`

***Spaces:*** Spaces are needed to separate constructed names from other symbols if not separating them produces an invalid name.

Examples - Valid:

```
3 FN⍳5
3. FN⍳5
```

Examples - Invalid:

```
3FN⍳5
3.FN⍳5
```

Spaces are *not* needed to separate primitive operations from their arguments or operands, or from a defined operation.

```
D⌈T ↔ D ⌈ T ↔ D⌈ T ↔ D    ⌈    T
```

Note that a space is also needed to maintain the meaning of adjacent constructed names if they are not enclosed in parentheses. However, no `SYNTAX ERROR` is generated if the space is omitted. For instance, `AB CD` is not the same as `ABCD`, nor is `'IN' 'OUT'` the same as `'IN''OUT'`. But:

```
AB CD ↔ (AB)(CD) ↔ (AB)CD ↔ AB(CD)

'IN' 'OUT' ↔ ('IN')('OUT') ↔ ('IN')'OUT'
```

***Redundant Spaces and Parentheses:*** Redundant spaces and parentheses are permitted and, in fact, are often employed to make an expression more readable. Redundant spaces do not change the meaning of an expression and they produce no errors.

Examples:

```
4 - ¯9 ↔ 4-¯9

8 1 4 3(+DOP÷)9 2 4 2 ↔ 8 1 4 3+DOP÷9 2 4 2
```

(See also “Redundant Parentheses” on page 37.)

## Defined Functions and Operators

The syntax of defined functions and operators is illustrated below, using the following arbitrarily chosen names:

| | |
|---|---|
| *Z* | Result name |
| *F* | Function name |
| *L* | Left argument name |
| *R* | Right argument name |
| *MOP* | Monadic operator name |
| *DOP* | Dyadic operator name |
| *LO* | Left operand name |
| *RO* | Right operand name |

| **Forms with Explicit Result** | **Forms without Explicit Result** |
|---|---|
| 1. `Z←F R` | 8. `F R` |
| 2. `Z←L F R` | 9. `L F R` |
| 3. `Z←(LO MOP) R` | 10. `(LO MOP) R` |
| 4. `Z←L (LO MOP) R` | 11. `L (LO MOP) R` |
| 5. `Z←(LO DOP RO) R` | 12. `(LO DOP RO) R` |
| 6. `Z←L (LO DOP RO) R` | 13. `L (LO DOP RO) R` |
| 7. `Z←F` | 14. `F` |

Forms 1 and 2 are defined functions syntactically equivalent to primitive functions. They can be substituted wherever a primitive function is used. Forms 3 through 6 are operators syntactically equivalent to primitive operators. Note that these forms show the arguments of the derived functions.

All syntactic rules apply to defined functions and defined operators with explicit results in the same way that they apply to primitive operations. For example, in the following expression the dyadic function *COMPOUND* is placed between its arguments. The array result of *COMPOUND* is the right argument of the monadic function *ROUND*.

```
        YEAREND←ROUND .12 COMPOUND 10000
```

A niladic defined function with explicit result (form 7) behaves syntactically as a variable and can be used in the same way as a variable except that it cannot be used to the immediate left of a specification arrow. For example:

```
        ∇Z←PI
[1]     Z←○1∇

        PI×3*2    ⍝ AREA OF A CIRCLE WITH RADIUS 3
```

Forms 8 through 14 do not include an explicit result. They constitute *valueless expressions*. Defined operations without explicit results must be the leftmost operation in the expression and cannot be enclosed in parentheses.

## System Functions and System Variables

System functions behave syntactically like primitive functions, and system variables behave syntactically like variables.

## Evaluating Expressions

Expressions can be syntactically correct, yet fail to evaluate. Syntactically correct APL2 expressions can give unexpected results or can generate errors other than *SYNTAX ERROR*. For example:

- *VALUE ERROR* is given if a constructed name that has not been associated with an object is used.
- *LENGTH ERROR* or *RANK ERROR* is given if the arguments are not conformable.
- *DOMAIN ERROR* is given if the function is not defined for the type of argument entered.

Error messages are described in Chapter 11, "Interpreter Messages" on page 461.

## Expressions with More Than One Function and No Operators

When an expression contains only one function and its argument(s), a syntactically correct expression is evaluated in only one way. The function, if any, is applied to its argument(s) to yield the result. However, when an expression is written containing more than one function, a rule is needed to determine which is to be evaluated first. For instance, is multiplication or addition applied first in the following expression:

```
2×3+4
```

The evaluation of this expression—and others that contain more than one function and no operators— follows the basic APL2 evaluation rule:

> **All functions execute according to their position in an expression. The rightmost function whose arguments are available is evaluated first.**

This rule is often called the *right-to-left* rule. Because of the right-to-left rule, addition in the expression `2×3+4` is executed first and then multiplication:

```
      2×3+4
14
```

If `2×3` were parenthesized, the expression within parentheses would be the left argument of `+`. It must be evaluated first; then its value would be available as the left argument of `+`:

```
      (2×3)+4
10
```

This explains the rule that the rightmost function *whose arguments are available* is evaluated first.

## Determining Function Valence

All functions are syntactically ambi-valent. They can take one or two arguments. The context in which a function appears determines whether the monadic or dyadic definition is used. If the object to its left is an array, the function is dyadic. If the object to its left is a function or operator or derived function expression, it is monadic. For example, `⍳` in the following expression is monadic (the interval function) because the name to its left is a function name:

```
      10×⍳10
10 20 30 40 50 60 70 80 90 100
```

In the next expression, however, `⍳` is dyadic (the function index of) because the name to its left is an array:

```
      8 9 7⍳5 9 6 8
4 2 4 1
```

Even when the same name is used both monadically and dyadically in an expression, its meaning is unambiguous. For example:

```
      9 4 6⍳⍳6
4 4 4 2 4 3
```

The rightmost `⍳` is monadic because the name to its left is a function. The next `⍳` is dyadic because the name to its left is an array. Its right argument is the array `⍳6`.

If a right parenthesis `)` is to the left of a function, the subexpression within parentheses must be evaluated before you can determine whether the function is monadic or dyadic.

***Function with Only Either a Monadic or Dyadic Definition:*** Some functions have only a monadic definition, for example, execute (`⍎`). Some functions have only a dyadic definition, for example, the relational functions. If these functions are entered with the wrong number of arguments, a `VALENCE ERROR` is generated, not a `SYNTAX ERROR`. It is always syntactically correct to write a function with one or two arguments.

## Name and Symbol Binding

The right-to-left rule is the fundamental evaluation rule of APL2. However, it does not cover all situations, such as when an array is written in vector notation or when an expression contains operators and syntactic construction symbols. To cover all situations, a rule of *binding strengths* supplements the right-to-left rule. Binding defines how names and symbols group for evaluation. Given three names (or symbols), binding strength determines if the center one is associated with the name (or symbol) on the left or the right.

The hierarchy of binding strengths is listed below in descending order.

| **Binding Strength** | **What Is Bound** |
|---|---|
| Brackets | Brackets to what is on their left |
| Specification left | Left arrow to what is on its left |
| Right operand | Dyadic operator to its right operand |
| Vector | Array to an array |
| Left operand | Operator to its left operand |
| Left argument | Function to its left argument |
| Right argument | Function to its right argument |
| Specification right | Left arrow to what is on its right |

For binding, the branch arrow behaves as a monadic function. Brackets and monadic operators have no binding strength on the right. Parentheses, discussed on page 36, change the default binding.

## Brackets

Brackets indicate indexing if the object to their left is an array; brackets indicate axis specification if the object to their left is a function or operator.

Brackets have the highest binding strength. If an expression contains brackets, the brackets bind first to the object on their left before any other binding occurs. For example, the following expression is a three-item vector whose first item is `A[1]`, whose second item is `B[2]`, and whose third item is `C[3]`:

```
      A←'HAT'
      B←4 8 10
      C←'W' 2 'X' 7

      A[1] B[2] C[3]
H 8 X
```

In contrast, the following expression is a three-item vector whose first item is `A`, whose second item is `B`, and whose third item is `C[2]`:

```
      A B C[2]
 HAT  4 8 10  2
```

**Note:** The expression `7 6 9[2]` generates a `RANK ERROR` because the brackets bind to the `9` only. To select the second item of a vector, use parentheses:

```
      (A B C)[2]
 4 8 10
      (7 6 9)[2]
6
```

Finally, in the following expression the brackets bind to the `/` to produce a new monadic operator, which binds to + as its operand:

```
      D←3 2ρ⍳6
      +/[1]D
9 12
```

## Specification—Left and Right

The specification arrow binds to the name or the expression naming array positions on its *left*. That is, A B←C ←→ A C. The expression A B←C has the side effect of assigning to the name B the value of C. For example:

```
      A←2 3 4
      B←8
      C←'NEW'

      A B←C
 2 3 4  NEW

      B
NEW

      A[3]←'∇'
      A
2 3 ∇
```

The entire expression to the *right* of the specification arrow is an array. That is, the expression is evaluated before the assignment is made. Therefore, specification right has the least binding strength.

## Right Operand and Left Operand

The right operand of a dyadic operator is the function or array to its immediate right.

(**Note:** No primitive dyadic operators take an array right operand.)

For example, the function expression +.×/ is a reduction by a +.× inner product because the × binds as right operand to the array product operator (.), and *not* as left operand to the slash operator (/). The + binds as left operand to the dot; then the resulting product binds to the slash as its left operand.

+.×/ ←→ (+.×)/ **not** +.(×/)

There is no binding between operators. In the expression ,¨/, catenate binds as left operand to the each operator, and then the derived function ,¨ binds as left operand to the slash operator.

,¨/ ←→ (,¨)/

## Vector Written with Vector Notation

When two arrays are written next to each other in vector notation, there is a binding between them. This binding is called *vector binding*.

Vector binding is stronger than the binding of a function to its arguments. Thus, the expression 2 3+4 5 yields a two-item vector 6 8, *not* the three-item vector 2 7 5. Parentheses can be used to override the default binding. So 2 (3+4) 5, for instance, yields the vector 2 7 5.

Vector binding is also stronger than the left operand of an operator. Thus, the expression 2 1 3/ (which yields the derived function replicate) replicates the subarrays of its argument 2, 1, and 3 times, respectively. It does *not* form a three-item vector whose first two items are `2 1` and whose last item is formed by the 3-replication of the argument of 3/.

`2 1 3/A ←→ (2 1 3)/A` **not** `2 1 (3/A)`

Vector binding is *not* stronger than right operand binding. This is important for defined dyadic operators, which may take an array right operand (none of the primitive dyadic operators takes an array right operand). For example, if *DOP* is a defined dyadic operator and *LO* is a function:

`LO DOP A B ←→ (LO DOP A) B` **not** `LO DOP (A B)`

### Left Argument and Right Argument

Stating that left argument binding is stronger than right argument binding is another way of stating that the evaluation of the expression begins with the rightmost function whose arguments are available.

For example, in the following expression, `3` is bound as the left argument of + rather than as the right argument of ×.

`2×3+4`

## Multiple Expressions in a Line

The diamond separator allows multiple APL expressions to appear in a single line. The expressions are processed from left to right. For example:

`A←B ◊ B←B+1`

First *A* is assigned the value of *B*, then *B* is assigned the value of *B* plus `1`.

## Parentheses

Parentheses are used for grouping and changing the default binding. They can be used anywhere as long as they are properly paired and what is inside the pair evaluates to an array, a function, or an operator. An expression within parentheses or one that validly can be put within parentheses is called a *subexpression*. Valid subexpressions always return explicit results. If an expression that returns no result is parenthesized, a *VALUE ERROR* occurs.

To evaluate an expression containing parentheses, evaluate the subexpression within the parentheses, substitute for the parenthesized expression the value it produces, remove the parentheses, and continue the evaluation of the expression. For example:

| | |
|---|---|
| `(9-4)÷25` | Evaluate `(9-4)` |
| `(5)÷25` | Remove the parentheses |
| `5÷25` | Continue the evaluation |
| `0.2` | Result |

If an expression within parentheses contains an expression within parentheses, the rightmost function whose arguments are available is the first evaluated. For example:

| | |
|---|---|
| `((6<10)∧6≥0)/'DIGIT'` | Evaluate `6≥0` |
| `((6<10)∧1)/'DIGIT'` | Evaluate `(6<10)` |
| `((1)∧1)/'DIGIT'` | Remove inner parentheses |
| `(1∧1)/'DIGIT'` | Evaluate `(1∧1)` |
| `(1)/'DIGIT'` | Remove parentheses |
| `1/'DIGIT'` | Continue the evaluation |
| `DIGIT` | Result |

## Redundant Parentheses

Some parentheses that are correct can be removed from an expression without affecting the result of the expression, because they do not change either the binding of the names or the syntactic construction symbols of the expression. These are called *redundant parentheses*.

Parentheses surrounding a primitive or constructed name, a character string (enclosed in quotation marks), or an already parenthesized expression are always redundant. For example:

| | |
|---|---|
| `2(+)3 ↔ 2+3` | Primitive function name |
| `A+(.)×B ↔ A+.×B` | Primitive operator name |
| `(2)+1 ↔ 2+1` | Primitive array name |
| `(A)←3 ↔ A←3` | Constructed array name |
| `('ABC') ↔ 'ABC'` | Character string |
| `((2-3))+1 ↔ (2-3)+1` | Parenthesized expression |

Redundant parentheses may be added to or removed from expressions freely without changing the value of the expression. Additional guidelines for removing parentheses are given below.

***Vector Expressions in Parentheses:*** In expressions of arrays, parentheses that do not separate and group are redundant.

Examples—Redundant Parentheses:

| | |
|---|---|
| `2 (3) 4 ↔ 2 3 4` | These separate but do not group. |
| `(2 3 4) ↔ 2 3 4` | These group but do not separate. |

Examples—Nonredundant Parentheses:

`'H' (2 2⍴⍳4)` is not the same as `'H' 2 2⍴⍳4` (which is an error). The parentheses separate.

`2 (3 4)` is not the same as `2 3 4`. The parentheses group and separate to create a two-item nested vector. (See also “Vector Notation” on page 14.)

***Array Expressions in Parentheses:*** Parentheses in an expression alter the default binding of arguments to functions. For instance, to subtract 3 from 8 and then divide the result by 4:

```
      (8-3)÷4
1.25
```

Enclosing `8-3` in parentheses causes `3` and `-` to be bound even though the left argument binding of `÷` is stronger than the right argument binding of `-`.

Parentheses in array expressions are redundant if they group the right argument of a function, a vector left argument of a function, or brackets to the array immediately to their left. For example:

| | |
|---|---|
| `8-(3÷4) ←→ 8-3÷4` | Groups right argument |
| `(2 3)×4 ←→ 2 3×4` | Groups vector left argument |
| `A B (C[2]) ←→ A B C[2]` | Groups brackets to array on their left |

***Function Expressions with Parentheses:*** Parentheses in an expression alter the default binding of operands to operators. For instance, to express an outer product where the function applied is an inner product:

`∘.(+.×)`

Enclosing `+.×` in parentheses causes `+` to be bound to the dot on its right even though the right operand binding of the leftmost dot is stronger.

Parentheses in function expressions are redundant if they group the left operand of an operator or if the left parentheses does not separate two arrays. For example:

| | |
|---|---|
| `(+.×)/ ←→ +.×/` | Groups left operand |
| `A(+.×)B ←→ A+.×B` | Groups function expression |

***Operator Expressions with Parentheses:*** In any syntactically valid operator expression, parentheses are redundant. For example:

| | |
|---|---|
| `+(.)× ←→ +.×` | Surrounds operator name |

# Specification of Variables

Specification or assignment is one way that an array associates with a name. For example:

```
      A←'ONE' 'TWO'        │       B←⍳10
      A                    │       B
 ONE TWO                   │  1 2 3 4 5 6 7 8 9 10
```

The explicit result of specification is the array on the right. This result does not produce a display but is available for further computation. To see the value of the variable, enter its name. The name, once *specified* or *set*, represents the array and can be used in place of the data in APL2 expressions.

```
      100,B←⍳10
100 1 2 3 4 5 6 7 8 9 10
```

An attempt to assign a value to a function, operator, or primitive name generates a `SYNTAX ERROR`.

When an expression containing a variable is evaluated, the value of the variable is substituted for the name before the function or operator is executed. For example, with `A` as specified above:

```
      2⊃A               Substitute the value of A
TWO
      2⊃'ONE' 'TWO'     Evaluate the expression
TWO
```

## Using a Variable

Use of a variable name without a specification arrow to its immediate right is a *reference* or *use* of the variable.

```
      D←7              │       C←⍳5
      D                │       C
7                      │ 1 2 3 4 5
                       │
      D×2              │       ⌽C
14                     │ 5 4 3 2 1
      D                │       C
7                      │ 1 2 3 4 5
```

## Respecifying a Variable

When the variable name appears to the immediate left of the specification arrow, a new value is assigned to it:

```
      D←7              │       C←⍳5
      D                │       C
7                      │ 1 2 3 4 5
                       │
      D←D×2            │       C←⌽C
      D                │       C
14                     │ 5 4 3 2 1
```

## Multiple Specification

Several variables can be assigned on one line; for example, the expression below initializes each of the variables *E*, *F*, *G*, and *H* with the value of `1`.

```
      E←F←G←H←1
      E
1
      F
1
      G
1
      H
1
```

## Vector Specification

Several variables can be given values from items of a vector.

```
      (A B C)←2 3 4
      A
2
      B
3
```

If a scalar is on the right, the item in the scalar is assigned to each name.

```
      (A B)←0
      A
0
      B
0
      (A B)←⊂4 5 6
      A
4 5 6
      ⍴B
3
```

The list of names must be variables or names with no value. On some platforms, shared variables, system variables, or external variables are not permitted in the list.

## Selective Specification

**Note:** The information in this section is based on the APL2 language definition. Deviations exist on some platforms and are documented in the separate user's guides.

Any expression that selects values from an array can be written on the left of an assignment arrow to mean replacement of those values. Such replacement is called a *selective specification*.

*Selective specification* replaces selected items of an array. In selective specification, an array expression using one of the functions listed in Figure 6 on page 41 appears to the left of the specification arrow. The items in the positions selected by the array expression are replaced by the items to the right of the specification arrow.

**Monadic Functions**

| | |
|---|---|
| (∊R)←N | Enlist |
| (↑R)←N | First |
| (,R)←N | Ravel |
| (,[X]R)←N | Ravel with axis |
| (⌽R)←N or (⊖R)←N | Reverse |
| (⌽[X]R)←N or (⊖[X]R)←N | Reverse with axis |
| (⍉R)←N | Transpose (reversed axes) |

**Dyadic Functions**

| | |
|---|---|
| R[L]←N | Bracket indexing |
| (L↓R)←N | Drop |
| (L↓[X]R)←N | Drop with axis |
| (L⌷R)←N | Index |
| (L⌷[X]R)←N | Index with axis |
| (L⊃R)←N | Pick |
| (L⍴R)←N | Reshape |
| (L⌽R)←N or (L⊖R)←N | Rotate |
| (L⌽[X]R)←N or (L⊖[X]R)←N | Rotate with axis |
| (L↑R)←N | Take |
| (L↑[X]R)←N | Take with axis |
| (L⍉R)←N | Transpose (general) |

**Derived Functions**

| | |
|---|---|
| (LO\R)←N or (LO⍀R)←N | Expand |
| (LO\[X]R)←N or (LO⍀[X]R)←N | Expand wit axis |
| (LO/R)←N or (LO⌿R)←N | Replicate |
| (LO/[X]R)←N or (LO⌿[X]R)←N | Replicate with axis |
| (LO¨ R)←N | Each (monadic) |
| (L LO¨ R)←N | Each (dyadic) |

---

**Notes:**

1. R is the name of the array being selectively specified.
2. N is the array of new items for R.
3. X is a scalar or vector indication of axes in R.
4. L and LO are simple integer arrays.
5. Parentheses are necessary for all functions but bracket indexing.
6. For pick (⊃), only one item may be selectively specified at a time.

*Figure 6. Selective Specification Functions*

*Selective specification* is used to replace whole arrays or subsets of arrays. When a whole array is replaced, the structure of the replaced array is not relevant. When a subset of an array is replaced, the shape of the replaced array does not change but the structure of the items replaced is not relevant.

In ordinary cases, *selective specification* can be understood if you understand how the selection expression works when it is not on the left of an assignment. For example:

```
      V← 10 20 30 40
      (2↑V)←100 200
      V
100 200 30 40
```

The function take does not select the first two items of V; instead, it selects the locations of the first two items of V. This resulting vector of locations is considered a simple vector even if the items at those locations are deeply nested. The data on the right of the assignment then replaces data at those locations.

As with ordinary specification, the explicit result of a selective assignment is the array on the right that does not produce a display but is available for further computation.

More complicated cases can be tricky because the selection does not operate on the values in an array but rather on the positions of values.

Any selection expression begins by identifying an array whose value will be modified. Initially, the whole array is subject to replacement. Functions in the selection expression serve to limit the part of the array that is actually modified:

1. The rightmost name in the expression, ignoring brackets used for indexing, is the name whose value is set or altered. Call it the assigned name. The whole array named is subject to modification.

   If no function appears in the selection expression, then the value on the right of the left arrow becomes the value of the assigned name and *selective specification* degenerates into ordinary *specification*:

   ```
         A←'ABCD'
         (A)←10 20 30
         A
   10 20 30
   ```

   Thus, in some sense, *specification* is a special case of *selective specification*.

   If any functions appear in the selection expression, then the name being assigned must have a value.

2. Pick with an empty left argument is the only function that returns the whole array to which it is applied. Thus, pick with an empty left argument as the only function in a selection expression causes the whole array associated with the assigned name to be replaced.

   ```
         A←'ABCD'
         ((⍳0)⊃A)←10 20 30
         A
   10 20 30
   ```

   This is equivalent to a *specification* except that the assigned name must have a value.

3. First selects the whole array that is the first item of its right argument.

   ```
         A←'ABCD'
         (↑A)←10 20 30
   ```

```
     A
10 20 30   BCD
```

4. Pick selects the whole array that is at the end of a specified path through its right argument.

```
      A←'ABCD'
      (2⊃A)←10 20 30
      A
A  10 20 30  CD
```

5. Any selection function other than first or pick selects a subset of an array.

```
      A←(2 3)(2 3 4 5 )(10 20)
      (3↑2⊃A)←'ABC'
      A
2 3  ABC 5  10 20

      B←(2 3)(2 3 4 5)(10 20)
      (∊B)←0
      B
0 0  0 0 0 0  0 0
```

Once the selection expression has been evaluated, the following rules govern the replacement of values. Apply the first rule that holds:

1. If the left is a whole array, the right array replaces it.
2. If the right is a scalar (or an array with empty shape when ones are removed), then the right is paired with each item from the left and these rules are applied recursively.
3. If the left and the right have the same shape (when ones in the shapes are ignored), then corresponding items from the left and from the right are paired and these rules apply recursively.

While any expression following the above rules is a legal assignment, not all are currently supported. The following restrictions apply:

1. The result of the select expression must be simple. Given that the structure of the items selected is ignored, the only way the result of the selection expression can be nested is if some function that increases depth is applied (for example, enclose, partition, or some operator expressions) and this structure is not removed (for example, by enlist).
2. Disclose is not supported.

For additional restrictions, see the appropriate workstation user's guide.

Various selections and replacements are shown below for the matrix *M*. The examples assume that each selective specification expression uses the *original* specification of *M*.

```
      M←3 4⍴'ABCDEFGHIJKL'
      M
ABCD
EFGH
IJKL

      M[2;3]←'⎕'
      M
ABCD
EF⎕H
IJKL
      (2 1↑M)←'⊖⌹'
      M
⊖BCD
⌹FGH
IJKL
      (4↑,⍉M)←'○*÷⎕'
      M
○⎕CD
*FGH
÷JKL
```

```
      M[1;2 4]←'∇○'
      M
A∇C○
EFGH
IJKL

      (,M)←⍳12
      M
1  2  3  4
5  6  7  8
9 10 11 12
      M[1 3;1 4]←'*'
      M
*BC*
EFGH
*JK*
```

The last example in the left column demonstrates the application of several functions in selective specification. The positions replaced were the first four taken in row-major order after *M* was transposed (its rows and columns interchanged). These are the characters *AEIB*, which are then replaced with the ○*÷⎕, respectively.

The last example in the right column shows that scalars being selectively assigned to a nonscalar array of locations are replicated as necessary.

The value of the variable being altered by a selective specification cannot be replaced to effect before the specification is complete.

```
      A←⍳10
      ((A←2)↑A)←0
      A
0 0 3 4 5 6 7 8 9 10
```

If *B* is a shared variable, then

```
      ((B=' ')/B)←'*'
```

is an error because the leftmost mention of *B* is a reference of the shared variable and causes *B* to receive a new value.

For each function that permits selective specification, the description in Chapter 5, “Primitive Functions and Operators” on page 62 shows examples of the function applied in selective specification.

## Conditions for Axis Specification

| | |
|---|---|
| **Functions** | |
| + | Add |
| ∧ | And |
| ! | Binomial |
| ○ | Circular |
| ⊃ | Disclose |
| ÷ | Divide |
| ↓ | Drop |
| ⊂ | Enclose |
| = | Equal |
| > | Greater than |
| ≥ | Greater than or equal |
| < | Less than |
| ≤ | Less than or equal |
| ⍟ | Logarithm |
| ⌈ | Maximum |
| ⌊ | Minimum |
| × | Multiply |
| ⍲ | Nand |
| ⍱ | Nor |
| ≠ | Not equal |
| ∨ | Or |
| ⊂ | Partition |
| * | Power |
| \| | Residue |
| , | Ravel, Catenate, Laminate |
| ⌽ or ⊖ | Reverse, Rotate |
| ⌷ | Index |
| - | Subtract |
| ↑ | Take |
| **Operators** | |
| \ or ⍀ | Backslash |
| / or ⌿ | Slash |

*Figure 7. Functions and Operators That Allow Axis Specification*

For axis specification, writing brackets next to a function or operator is always syntactically correct, but evaluation of the related function succeeds only when the following specific conditions are true:

- The bracket expression contains no semicolons
- The data in brackets is the proper type
- The data in brackets is the proper rank
- The function or operator is one of those shown in Figure 7
- The data in brackets is within the range defined by the function or operator

Otherwise, an `AXIS ERROR` occurs.

# Chapter 4. General Information

The topics discussed in this chapter pertain to functions, operators, variables, and commands in general. They are discussed here because they affect the entire system and not just a single function or variable.

The descriptions of the APL2 functions, operators, variables, and commands require an understanding of the following topics:

- Types and prototypes
- Fill items
- Empty arrays
- Scalar and nonscalar functions
- Fill functions
- System effects on evaluation
- Errors and interrupts in immediate execution
- Shared variables

## Type and Prototype

**Note:** The information in this section is based on the APL2 language definition. Deviations exist on some platforms and are documented in the separate user's guides.

The *type* of array yields a zero for each number in the array and a blank for each character. The type of array has the same structure as the array. Type can be determined by the expression:

```
Type  ⇔  ↑0ρ⊂R
```

In this expression:

| | |
|---|---|
| `⊂` | makes R into a scalar that contains R. |
| `0ρ` | turns the scalar into an empty vector. |
| `↑` | selects the first item. |

The *prototype* of an array is defined as the type of its first item:

```
Prototype  ⇔  ↑0ρ⊂↑R
```

For example, for the three-item vector R:

```
      R←(2 3ρ1 'A' 2 3 'B' 'C') 'WORD' (9 10 11)

      R
 1 A 2   WORD  9 10 11
 3 B C
```

**Type**

```
      DISPLAY ↑0ρ⊂R
.→-----------------------.
| .→----. .→---. .→----. |
| ↓0   0| |    | |0 0 0| |
| |0    | '----' '~----' |
| '+----'                |
'∊-----------------------'
```

**Prototype**

```
      DISPLAY ↑0ρ⊂↑R
.→----.
↓0   0|
|0    |
'+----'
```

# Fill Item

**Note:** The information in this section is based on the APL2 language definition. Deviations exist on some platforms and are documented in the separate user's guides.

The prototype of an argument is used as a *fill item* when the operations take, expand, replicate, and disclose apply to certain arguments, as described below:

- Take (`L↑R`), page 244, and take with axis (`L↑[X]R`), page 247, use a fill item when the left argument specifies more items than the right argument contains. This application is called an *overtake*.
- Expand (`L\R` or `L⍀R`), page 122, and expand with axis (`L\[X]R` or `L⍀[X]R`), page 124, use a fill item to fill the expanded structure.
- Replicate (`L/R` or `L⌿R`), page 220, and replicate with axis (`L/[X]R` or `L⌿[X]R`), page 222, insert `|L[I]` fill items to correspond to a negative `I`th item in the left argument.
- Disclose (`⊃`), page 94, and disclose with axis (`⊃[X]R`), page 96, expand smaller items to the structure of the largest item in the array by padding with the fill item.

As an example, take the following assignment and display of a four-item nested vector `N`:

```
      N←((2 3)(4 5(6 7)))(8 9 10)11 12
      ρN
4
      ≡N
4
      DISPLAY N
.→---------------------------------------.
| .→--------------------. .→-----.       |
| | .→--. .→----------. | |8 9 10| 11 12 |
| | |2 3| |     .→--. | | '~-----'       |
| | '~--' |  4 5 |6 7| | |                |
| |       |     '~--' | |                |
| |       '∊----------' |                |
| '∊--------------------'                |
'∊---------------------------------------'
```

Note how the prototype fills the result on an overtake of the nested vector *N*:

```
      Z←5↑N
      DISPLAY Z
.→---------------------------------------------------------------.
| .→--------------------. .→-----.       .→--------------------. |
| | .→--. .→----------. | |8 9 10| 11 12 | .→--. .→----------. | |
| | |2 3| |     .→--. | | '~-----'       | |0 0| |     .→--. | | |
| | '~--' | 4 5 |6 7| | |                | '~--' | 0 0 |0 0| | | |
| |       |     '~--' | |                |       |     '~--' | | |
| |       '∊----------' |                |       '∊----------' | |
| '∊--------------------'                '∊--------------------' |
'∊---------------------------------------------------------------'
```

Prototypes are used to complete the definitions of functions and make them work in expected ways even at limiting cases. For example, the use of the prototype as the fill item causes the following to be true after application of take, expand, replicate, and disclose:

- Simple arguments give simple results.
- All numeric arguments give all numeric results.
- All character arguments give all character results.
- Uniformly nested arrays (each of whose items have the same structure) give uniformly nested results.

## Empty Arrays

An array is *empty* when the length of one or more of its axes is `0`. There is no empty scalar, but empty arrays may be of any other rank. Empty arrays have type and prototype. Figure 8 illustrates four empty arrays and explains how they are created.

***Uses of Empty Arrays:*** The following are common uses of an empty array:

- As right argument of the branch arrow (→) to resume execution of an expression in immediate execution (page 59) or within a defined function or defined operator to continue evaluation with the next line (page 349).
- As left argument of reshape (`⍴`) to create a scalar from an array (page 225).
- In variable specification to initialize a variable.
- As the value for a trace or stop control vector (*T*∆*name* and *S*∆*name*) to turn off the trace or stop (pages 361 and 362).

*Figure 8. Creating Simple Empty Arrays*

```
      MTN1←⍳0              Simple, empty numeric vector.
      MTN1
                           Empty vector displays as blank line.
      ⍴MTN1                Shape of array shows that it
0                          is a vector of length 0.
      DISPLAY MTN1         Picture display shows the vector.
.⊖.
|0|
'∼'
      MTN2←2 0⍴5           Simple empty 2-row numeric matrix.
      MTN2
                           Displays as two blank lines.
      ⍴MTN2                Shape of matrix is 2 0.
2 0
      DISPLAY MTN2         Picture display shows the matrix.
.⊖.
↓0|
|0|
'∼'
      MTC1←''              Empty character vector.
      MTC1
                           Vector displays as a blank line.
      ⍴MTC1                Shape is 0 for the empty vector.
0
      DISPLAY MTC1         Picture display shows the vector.
.⊖.
| |
'-'
      MTC2←0 4⍴''          Simple empty 4-column character
      MTC2                 matrix does not display as blank
      ⍴MTC2                line(s) because its row-axis has length 0.
0 4
      DISPLAY MTC2         Picture display shows the matrix.
.→---.
⌽    |
'----'
```

## Prototypes of Empty Arrays

As does any other array, an empty array has a depth and a prototype. The prototype of an empty array *T* is ↑*T*.

```
      ↑⍳0
0
      ↑2 0⍴0
0
```

```
      ≡2 0⍴⊂2 3 4
2
      ↑2 0⍴⊂2 3 4
0 0 0
```

See “Fill Functions” on page 56 for a discussion of the use of the prototype when an empty array is the argument of a primitive function.

## Empty Arrays and Nesting

A nested array may have empty arrays among its items. The following expression, for instance, creates a four-item nested vector of depth 2 that contains an empty array as its second item:

```
      VEC←'AGNES' (⍳0) 'HERB' 10
      VEC
 AGNES   HERB 10
      ⍴VEC
4
      ≡VEC
2
      DISPLAY VEC
.→----------------------.
| .→----. .⊖. .→---.     |
| |AGNES| |0| |HERB| 10 |
| '-----' '~' '----'     |
'∊----------------------'
```

**Note:** The *DISPLAY* function shows the prototype of empty arrays or items.

A nested array can contain only empty items, yet not be an empty array. Its prototype, however, is empty.

In contrast, the following is an empty nested array. It is nested because its prototype is not a simple scalar (either `0` or `' '`).

```
      T←0 2⍴⊂0 0
      ⍴T
0 2
      DISPLAY T
.→------------.
⌽ .→--. .→--. |
| |0 0| |0 0| |
| '∼--' '∼--' |
'∊------------'
      ≡T
2
      ↑T
0 0
```

Nested empty arrays are important because they allow expressions to work at the limit. For example:

- `5⍴⊂2⍴X` is a five-item vector of two-item vectors.
- `N⍴⊂2⍴X` is an *N*-item vector of two-item vectors. This is true even when *N* is `0`. That is, `0⍴⊂2⍴X` is an empty vector that has a two-item prototype.

## Scalar and Nonscalar Functions

According to the way they manipulate data, the primitive functions are either *scalar* or *nonscalar*.

Scalar functions include most computational functions. Figure 9 lists the scalar functions and Figure 10 on page 52 lists the nonscalar functions.

*Figure 9. Primitive Scalar Functions (All dyadic forms may take an axis.)*

| **Monadic Scalar** | **Function Symbol** | **Dyadic Scalar** |
|---|---|---|
| Conjugate | + | Add |
| Negative | - | Subtract |
| Direction | × | Multiply |
| Reciprocal | ÷ | Divide |
| Magnitude | \| | Residue |
| Floor | ⌊ | Minimum |
| Ceiling | ⌈ | Maximum |
| Exponential | * | Power |
| Natural Log | ⍟ | Logarithm |
| Pi Times | ○ | Circular |
| Factorial | ! | Binomial |
| Not | ~ | {Nonscalar Function} |
| Roll | ? | {Nonscalar Function} |
|  | ∧ | And |
|  | ∨ | Or |
|  | ⍲ | Nand |
|  | ⍱ | Nor |
|  | < | Less |
|  | ≤ | Not Greater |
|  | = | Equal |
|  | ≥ | Not Less |
|  | > | Greater |
|  | ≠ | Not Equal |

Formally, a function is a scalar function if indexing distributes over it. The primitive scalar functions have the additional property that "pick" (⊃) distributes over them. This property is called *pervasive*.

`F` is monadic scalar if:

`(F R)[I] ←→ F R[I]`

`F` is dyadic scalar if:

`(L F R)[I] ←→ L[I] F R[I]`      (scalar extension ignored) where indexing is taken as indexing an arbitrary rank array.

*F* is monadic pervasive if:

`(I⊃F R) ←→ F I⊃R`

*F* is dyadic pervasive if:

`(I⊃L F R) ←→ (I⊃L) F (I⊃R)`    (scalar extension ignored)

*Figure 10. Primitive Nonscalar Functions (Brackets indicate that an axis specification is optional.)*

| **Monadic Nonscalar** | **Function Symbol** | **Dyadic Nonscalar** |
|---|---|---|
| Shape | ⍴ | Reshape |
| Ravel [ ] | , | Catenate, Laminate [ ] |
| Reverse [ ] | ⌽ ⊖ | Rotate [ ] |
| Transpose | ⍉ | Transpose |
| Enclose [ ] | ⊂ | Partition [ ] |
| Disclose [ ] | ⊃ | Pick |
|  | ↓ | Drop [ ] |
| First | ↑ | Take [ ] |
| {Scalar Function} | ~ | Without |
| Interval | ⍳ | Index of |
| Enlist | ∊ | Member |
| Grade Up | ⍋ | Grade Up |
| Grade Down | ⍒ | Grade Down |
| {Scalar Function} | ? | Deal |
|  | ⍷ | Find |
|  | ⊤ | Encode |
|  | ⊥ | Decode |
| Matrix Inverse | ⌹ | Matrix Divide |
| Depth | ≡ | Match |
| Execute | ⍎ |  |
| Format | ⍕ | Format |
|  | [ ; ] | Indexing |
|  | ⌷ | Index [ ] |

## Conformability of Arguments

Permissible arguments for a particular dyadic function are determined by their structure and data and by their relationship to one another. Arguments are said to *conform* when they are compatible according to the requirements of the function.

Each scalar function applies to its argument(s) in a similar way and follows the conformability rules described below. These rules are not repeated in the descriptions of the scalar functions.

For nonscalar functions, the conformability rules and the way arguments relate follow no set pattern. The function descriptions explain these in detail.

## Monadic Scalar Function

Monadic scalar functions are defined on a simple scalar argument, then extended to other arguments, according to the following rules:

If the argument is a simple scalar, apply the function.

```
      ⌈3.6
4
```

If the argument is not empty, apply the function independently to each simple scalar in its argument.

```
      !3 6 9
6 720 362880
```

That is:

```
      (!3) (!6) (!9)
6 720 362880
```

```
      ÷2 3⍴⍳6
1    0.5 0.3333333333
0.25 0.2 0.1666666667
```

The result has a structure (rank, shape, and depth) identical to that of its argument.

If the argument is empty, apply the related fill function to ↑*R* (the prototype of the argument). Fill functions are discussed on page 56.

The following example illustrates the application of a monadic scalar function to a nested array.

```
      D←(2 8 6)(2 2⍴3 7 1)
      DISPLAY D
.→--------------.
| .→----. .→--. |
| |2 8 6| ↓3 7| |
| '~----' |1 3| |
|         '~--' |
'∊--------------'
      T←-D
      DISPLAY T
.→-------------------.
| .→-------. .→----. |
| |¯2 ¯8 ¯6| ↓¯3 ¯7| |
| '~-------' |¯1 ¯3| |
|            '~----' |
'∊-------------------'
```

# Dyadic Scalar Function

Dyadic scalar functions are defined on simple arguments, then extended to other arguments, according to the following rules.

## Scalar Conformability Rules

If both arguments are simple scalars, apply the function.

```
      2+3
5
```

If one or both arguments are empty arrays, apply the related fill function to ↑*L* and/or ↑*R* (the prototype of the empty array). Fill functions are discussed in the next section, page 56.

If arguments have the same shape, apply the function to corresponding items. The result has the same shape as the arguments.

```
      |¯¯¯¯¯¯| (1)
      | |¯¯¯¯|¯¯| (2)
      | | |¯¯|¯¯|¯¯|(3)
      5 6 7+10 20 30
      5 6 7+10 20 30
15 26 37
```

That is:

```
      (5+10) (6+20) (7+30)
        15      26      37
```

If one argument is a scalar or a one-item vector, pair the scalar or one-item vector with each item. The result has the same shape as the nonscalar argument.

```
      |¯¯¯¯¯|
      |¯¯¯| |
      |¯| | |
      1+2 3 4
      1+2 3 4
3 4 5
```

That is, the scalar extends to each item:

```
       (1+2) (1+3) (1+4)
         3     4     5
```

This extension is called *scalar extension*. (Scalar extension when the nonscalar argument is empty is discussed on page 57.)

When a dyadic scalar function is applied to nested arguments, the items are paired by the above rules. Then the rules are applied again to the resulting subexpressions. The shape of the result is the shape of the nonscalar argument. The structure of the result depends on the structure of the items.

In Figure 11 on page 55, both arguments are vectors of length 3. The left argument is composed of a scalar and two vector items. The right argument is composed of a nested vector, a scalar, and a vector item.

```
        |¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯|(1)
        |   |¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯|¯¯¯¯¯¯|(2)
        |   |      |¯¯¯¯¯¯¯¯¯|¯¯¯¯¯¯|¯¯¯¯¯¯|(3)
        |   |      |         |      |      |
      Z←2 (3 4) (5 6 7)+((8 9) 10) 11 (12 13 14)
      DISPLAY Z
.→----------------------------------.
| .→-----------. .→---. .→-------. |
| | .→----.     | |14 15| |17 19 21| |
| | |10 11| 12 | '∼----' '∼-------' |
| | '∼----'     |                    |
| '∊-----------'                    |
'∊----------------------------------'
```

That is:

```
1⊃Z  ⇔  2+(8 9) 10                  Pair the scalar with each item.
     ⇔  (2+8 9) (2+10)              Again, pair the scalar with
     ⇔  ((2+8) (2+9)) (2+10)        each item.
     ⇔  (10 11) 12

2⊃Z  ⇔  3 4+11                      Pair the scalar with each item.
     ⇔  (3+11) (4+11)
     ⇔  14 15

3⊃Z  ⇔  5 6 7+12 13 14              Pair corresponding items.
     ⇔  (5+12) (6+13) (7+14)
     ⇔  17 19 21
```

*Figure 11. Application of a Dyadic Scalar Function to Nested Arguments*

## Axis Specification with Scalar Dyadic Functions

An axis can be specified with each scalar dyadic function, as:

```
Z←L F[X] R
```

For example:

```
      L←1 10 100
      R←3 4ρ⍳12
      L×[1]R
  1    2    3    4
 50   60   70   80
900 1000 1100 1200
```

```
      S←1 10 100 1000
      S×[2]R
1  20  300  4000
5  60  700  8000
9 100 1100 12000
```

The axis indication X must be a simple scalar or vector selection of axes, not containing repetitions, such that:

```
(ρ,X) ←→ (ρρL)⌊ρρR
∧/X∊⍳(ρρL)⌈ρρR
```

The arguments are conformable if:

`(⍴L) ←→ (⍴R)[X]` or `(⍴R) ←→ (⍴L)[X]`
when `X≡X[⍋X]` (i.e., `X` is in ascending order)

The shape of the result is the shape of the array with greater rank.

```
      K←2 3⍴.1×⍳6
      K
0.1 0.2 0.3
0.4 0.5 0.6

      J←2 3 4⍴⍳24

      J+[1 2]K
 1.1  2.1  3.1  4.1
 5.2  6.2  7.2  8.2
 9.3 10.3 11.3 12.3

13.4 14.4 15.4 16.4
17.5 18.5 19.5 20.5
21.6 22.6 23.6 24.6
```

The order in which the axes appear does not affect the result. For the above example, for instance, `J+[1 2]K ←→ J+[2 1]K`.

---

# Fill Functions

When a primitive scalar function is presented with empty arguments or when a function derived from the operators each (`¨`) or array product (`.`) is presented with empty arguments, the function is not executed. Instead a related *fill function*, if defined, is executed with arguments `↑L` and/or `↑R` (the prototypes of the empty arguments).

## Fill Function for Primitive Scalar Functions

All primitive monadic and dyadic scalar functions have the same fill function as described below.

When the prototypes of the empty arguments are simple scalars, return a zero prototype. A ramification of this rule is that empty character arrays can be arguments to scalar functions whose range is numeric. The result has numeric type, as shown in the following examples:

```
      W←(⍳0)⌈⍳0
      DISPLAY W
.⊖.
|0|
'~'
```

```
      X←''÷''
      DISPLAY X
.⊖.
|0|
'~'
```

When prototypes of the empty arguments are not simple scalars, apply the fill function to each item recursively until simple scalars are reached.

```
      S←÷0 2⍴⊂1 2 3
      DISPLAY S
.→----------------.
⌽ .→----. .→----. |
| |0 0 0| |0 0 0| |
| '~----' '~----' |
'∊----------------'
```

When one argument is a scalar and the other is empty, apply the fill function between the item of the scalar and the prototype of the empty argument.

That is:

`Z←S⍴⊂(↑L)` *fill fn* `(↑R)`

where `S` is the shape of the empty argument.

For example:

```
      Z←2+0⍴⊂0 0
      ⍴Z
0
      DISPLAY Z
.⊖------.
| .→--. |
| |0 0| |
| '~--' |
'∊------'
```

## Fill Functions for Primitive Nonscalar Functions

Fill functions for primitive nonscalar functions are applied when the functions derived from the operators Each (¨) and Array product (.) are presented with empty arguments. The use of Each is discussed on pages 107 and 109. For more information on Array product, see pages 165 and 186. Figure 20 on page 110 shows the fill function related to each function for which a fill function exists.

---

## System Effects on Evaluation

The evaluation of expressions is affected by the limitations of the system.

## Size Limitations

Appendix C, “System Limitations for APL2” on page 489 lists size limitations of APL2, such as the smallest and largest representable numbers and the maximum rank and depth of an array.

## Precision

Calculations are carried out to 16 or 18 places depending on the hardware; however, use of certain primitives causes increased precision in calculation. The number of significant digits displayed depends on the setting of `⎕PP` (printing precision). For more information about `⎕PP`, see "`⎕PP` Printing Precision" on page 315.

Examples in this manual are shown with the default printing precision of `10`, unless noted otherwise.

## Comparison Tolerance

When comparing numbers that differ by only a very small amount, the limitations of the system can affect the results of the relational functions and the results of a few other functions that compare arguments to determine the result. (Figure 12 lists the affected functions.) To control these limitations, APL2 provides a *comparison tolerance* that is used to determine whether two numbers are considered equal.

Comparison tolerance, whose default value is `1E¯13`, can be set with the system variable `⎕CT`, page 275. It is used to compute a *relative fuzz* as follows:

```
      RFUZZ←⎕CT×(|A)⌈|B
```

Then, if `RFUZZ` is greater than or equal to `|A-B`, `A` and `B` are reported equal. For example:

```
      A←1.0000000000000001    |      A←1.00000001
      B←1.0000000000000009    |      B←1.00000009
      A=B                     |      A=B
1                             |0
```

*Figure 12. Functions Affected by Comparison Tolerance.*

Comparison tolerance is an implicit argument of the following functions:

All relational functions, page 219
Ceiling (`⌈R`), page 79
Equal (`L=R`), page 219
Find (`L⍷R`), page 129
Floor (`⌊R`), page 133
Greater than (`L>R`), page 219
Greater than or equal (`L≥R`), page 219
Index of (`L⍳R`), page 162
Less than (`L<R`), page219
Less than or equal (`L≤R`), page219
Match (`≡`), page 173
Member (`L∊R`), page 181
Not equal (`L≠R`), page 219
Residue (`L|R`), page 227

## System Tolerance

When a nonreal number is close to being a real number, a noninteger is close to being an integer, or a non-Boolean number is close to being Boolean, *system tolerance* or *system fuzz* defines how close the number must be before it is treated as an integer, a real number, or a Boolean number.

In contrast to comparison tolerance, which is used to determine a *relative fuzz*, system tolerance is an *absolute fuzz*.

***Real:*** A nonreal number is treated as real if the greater of the absolute values of the imaginary part and the tangent of the angle is less than approximately `1E¯13` for APL2/370 and `5E¯15` for the workstation systems.

***Integer:*** A number `R` is treated as an integer if it satisfies the condition above for being treated as real (or **is** real) and the difference between the real part of `R` and some integer is less than approximately `1E¯13×1⌈|R` for APL2/370 and `5E¯15×1⌈|R` for the workstation systems.

***Boolean:*** A non-Boolean number is treated as Boolean if the distance between it and `0` or `1` on the complex plane is less than approximately `1E¯13` for APL2/370 and `5E¯15` for the workstation systems.

System tolerance is fixed for the system and cannot be specified.

## Errors and Interrupts in Immediate Execution

If either an expression in immediate execution generates an error or you have signaled an interrupt, execution of the expression is suspended and a message is displayed:

```
      T+4
VALUE ERROR+
      T+4
      ^^
```

The first line of the message indicates the cause of the suspension. The second line repeats the expression as entered. And the third line contains two carets. The left caret indicates how far execution of the expression progressed before the suspension occurred. The right caret indicates the likely point of the error. (On occasion, the two carets overlap so that only one is displayed.)

("Suspension of Execution" on page 354 further discusses suspension of execution.)

The state indicator (page 355) shows that the expression is suspended. The asterisk indicates a suspended immediate execution expression. (If a defined function or operator is suspended, its name and line number are shown in the state indicator.)

```
      )SIS
*   T+4
    ^^
```

Expressions should be cleared from the state indicator. Clearing the state indicator is fully discussed in “Clearing the State Indicator” on page 357. The example below shows that the state indicator is cleared by correcting the error that caused the interruption and resuming execution. `T` is assigned a value and then execution of the expression is resumed by `→⍳0`:

```
      T←3
      →⍳0
7
      )SIS
```

If the state indicator shows several errors and you do not want to resume execution, you can use:

- Escape (`→`) for each suspension to be removed from the state indicator
- `)RESET` *n* to remove *n* lines from the state indicator
- `)RESET` to clear the state indicator entirely.

In the following example, for instance, the state indicator contains three suspended immediate execution expressions. The `)RESET` command is used to clear the state indicator without resumption of execution of any of them.

```
      )SIS
*  1 2 3+2 3⍴⍳6
   ^    ^
*  ⍳2.2
   ^
*  4÷0
   ^^
      )RESET
      )SIS
```

Keeping the state indicator clear is good practice. This makes it easier to use the state indicator in diagnosing problems in defined functions and operators; it can even prevent a `WS FULL` condition caused by large suspensions awaiting resolution.

## Shared Variables

Shared variables are the means by which two processors can communicate with each other. A processor can be an *auxiliary processor*, which provides system services, or another APL2 session.

Any user-named variable can be a shared variable. System variables (which are actually shared with the APL2 system) cannot be shared with other processors. When the term *variable* is used in this chapter, it means only user-named variables.

The APL2 Program Products include auxiliary processors, which communicate with an APL2 user through shared variables. Auxiliary processors are programs that perform services for APL2 users, such as writing to a data file. See Chapter 8, “Shared Variables” on page 364 for a full discussion of shared variable concepts. The workstation user's guides contain descriptions of the auxiliary processors distributed with the specific workstation platform. See *APL2/370 Programming*:

*System Services Reference* for detailed descriptions of each of the auxiliary processors distributed with APL2/370.

***Degree of Coupling:*** Variables used to pass data between processors are *shared* by the two processor *partners*. *Degree of coupling* describes the share status and is the explicit result of ⎕*SVO* and ⎕*SVR*. Figure 13 describes the meaning of coupling degrees for each system function.

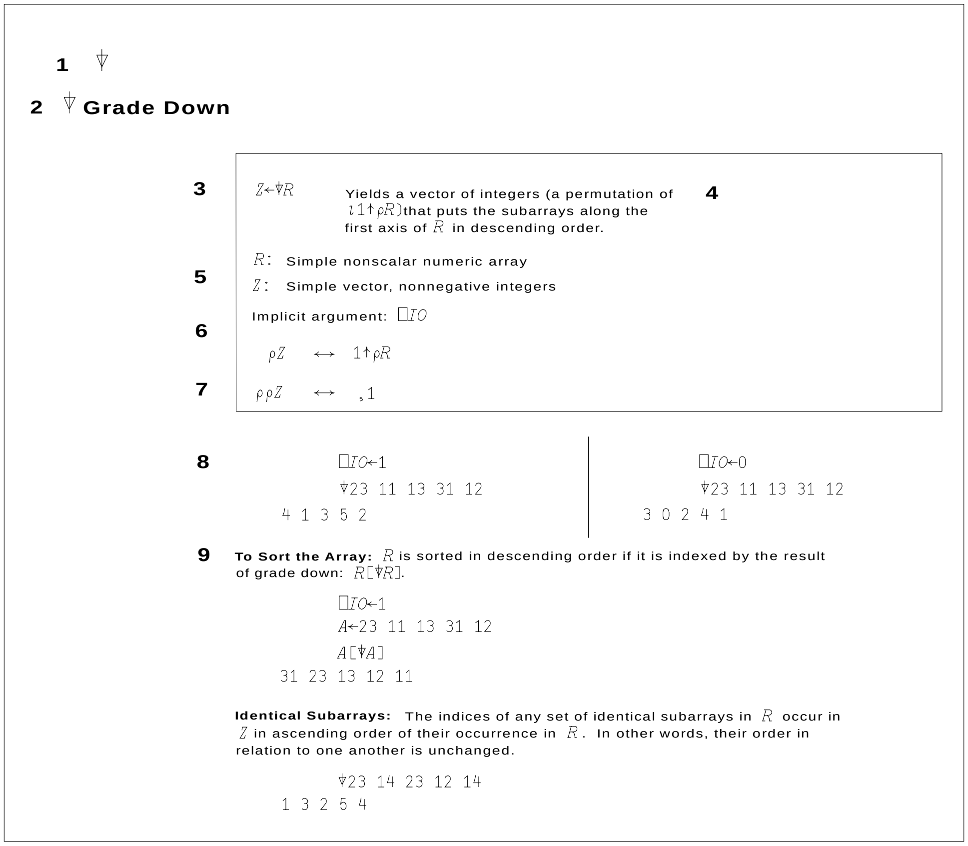
*Figure 13. Degree of Coupling Returned from System Functions*

| | **Offer** | **Inquire** | **Retract** |
|---|---|---|---|
| | *L* ⎕*SVO R* | ⎕*SVO R* | ⎕*SVR R* |
| **0** | Offer failed—APL2 refused your offer. | The variable is not a shared variable. Either no offer was made or the offer failed. | The variable was not a shared variable. Either no offer was made or the offer failed. |
| **1** | Offer is pending—your offer has not yet been matched by your partner. | Offer is pending. Your partner has not matched your offer.<br>Or, your partner has retracted the variable or APL2 has retracted the variable as a result of an error condition. | The variable was waiting to be matched or it was already retracted by your partner. |
| **2** | The offered variable is fully coupled. | The variable is fully coupled. | The variable was fully coupled. |

# Chapter 5. Primitive Functions and Operators

This chapter describes all primitive functions and operators alphabetically. The operators are described in the context of their derived functions. Each description of a function or operator consists of a summary and several detailed sections.

Figure 14 shows a sample page. The callouts in the figure are explained immediately following the figure.



**1** ⍒

**2** ⍒ **Grade Down**

**3** Z←⍒R Yields a vector of integers (a permutation of ⍳1↑⍴R) that puts the subarrays along the first axis of R in descending order. **4**

**5** R: Simple nonscalar numeric array

Z: Simple vector, nonnegative integers

**6** Implicit argument: ⎕IO

```
 ⍴Z   ←→  1↑⍴R
⍴⍴Z   ←→  ,1
```

**7**

**8**

```
      ⎕IO←1
      ⍒23 11 13 31 12
4 1 3 5 2
```

```
      ⎕IO←0
      ⍒23 11 13 31 12
3 0 2 4 1
```

**9** **To Sort the Array:** R is sorted in descending order if it is indexed by the result of grade down: R[⍒R].

```
      ⎕IO←1
      A←23 11 13 31 12
      A[⍒A]
31 23 13 12 11
```

**Identical Subarrays:** The indices of any set of identical subarrays in R occur in Z in ascending order of their occurrence in R. In other words, their order in relation to one another is unchanged.

```
      ⍒23 14 23 12 14
1 3 2 5 4
```

*Figure 14. Sample Page of Primitive Functions and Operators*

1. The operation symbol
2. Operation symbol and name as they appear in the table of contents
3. Each primitive has a subset of the following operation syntax:

| | |
|---|---|
| L | Left argument |
| R | Right argument |
| LO | Left operand |
| RO | Right operand |
| Z | Result |
| X | Axis |

4. Summary definition of the operation
5. Properties of the argument(s) or operand(s), the result, and axis. **Properties are listed on an exception-basis. The most general property is always assumed, and only limitations are listed**. For example, “R: Numeric” means arrays of any rank, depth, or count (empty or nonempty) that contain real and/or nonreal numbers.
6. Implicit argument. Those system variables, such as ⎕IO (index origin) and ⎕CT (comparison tolerance), that affect the result of the function.
7. Shape and rank of the result. Whenever possible, an expression for determining these characteristics of the result is given. Otherwise, the characteristic is listed as “data-dependent.”
8. Detailed description of the function, including such topics as:
    - Conformability, if the function is not a scalar function
    - Behavior with various arguments, including nested arrays and edge cases (scalar arguments for functions whose primary definition is based on nonscalar arrays and empty arrays)
    - Identities, showing the relationship of the operation to other operations.
9. Examples. When the specification of the arguments is not shown, the values of the arguments are shown along with their shape and depth, or the argument is illustrated with DISPLAY.

    Most examples are shown with the default printing precision (⎕PP) of 10 and in origin 1 (⎕IO). If an example changes either the printing precision or the origin, the specification of the appropriate system variable is shown, and the next example returns ⎕IO or ⎕PP to its default.

## APL2 Expressions Used in the Descriptions

APL2 expressions are used in the descriptions to add precision and conciseness to the text. The following expressions are commonly used:

| **Expression** | **Meaning** |
|---|---|
| ⍴A | Shape of A |
| ⍴⍴A | Rank of A |
| ≡A | Depth of A |
| ¯1↑⍴A | The last axis (columns) of A |
| ¯1↓⍴A | All but the last axis of A |
| 1↑⍴A | The first axis of A |
| 1↓⍴A | All but the first axis of A |
| ¯1↑1,⍴A | 1 if A is a scalar; the last axis of A otherwise |
| ⍳⍴A | The integers 1 through ⍴A |
| 0⌈¯1+⍴⍴A | A rank of one less than the rank of A. If A is a scalar or a vector, the rank is 0. |

## Meta Notation Used in Descriptions

←→ or ⇔ The expressions on each side evaluate to the same array.

## Multivalued Functions

When a function mathematically has more than one value, APL2 chooses a principal value. For example, the cube root of a negative number in APL2 is the one with the smallest nonnegative angle in the complex plane.

# + Add

`Z←L+R`   Adds `R` to `L`.

`L`, `R`, and `Z`: Numeric

*Scalar Function*

Add is the arithmetic addition function.

```
      .4+6
6.4
      ¯5+¯.3 6 3J4
¯5.3 1 ¯2J4
```

```
      1J2+3J4
4J6
      0 .3 ¯8+0 ¯.3 8
0 0 0
```

# ! Binomial

`Z←L!R` For nonnegative integer arguments, yields the number of distinct combinations of `R` things taken `L` at a time.

In the following table, `<0` means that `L`, `R`, or `R-L` is a negative integer and `≥0` means that `L`, `R`, or `R-L` is a nonnegative integer. The corresponding definition is used.

| **Case** | | | **Definition** |
|---|---|---|---|
| `L` | `R` | `R-L` | |
| `≥0` | `≥0` | `≥0` | Return `(!R)÷(!L)×!R-L` |
| `≥0` | `≥0` | `<0` | Return `0` |
| `≥0` | `<0` | `≥0` | (Case cannot occur.) |
| `≥0` | `<0` | `<0` | Return `(¯1*L)×L!L-R+1` |
| `<0` | `≥0` | `≥0` | Return `0` |
| `<0` | `≥0` | `<0` | (Case cannot occur.) |
| `<0` | `<0` | `≥0` | Return `(¯1*R-L)×(-R+1)!(|L+1)` |
| `<0` | `<0` | `<0` | Return `0` |

*Scalar Function*

```
      2!5
10
      2 3 4!6 18 24
15 816 10626
```

```
      2!3J2
1J5
      3!.05 2.5 ¯3.6
0.0154375 0.3125 ¯15.456
```

Although the domain of factorial excludes negative integers, the domain of the binomial does not. Any implied division by zero in the numerator `!R` is usually accompanied by corresponding division by zero in the denominator. The binomial function, therefore, extends to all numbers, except in the case where `R` is a negative integer and `L` is not an integer.

```
      A←¯6+⍳11
      A ∘.!A
   1  ¯4   6 ¯4  1 0 0 0 0 0  0
   0   1  ¯3  3 ¯1 0 0 0 0 0  0
   0   0   1 ¯2  1 0 0 0 0 0  0
   0   0   0  1 ¯1 0 0 0 0 0  0
   0   0   0  0  1 0 0 0 0 0  0
   1   1   1  1  1 1 1 1 1 1  1
  ¯5  ¯4  ¯3 ¯2 ¯1 0 1 2 3 4  5
  15  10   6  3  1 0 0 1 3 6 10
 ¯35 ¯20 ¯10 ¯4 ¯1 0 0 0 1 4 10
  70  35  15  5  1 0 0 0 0 1  5
¯126 ¯56 ¯21 ¯6 ¯1 0 0 0 0 0  1
```

***Binomial Expansion:*** The coefficients of the binomial expansion `(X+1)*R` can be determined by this expression:

```
(0,⍳R)!R
```

For example, the coefficients of `(X+1)*3` are:

```
      0 1 2 3!3
1 3 3 1
```

***Relationship to Beta Function:*** Binomial is related to the Beta Function as follows:

```
(β)(L,R) ←→ ÷R×(L-1)!L+R-1
```

## ∧∨⍲⍱~ Boolean Functions

| | |
|---|---|
| `Z←~R` | Not |
| `Z←L∧R` | And |
| `Z←L∨R` | Or |
| `Z←L⍲R` | Nand |
| `Z←L⍱R` | Nor |

`L`, `R`, and `Z`: Boolean

*Scalar Functions*

The monadic Boolean function Not changes its argument either from `0` to `1` or from `1` to `0`.

```
      ~0
1
```

```
      ~1
0
```

The following tables define the dyadic Boolean functions.

**And**

```
∧ | 0 1
- - - -
0 | 0 0
1 | 0 1
```

**Or**

```
∨ | 0 1
- - - -
0 | 0 1
1 | 1 1
```

**Nand**

```
⍲ | 0 1
- - - -
0 | 1 1
1 | 1 0
```

**Nor**

```
⍱ | 0 1
- - - -
0 | 1 0
1 | 0 0
```

***Relational Functions as Boolean Functions:*** The relational functions (`<  ≤  =  ≥  >  ≠` ) (see "<≤=≥>≠ Relational Functions" on page 219), when applied to Boolean arguments, produce only Boolean results. For example, `L≠R` is the *exclusive-or* of `L` and `R`, and `L≤R` is *material implication*.

Figure 15 shows all possible Boolean results for `L` *fn* `R` and the functions that produce them, where `L` and `R` are specified to produce all possible combinations of `0` and `1`.

*Figure 15. Boolean Functions*

```
L←0 0 1 1
R←0 1 0 1
```

| Name | Syntax | Result |
|---|---|---|
|  | 0 ∧ *R* | 0 0 0 0 |
| And | *L* ∧ *R* | 0 0 0 1 |
| Greater than | *L* > *R* | 0 0 1 0 |
|  | *L* | 0 0 1 1 |
| Less than | *L* < *R* | 0 1 0 0 |
|  | *R* | 0 1 0 1 |
| Not equal | *L* ≠ *R* | 0 1 1 0 |
| Or | *L* ∨ *R* | 0 1 1 1 |
| Nor | *L* ⍱ *R* | 1 0 0 0 |
| Equal | *L* = *R* | 1 0 0 1 |
| Not | ∼ *R* | 1 0 1 0 |
| Greater than or equal | *L* ≥ *R* | 1 0 1 1 |
| Not | ∼ *L* | 1 1 0 0 |
| Less than or equal | *L* ≤ *R* | 1 1 0 1 |
| Nand | *L* ⍲ *R* | 1 1 1 0 |
|  | 1 ∨ *R* | 1 1 1 1 |

# [ ] Bracket Index

`Z←A[I]` Selects subarrays from `A` according to the index arrays `I`. Within `I`, semicolons separate arrays that define positions along each axis.

`I`: Simple nonnegative integer array
`A`: Nonscalar array

Implicit argument: `⎕IO`

`⍴Z` ←→ Catenated shapes of the index arrays
`⍴⍴Z` ←→ Sum of the ranks of the index arrays

In form, bracket index is similar to subscript notation. An index array defines the positions to be selected along each axis.

For example, if `A` is a matrix, the item $A_{ij}$ is that item which is in the *i*th row and *j*th column of `A`. In APL2, the bracket index of the item in `I`th row and `J`th column is denoted by `A[I;J]`.

```
      ⎕IO←1
      'CURTAIL'[1 2 4]
CUT
```

```
      ⎕IO←0
      'CURTAIL'[0 1 3]
CUT
```

***When a Vector Is Indexed:*** If `A` is a vector, `I` is a single index array and `I∊⍳⍴A`.

```
      ⎕IO←1
      A←23 9 6.3 8 ¯3 7
      Z←A[3]
      Z
6.3
      ⍴Z
(empty)
      Z←A[2 5 1]
      Z
9 ¯3 23
      ⍴Z
3
```

```
      B←2 3⍴1 4 3 2 6 5
      B
1 4 3
2 6 5
      ⍴B
2 3

      Q←A[B]
      Q
23 8  6.3
 9 7 ¯3

      ⍴Q
2 3
```

***When a Matrix Is Indexed:*** If `A` is a matrix, then two arrays of indexes can be given, separated by a semicolon: `[I;J]` and `I∊⍳1↑⍴A` and `J∊⍳¯1↑⍴A`. The index arrays `I` and `J` reference the rows and columns of `A`, respectively.

The array of items selected represents the `J`th items of the `I`th rows. For example, `A[1 2;1 3]` selects `A[1;1]`, `A[1;3]`, `A[2;1]`, `A[2;3]` , not just `A[1;1]`, `A[2;3]`.

```
      C←'ABCDEFGHIJKLMNOPQR'
      C←3 6⍴C
      C
ABCDEF
GHIJKL
MNOPQR
      J←C[2;3]
      J
I
      ⍴J
(empty)
      P←C[1;3 1 4]
      P
CAD
      ⍴P
3
```

```
      M←C[1 2;1 3]
      M
AC
GI
      ⍴M
2 2

      N←C[1 3;2 3⍴6 1 3 4 5 2]
      N
FAC
DEB

RMO
PQN
      ⍴N
2 2 3
```

***Eliding Index Arrays:*** Index arrays may be elided to indicate all indexes for the corresponding axes. If all indexes are elided, the result is `A`.

```
      D←3 4⍴C[1 2;]
      D
ABCD
EFGH
IJKL
```

```
      D[;1]
AEI
      D[3;]
IJKL
```

***Repetitions of Index Values:*** Index values can be repeated. The indicated item is selected repeatedly.

```
      H←2 4⍴3 4 1 2 2 3 4 1
      H
3 4 1 2
2 3 4 1
      'EMIT'[H]
ITEM
MITE
```

```
      'NAB'[3 2 1 2 1 2]
BANANA
```

***When a Higher-Rank Array Is Indexed:*** The pattern of representing index arrays established for matrixes is the same for arrays of higher rank. There must be `¯1+ρρA` semicolons, and an index array for any axis may be elided.

```
      U←2 3 4ρ(,C),'STUVWX'
      U
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
      U[1;2;4]
H
      U[2;1;1 3 4]
MOP
```

```
      U[;2;4]
HT
      U[1;1 3;2 4]
BD
JL
      U[1;;3]
CGK
      U[2;1;]
MNOP
      U[;3;]
IJKL
UVWX
```

***When a Nested Array Is Indexed:*** Bracket index does not affect the depth of any selected item. With bracket index, only an item in the outermost structure can be selected.

```
      V←'H' 'HI' ('HIM' 'HIS')
      Z←V[1]
      Z
H
      ≡Z
0
      ρρZ
0
      E←V[2]
      E
 HI
      ≡E
2
      S←V[3]
      S
  HIM HIS
      ≡S
3
      ρρS
0
```

***Selective Specification:*** Bracket index can be used for selective specification:

For the `V` shown above:

```
      ρV
3
      ≡V
3
      V[3]←'H'
      V
 H HI H
      ≡V
2
```

```
      W←2 3ρ'ABCDEF'
      W[1;1 3]←8 9
      W
8 B 9
D E F
      B←3 4 5
      B[]←9
      B
9 9 9
```

**Note:** Bracket index does not follow the syntax of a dyadic function and is not in the function domain of operators.

## , Catenate

| | |
|---|---|
| `Z←L,R` | Joins `L` and `R`. If `L` and `R` are nonscalar arrays, `L` and `R` are joined along the last axis. If `L` and `R` are scalars, `Z` is a two-item vector. |
| `¯1↑⍴Z` | ↔ Case dependent; see below. |
| `⍴⍴Z` | ↔ `,⌈/(⍴⍴L),(⍴⍴R),1` |

```
      Z←2 4 6,1 3 5
      Z
2 4 6 1 3 5
      ⍴Z
6

      Z←'ABC',1 2 3 4
      Z
ABC 1 2 3 4
      ⍴Z
7
```

```
      K←2 3⍴⍳6
      K
1 2 3
4 5 6

      Q←2 2⍴7 8 9 10
      Q
7  8
9 10

      H←K,Q
      H
1 2 3 7  8
4 5 6 9 10
```

***Catenate and Vector Notation:*** The result of catenate applied to simple scalars or vectors is the same as a simple vector created by vector notation:

```
      M←2,3
      M
2 3
      N←2 3
      M≡N
1
```

```
      X←9 8 7,6 5 4
      X
9 8 7 6 5 4
      Q←9 8 7 6 5 4
      X≡Q
1
```

**Note:** For vector notation `A B C` ↔ `(⊂A),(⊂B),⊂C`; vector notation and catenate cannot be used interchangeably. Compare:

```
      E←'TO', 'KEN'
      E
TOKEN
      ⍴E
5
      ≡E
1
```

```
      F←'TO' 'KEN'
      F
 TO KEN
      ⍴F
2
      ≡F
2
```

***Conformability:*** The arguments are conformable for catenate in one of three ways:

- They have the same rank.
- At least one argument is a scalar.
- They differ in rank by 1.

The last two cases involve reshaping the argument of smaller rank so the arguments have the same rank. After this extension, the shape of the result is described as follows:

```
(ρZ) ←→ (¯1↓ρL),(¯1↑ρL)+(¯1↑ρR)
```

***Arguments Have the Same Rank:*** Vectors can be of any length. For matrixes and higher order arrays, the lengths of all axes but the last must be the same: `(¯1↓ρL) ←→ ¯1↓ρR`.

```
      A←3 4ρ'BLUESHOEFOOT'
      A          ⍝ ρA is 3 4
BLUE
SHOE
FOOT

      B←3 5ρ'BERRYLACESSTOOL'
      B          ⍝ ρB is 3 5
BERRY
LACES
STOOL

      Z←A,B
      Z          ⍝ ρZ is 3 9
BLUEBERRY
SHOELACES
FOOTSTOOL

      C←2 1ρ'THOMAS' 'WILLIAM'
      ≡C
2
      D←2 1ρ('AQUINAS' 'MORE')('OCKHAM' 'SHAKESPEARE')
      ≡D
3
      C,D
 THOMAS    AQUINAS MORE
 WILLIAM   OCKHAM SHAKESPEARE
      ≡C,D
3
```

If the two arguments are different types of empty arrays, the type of the result is the type of *R*.

```
      J←'',ι0
      ↑J
0
      K←(ι0),''
      ↑K
```

(*Prototype is a character blank*)

***One Argument Is a Scalar:*** The scalar argument is reshaped with a last axis of length 1 to match the nonscalar argument. If *L*, for instance, is the scalar argument, it is reshaped as follows: `L←((¯1↓ρR),1)ρL`.

```
      'S',2 4ρ'PRIGTRAY'
SPRIG
STRAY
```

```
      (2 2 3ρι12),'*'
 1  2  3 *
 4  5  6 *

 7  8  9 *
10 11 12 *
```

***The Arguments Differ in Rank by 1:*** The lengths of all axes but the last of the array with greater rank must be the same as the array with smaller rank. If *L* is the argument with greater rank, `(¯1↓ρL) ←→ ρR`.

The argument of smaller rank is augmented to conform with the argument of greater rank by including a last axis of length 1. If, for instance, *L* is the argument of smaller rank, it is reshaped as follows: `L←((ρL),1)ρL`.

```
      U←'SAT'
      U
SAT
      V←'TEAMMAZERAIL'
      V
TEAM
MAZE
RAIL
      U,V
STEAM
AMAZE
TRAIL
```

```
      W←'1: ' '2: '
      Y←,[ι0]'LOG ON' 'LOG OFF'
      G←W,Y
      G
 1: LOG ON
 2: LOG OFF

      ρG
2 2
      ≡G
2
```

## ,[ ] Catenate with Axis

`Z←L,[X]R` Joins `L` and `R` along the axis indicated by `X`.

`Z`: Nonscalar
`X`: Simple scalar or one item vector, integer: `X∊⍳(⍴⍴L)⌈⍴⍴R`

Implicit argument: `⎕IO`

```
 ⍴Z   ←→ Case dependent; see below.
⍴⍴Z   ←→ (⍴⍴L)⌈⍴⍴R
```

Catenate with axis is similar to catenate except that the arrays are joined along the indicated axis instead of along the last axis.

Catenate with axis is not defined if both arguments are scalars. If both arguments are vectors or if one is a vector and one is a scalar, catenate with axis is equivalent to catenate.

***Conformability:*** The conformability requirements for catenate with axis are similar to those for catenate. After scalar extension, the shape of the result is described by the following formula:

```
(⍴Z)[X] ←→ (⍴L)[X]+(⍴R)[X]
```

***One Argument Is a Scalar:*** The scalar argument is reshaped to have the same shape as the nonscalar argument except that the `X`th axis has length 1.

```
      A←3 4⍴'BATHBEATBIND'
      A
BATH
BEAT
BIND

      A,[1]'X'
BATH
BEAT
BIND
XXXX
```

```
      0,[1]2 5⍴⍳10
0 0 0 0  0
1 2 3 4  5
6 7 8 9 10
```

***Arguments Have the Same Rank:*** Except for the $X$th axis, the lengths of all axes must be the same. Then `(⍴Z)[X] ←→ (⍴L)[X]+(⍴R)[X]`.

```
      A
BATH
BEAT
BIND
      B←2 4⍴'ZOOMZERO'
      B
ZOOM
ZERO
      A,[1]B
BATH
BEAT
BIND
ZOOM
ZERO
```

```
      C←2 2 3⍴⍳12
      D←2 3 3⍴-⍳18
      C,[2]D
  1   2   3
  4   5   6
 ¯1  ¯2  ¯3
 ¯4  ¯5  ¯6
 ¯7  ¯8  ¯9

  7   8   9
 10  11  12
¯10 ¯11 ¯12
¯13 ¯14 ¯15
¯16 ¯17 ¯18
```

***The Arguments Differ in Rank by 1:*** Except for the $X$th axis of the array of greater rank, the lengths of all axes must be the same as the lengths of the axes of the array of lesser rank.

The argument with the lower rank is augmented to conform with the higher rank argument by including an $X$th axis of length 1.

```
      H←'words'
      H
words
      K←2 5⍴'STRAWBERRY'
      K
STRAW
BERRY
      H,[1]K
words
STRAW
BERRY
```

```
      Q←3 5⍴⍳15
      S←3 3 5⍴-⍳45
      Z←Q,[1]S
      ⍴Z
4 3 5

      Z←Q,[2]S
      ⍴Z
3 4 5
```

## ⌈ Ceiling

`Z←⌈R` For real numbers, yields the smallest integer that is not less than `R` (within the comparison tolerance).

For complex numbers, depends on the relationship of the real and imaginary parts of `R`.

`R` and `Z`: Numeric

Implicit argument: `⎕CT`

*Scalar Function*

Ceiling is defined in terms of floor:

`⌈R ←→ -⌊-R`

(For the determination of the result based on the relationship of the real and imaginary parts of `R`, see page 133).

```
      ⌈2.3
3
      ⌈¯2.7 3 .5
¯2 3 1
```

```
      ⌈1.5J2.5
1J3
      ⌈1J2 1.2J2.5 ¯1.2J¯2.5
1J2 1J3 ¯1J¯2
```

Figure 16 illustrates the ceiling of a complex number. Any number within the rectangle has point `B` as its ceiling.

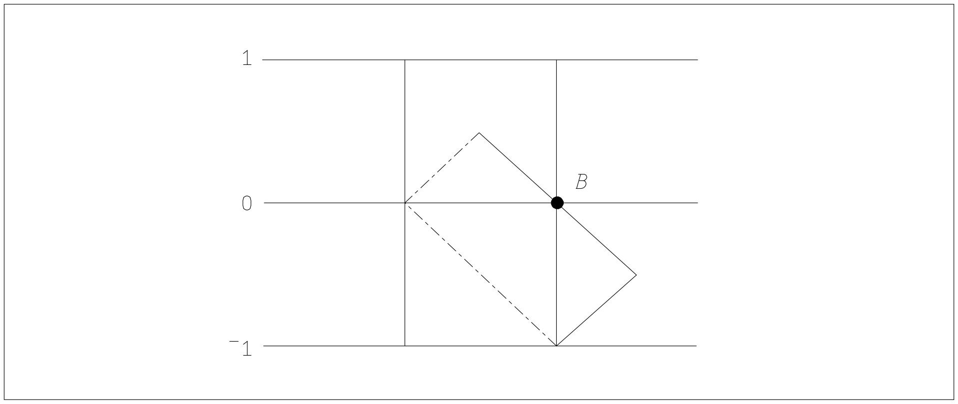


*Figure 16. The Shape of the Complex Ceiling Area*

The rectangle of sides square root of 2 by square root of .5 is oriented so that the center of one long side is coincident with a lattice point `B`, and with the ends of the opposite long side coincident with the lattice points below and to the left of `B`. The points within the rectangle all have `B` as ceiling. The two edges of the rectangle associated with `B` as ceiling are the top one, on which `B` lies, and the one to the right, as shown by the darker lines in the figure.

# ○ Circle Functions

*Z*←*L*○*R* *L* determines which of a family of circular, hyperbolic, Pythagorean, and complex number functions to apply to *R*.

*L*: Integer such that ¯12≤*L* and *L*≤12
*R* and *Z*: Numeric

*Scalar Function*

Figure 17 lists left arguments and names the functions they generate. Figure 18 on page 81 provides formulas for the functions ¯8≤*L* and *L*≤8 for complex *R*.

*Figure 17. Circular, Hyperbolic, Pythagorean, and Complex Number Functions*

| *L* | *L* ○ *R* | *L* | *L* ○ *R* |
|---|---|---|---|
| | | 0 | (1-*R**2)*.5 |
| ¯1 | Arcsin *R* | 1 | Sine *R* |
| ¯2 | Arccos *R* | 2 | Cosine *R* |
| ¯3 | Arctan *R* | 3 | Tangent *R* |
| ¯4 | (¯1+*R**2)*.5 | 4 | (1+*R**2)*.5 |
| ¯5 | Arcsinh *R* | 5 | Sinh *R* |
| ¯6 | Arccosh *R* | 6 | Cosh *R* |
| ¯7 | Arctanh *R* | 7 | Tanh *R* |
| ¯8 | -(8○*R*) | 8 | -(¯1-*R**2)*.5 for *R*≥0<br>(¯1-*R**2)*.5 for *R*<0 |
| ¯9 | *R* | 9 | Real *R* |
| ¯10 | +*R* | 10 | \|*R* |
| ¯11 | 0J1×*R* | 11 | Imaginary *R* |
| ¯12 | *0J1×*R* | 12 | Phase *R* |

In the descriptions of the circle functions on nonreal values, the following functions on real numbers are assumed:

```
   SIN X ←→ 1○X
   COS X ←→ 2○X
  SINH X ←→ 5○X
  COSH X ←→ 6○X
  TANH X ←→ 7○X
```

The following variables are also assumed:

```
  I←0J1
  R←X+0J1×Y
  PI←○1
```

In the following formulas, redundant parentheses are used for clarity.

```
 0○R ←→ (1-R*2)*.5

 1○R ←→ SINZ R
     ←→ ((SIN X)×(COSH Y))+I×(COS X)×(SINH Y)

¯1○R ←→ ASINZ R
     ←→ -I×ASINHZ (I×R)

 2○R ←→ COSZ R
     ←→ ((COS X)×(COSH Y))-I×(SIN X)×(SINH Y)

¯2○R ←→ ACOSZ R
     ←→ (.5×PI)-ASINZ R

 3○R ←→ TANZ R
     ←→ ((SIN X)+I×(COS X)×(TANH Y))÷(COS X)-I×(SIN X)×(TANH Y)

¯3○R ←→ ATANZ R
     ←→ -I×ATANHZ (I×R)

 4○R ←→ (1+R*2)*.5

¯4○R ←→ (¯1+R*2)*.5          for X≥0
                             or for ¯1<X and X<0 and Y=0
     ←→ -(¯1+R*2)*.5         otherwise

 5○R ←→ SINHZ R
     ←→ -I×SINZ (I×R)

¯5○R ←→ ASINHZ R
     ←→ -I×ASINZ (I×R)

 6○R ←→ COSHZ R
     ←→ COSZ I×R

¯6○R ←→ ACOSHZ R
     ←→ ⍟(R+¯4○R)

 7○R ←→ TANHZ R
     ←→ -I×TANZ (I×R)

¯7○R ←→ ATANHZ R
     ←→ -I×ATANZ (I×R)

 8○R ←→ (¯1-R*2)*.5          for X>0 and Y>0
                              or X=0 and Y>1
                              or X<0 and Y≥0
     ←→ -(¯1-R*2)*.5         otherwise

¯8○R ←→ -8○R
```

*Figure 18. Formulas for Circular, Hyperbolic, and Pythagorean Functions Applied to Complex Arguments*

## Circular Functions

The circular functions *sine*, *cosine*, and *tangent* (`1○R`, `2○R`, and `3○R`) require a right argument expressed in radians.

```
      1○1.570796327                     2○1
1                                 0.5403023059
      3○2                               ÷3○2
¯2.185039863                      ¯0.4576575544
```

The last example in the right column is the cotangent of 2 radians.

Degrees can be converted to radians with the expression:

```
RADIANS←○DEGREES÷180
```

```
      1○○30÷180                         2○○45÷180
0.5                               0.7071067812
```

***Inverses of Circular Functions:*** The inverses of the circular functions *arcsine*, *arccosine*, and *arctangent* (`¯1○R`, `¯2○R`, and `¯3○R`) yield their result in radians.

```
      ¯1○1                              ¯2○.54032023059
1.570796327                       0.9999786982
```

Radians can be converted to degrees with the expression:

```
DEGREES←180×RADIANS÷○1
```

```
                                        (¯3○1)×180÷○1
      (¯1○.5)×180÷○1               45
30
```

Because sine, cosine, and tangent are cyclic, their inverses are many-valued. The principal values for *real* `R` are chosen in the following intervals:

| | | |
|---|---|---|
| Arcsin | `Z←¯1○R` | `(|Z)≤○0.5` |
| Arccos | `Z←¯2○R` | `(Z≥0)∧(Z≤○1)` |
| Arctan | `Z←¯3○R` | `(|Z)≤○0.5` |

## Hyperbolic Functions

The hyperbolic functions *sinh* and *cosh* (`5○R` and `6○R`) are the odd and even components of the exponential function; that is, `5○R` is odd, `6○R` is even, and the sum `(5○R)+6○R` approximates `*R`. Consequently:

`5○ R  ←→  .5×(*R)-(*-R)`

$$\frac{e^{R}-e^{-R}}{2}$$

`6○ R  ←→  .5×(*R)+(*-R)`

$$\frac{e^{R}+e^{-R}}{2}$$

```
      5○1                   |        6○1
1.175201194                 | 1.543080635
```

The definition of the hyperbolic tangent function *tanh* (`7○R`) is analogous to that of the tangent, that is:

`7○R ←→ (5○R)÷6○R`

***Inverse Hyperbolic Functions:*** Arcsinh, arccosh, and arctanh are provided by left arguments `¯5`, `¯6`, and `¯7`, respectively.

```
      ¯5○1.175201194    |        ¯6○1.543080635
1                       | 1
```

## Pythagorean Functions

The Pythagorean functions `0○R`, `4○R`, and `¯4○R`, defined in Figure 17 on page 80, for nonnegative *real* `R` are related to the properties of a right triangle as indicated in Figure 19. They can also be defined as follows:

```
 0○R  ⇔  2○¯1○R or 1○¯2○R

 4○R  ⇔  6○¯5○R

¯4○R  ⇔  5○¯6○R
```

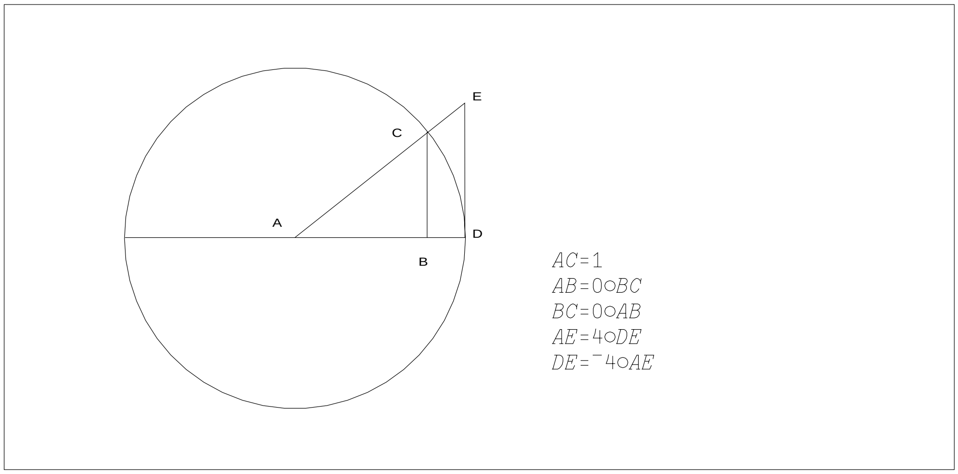


*Figure 19. Pythagorean Functions with Real Argument*

The principal values for the Pythagorean functions for *real* `R` are chosen in the interval `R≥0`.

## Complex Number Functions

The complex number functions (`L∊¯12 ¯11 ¯10 ¯9 ¯8` and `L∊8 9 10 11 12`) are defined in Figure 17 on page 80.

The formulas given for `¯8○R` and `8○R` in Figure 17 apply only to complex numbers with positive real and imaginary parts (the first quadrant). The phase of the result for other arguments is adjusted for proper placement of the cuts of the complex function.

```
      9 10 11 12○3J4
3 5 4 0.927295218
      8 ¯8○0J1
0 0
```

```
      ¯12○ ○1
¯1
      8 ¯8○2
0J¯2.236067977 0J2.236067977
```

The following identities apply:

```
¯8○R ←→ -8○R
   R ←→ ¯10 ¯11 +.○  9 11 ∘.○ R
   R ←→  ¯9 ¯12 ×.○ 10 12 ∘.○ R
```

## / Compress (from Slash)

`Z←LO/R` Selects subarrays along the last axis under the control of the vector `LO`.

`LO`: Simple scalar or vector, Boolean
`Z`: Nonscalar array

```
¯1↓ρZ  ←→  ¯1↓ρR
  ρρZ  ←→  ρρR
```

This is a special case of replicate (see “/ Replicate (from Slash)” on page 220).

Compress is often used to create a conditional branch expression, where `LO` is the condition (such as `X≥0`) and `R` represents a statement number—for example, `→(X≥0)/END`.

```
      1 1 0 0 1/'STRAY'
STY
```

```
      Q←3 4ρ⍳12
      Q
1  2  3  4
5  6  7  8
9 10 11 12
      1 0 1 0/Q
1  3
5  7
9 11
```

***Selective Specification:*** Compress can be used for selective specification:

```
      M←3 2ρ⍳6
      M
1 2
3 4
5 6

      (1 0/M)←'ABC'
      M
A 2
B 4
C 6
```

## /[] ⌿[] Compress with Axis (from Slash)

`Z←LO/[X]R` Selects subarrays along the X axis under the control of the vector `LO`.

`LO`: Simple scalar or vector, Boolean
`Z`: Nonscalar array

```
¯1↓ρZ  ←→  ¯1↓ρR
  ρρZ  ←→  ρρR
```

This is a special case of replicate with axis (see “/[] ⌿[] Replicate with Axis (from Slash)” on page 222).

```
      N←3 2 4ρ'HIGHLOW HOT COLD UP  DOWN'
      N
HIGH
LOW

HOT
COLD

UP
DOWN

      1 0/[2]N
HIGH

HOT

UP
      1 0 1/[1]N
HIGH
LOW

UP
DOWN
```

***Applied to First Axis:*** The symbol ⌿ is an alternate symbol for /.

***Selective Specification:*** Compress with axis can be used for selective specification:

```
      M←3 2⍴⍳6
      M
1 2
3 4
5 6

      T←2 2⍴'ABCD'
      (1 0 1/[1]M)←T
      M
A B
3 4
C D
```

# + Conjugate

`Z←+R`  Z is R with its imaginary part negated.

R and Z: Numeric

*Scalar Function*

For real R, conjugate returns its argument unchanged.

```
      +¯4
¯4
      +4 2.3 ¯3 ¯.7
4 2.3 ¯3 ¯0.7
```

```
      +1J2
1J¯2
      1J2×+1J2
5
```

# ? Deal

`Z←L?R` Selects `L` integers at random from the population `⍳R` without replacement.

`L` and `R`: Simple scalar or one-item vector, nonnegative integer
`Z`: Simple vector, integer in set `⍳R`

Implicit arguments: `⎕IO` and `⎕RL`
`ρZ   ←→  ,L`
`ρρZ   ←→  ,1`

The value of `L` must be between `0` and `R`, inclusive. Items are selected without replacement.

If `L=R`, `Z` is a random permutation of the integers `⍳R`.

The result depends on the value of `⎕RL`. A side effect of deal is to change the value of `⎕RL` (random link).

Both examples below show the value of `⎕RL` prior to execution of the function. To duplicate these results, specify `⎕RL` to be this value.

```
      ⎕IO←1
      ⎕RL
1474833169
      5?10
5 1 2 4 6

      ⎕RL
197493099
      10?10
4 6 3 1 2 10 5 7 9 8
```

```
      ⎕IO←0
      ⎕RL
1474833169
      5?10
4 0 1 3 5

      ⎕RL
197493099
      10?10
3 5 2 0 1 9 4 6 8 7
```

## ⊥ Decode

`Z←L⊥R` Yields the values of array `R` evaluated in a number system with radices `L`.

`L`, `R`, and `Z`: Simple numeric array

```
 ρZ   ←→ (¯1↓ρL),1↓ρR
ρρZ   ←→ (0⌈¯1+ρρL)+(0⌈¯1+ρρR)
```

***Polynomial Evaluation:*** In its simplest form (with scalar `L` and vector `R`), decode determines the value of a polynomial evaluated at `L`. `R` defines the coefficients of the polynomial arranged in descending order of powers on the powers of `L`. For example, the expression `3⊥1 2 1` evaluates the polynomial $x^2+2x+1$ at `3`.

```
      3⊥1 2 1
16
```

```
      1J1⊥1 2 3 4
5J9
```

***Base Value:*** If each item of `R` is a nonnegative integer less than `L`, decode determines the base-10 equivalent of a number stated in base-`L`. The digits of the base-`L` number are stated as the items of `R`. Sometimes, therefore, decode is referred to as the *base value* function. For example, the following expression determines the base-10 equivalent of `1111`-base 2.

```
      2⊥1 1 1 1
15
```

***General Decode:*** Decode is defined in terms of the inner product for any valid nonscalar `L` and `R` after extension of length 1 axes.

```
L ⊥ R ←→ ((ρL)↑⌽1,×\⌽1↓[ρρL]L) +.× R
```

***Conformability:*** Scalar arguments are treated as one-item vectors. Conformability requires that `¯1↑ρL ←→ 1↑ρR`. If either the first axis of `R` or the last axis of `L` is 1, it is extended (by replication of the item) as necessary to match the length of the other argument.

```
      L←2 1ρ2 10
      L
 2
10
      R←3 2ρ1 4 0 3 1 2
      R
1 4
0 3
1 2
```

```
      L⊥R
  5  24
101 432
      24 60 60⊥2 23 12
  8592
```

The example in the second column shows an evaluation in a mixed radix system. It determines the number of seconds in 2 hours, 23 minutes, and 12 seconds.

# ≡ Depth

`Z←≡R` Reports levels of nesting: `0` for a simple scalar; for other arrays, `1` plus the depth of the item with the maximum depth.

`Z`**:** Simple scalar, nonnegative integers

```
 ρZ   ↔  Empty
ρρZ   ↔  ,0
```

For a nonempty array, depth shows the degree of nesting:

Depth is 0 when `R` is a simple scalar.

```
      ≡5
0
```

```
      ≡'A'
0
```

Depth is 1 when `R` is a simple, nonscalar array. `R` contains only simple scalars as items.

```
      ≡2 2ρι4
1
```

```
      ≡3 2 4 5ρι120
1
```

Depth is *n* when `R` contains, as an item, at least one array of depth *n*-1. It may contain other arrays of lesser depths as well.

```
      B←'JIM' 'AL' 'EV'
      ρB
3
      ≡B
2
      ≡¨B
1 1 1
```

```
      C←'AB' 1 2 3
      ρC
4
      ≡C
2
      ≡¨C
1 0 0 0
```

```
      D←'ONE' 'TWO' ('BUCKLE' ('MY' 'SHOE'))
      ρD
3
      DISPLAY D
.→-------------------------------.
|.→--..→--..→-------------------.|
||ONE||TWO||.→-----..→---------.||
|'---''---'||BUCKLE||.→-..→---.|||
|          |'------'||MY||SHOE||||
|          |        |'--''----'|||
|          |        '∊---------'||
|          '∊-------------------'|
'∊-------------------------------'
```

```
      ≡D
4
      ≡¨D
1 1 3
```

```
      ≡¨¨D
 0 0 0  0 0 0  1 2
```

For empty R, the depth is ≡⊂↑R.

```
      ≡⍳0
1
      ↑⍳0
0
      ≡' '
1
      ↑' '
```

(*blank character*)

```
      H←0⍴⊂1 2 3
      ⍴H
0
      ≡H
2
      ↑H
0 0 0
```

```
      Q←0⍴15(⊂1 2 3)
      ⍴Q
0
      ≡Q
1
      ↑Q
0
```

```
      S←0⍴⊂(1 2 3(3 4))5 6
      ⍴S
0
      ≡S
4
      ↑S
  0 0 0  0 0   0 0
```

# × Direction

`Z←×R` Yields the number of magnitude 1 with the same phase as `R` for nonzero `R`. If `R` is 0, `Z` is 0.

`R` and `Z`: Numeric

*Scalar Function*

Formally for all `R`: `Z←R÷|R`

For real `R`, `×R` is often called *signum* and yields the following values:

| R | Z |
|---|---|
| negative | ¯1 |
| zero | 0 |
| positive | 1 |

```
      ×¯5
¯1
      ×¯4 0 4
¯1 0 1
```

```
      ×3J4
0.6J0.8
      ×0J1 0J¯1
0J1 0J¯1
```

# ⊃ Disclose

| | |
|---|---|
| `Z←⊃R` | Structures the items of *R* into an array, whose rightmost axes come from the axes of the items of *R*. |

```
 (⍴Z) ←→ (⍴R),↑⌈/(⍴¨(,R),⊂↑R)~⊂⍳0
(⍴⍴Z) ←→ (⍴⍴R)+↑⌈/⍴¨⍴¨(,R),⊂↑R
```

All items of *R* must be scalars and/or arrays of the same rank. It is not necessary that nonscalar items have the same shape.

In the identities for rank and shape, the `⊂↑R` takes care of the empty case.

***Shapes of Items the Same:*** If all items of *R* have the same shape, the last `⍴⍴↑R` axes of the result are filled with the items of *R*.

```
      V←(2 3 4) (5 6)
      ⊃V
2 3 4
5 6 0
```

In the following example, the last axis (the rank of the first item of *R* is 1) is filled with the items of *R*, taken in row-major order.

```
      R←2 3⍴(⍳4)'ABCD' '****'(5 6 7 8)'EFGH' '∆∆∆∆'
      R
 1 2 3 4  ABCD ****
 5 6 7 8  EFGH ∆∆∆∆

      ⍴R
2 3
      ≡R
2

      Z←⊃R
      Z
1 2 3 4
A B C D
* * * *

5 6 7 8
E F G H
∆ ∆ ∆ ∆

      ⍴Z
2 3 4
      ≡Z
1
```

***Shapes of Items Differ:*** If items of *R* are scalar or have different shapes, each is padded to a shape that represents the greatest length along each axis of all items of *R*; that is, the shape of each item is padded to `↑⌈/(,⍴¨R)~⊂⍳0`.

Each item's corresponding fill item is used for its new positions.

```
      E←(2 4⍴⍳8) 9 (3 2⍴'ABCDEF')
      E
 1 2 3 4   9  AB
 5 6 7 8      CD
              EF

      N←⊃E
      N
1 2 3 4
5 6 7 8
0 0 0 0

9 0 0 0
0 0 0 0
0 0 0 0

A B
C D
E F

      ⍴N
3 3 4
      ≡N
1
```

Because of this padding, using disclose on a vector of vectors is a convenient way to create a simple matrix without needing to know how many columns to specify. For example:

```
      D←'WHEEL' 'OF' 'FORTUNE'
      V←⊃D
      V
WHEEL
OF
FORTUNE

      ⍴V
3 7
      ≡V
1
```

***Relationship to Disclose with Axis:*** After padding and ignoring scalar extension, disclose is related to disclose with axis as follows:

```
⊃R ←→ ⊃[(⍴⍴R)+⍳⍴⍴↑R]R
```

***Relationship to Enclose:*** Disclose is the left inverse of enclose:

```
R ←→ ⊃⊂R
```

# ⊃[ ] Disclose with Axis

`Z←⊃[X]R` Structures the items of *R* into an array. *X* defines the axes of *Z*, into which items of *R* are structured.

*X*: Simple scalar or vector, nonnegative integers

Implicit argument: `⎕IO`

```
(⍴Z)[,X]   ←→  ↑⌈/(⍴¨(,R),⊂↑R)~⊂⍳0
     ⍴⍴Z   ←→  (⍴⍴R)+⌈/∊⍴¨⍴¨(,R),⊂↑R
```

All items of *R* must be scalars and/or arrays of the same rank. It is not necessary that nonscalar items have the same shape.

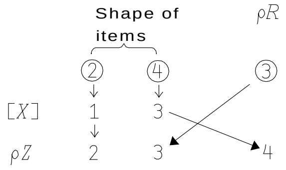
*X* specifies the axes of the result that are filled with the disclosed items of *R*. The number of items in *X* must be `⌈/∊⍴¨⍴¨R`. The values of *X* must be contained in `⍳(⍴⍴R)+⌈/∊⍴¨⍴¨R`.

```
      H←'ABCD' (1 2 3 4) 'WXYZ'
```

```
      Z←⊃[1]H
      Z
A 1 W
B 2 X
C 3 Y
D 4 Z
      ⍴Z
4 3
      ≡Z
1
```

```
      W←⊃[2]H
      W
A B C D
1 2 3 4
W X Y Z
      ⍴W
3 4
      ≡W
1
```

The shape of the nonscalar items of *R* and the shape of *R* itself map as indicated by *X* to form the shape of the result. The diagram below shows the mapping of axes for `⊃[1 3]R`, where *R* is a three-item vector whose items are matrixes of shape 2 4.



Indicates axes of *Z* to receive corresponding shape.

The following examples show disclose with axis for various axes applied to the three-item vector of matrixes described below.

```
      Z←2 4⍴'PA' 'MA' 'WE' 'BY' 'IT' 'UP' 'ON' 'HI'
      R←(2 4⍴⍳8) (2 4⍴'ABCDEFGH') (Z)
      DISPLAY R
.→-----------------------------------------.
| .→------. .→---. .→--------------------. |
| ↓1 2 3 4| ↓ABCD| ↓ .→-. .→-. .→-. .→-. | |
| |5 6 7 8| |EFGH| | |PA| |MA| |WE| |BY| | |
| '∼------' '----' | '--' '--' '--' '--' | |
|                  | .→-. .→-. .→-. .→-. | |
|                  | |IT| |UP| |ON| |HI| | |
|                  | '--' '--' '--' '--' | |
|                  '∊--------------------' |
'∊-----------------------------------------'
```

```
      B←⊃[1 2]R
      ⍴B
2 4 3

      B
 1 A PA
 2 B MA
 3 C WE
 4 D BY

 5 E IT
 6 F UP
 7 G ON
 8 H HI

      ≡B
2
```

```
      Y←⊃[1 3]R
      ⍴Y
2 3 4

      Y
  1  2  3  4
  A  B  C  D
 PA MA WE BY

  5  6  7  8
  E  F  G  H
 IT UP ON HI
```

```
      V←⊃[2 3]R
      ⍴V
3 2 4

      V
  1  2  3  4
  5  6  7  8

  A  B  C  D
  E  F  G  H

 PA MA WE BY
 IT UP ON HI
```

***Order of Axes:*** The order in which the axes are listed in *X* affects the shape of the result.

```
      N←⊃[2 1]R
      N
 1 A PA
 5 E IT

 2 B MA
 6 F UP

 3 C WE
 7 G ON

 4 D BY
 8 H HI
```

```
      M←⊃[3 1]R
      M
  1  5
  A  E
 PA IT

  2  6
  B  F
 MA UP

  3  7
  C  G
 WE ON

  4  8
  D  H
 BY HI
```

```
      P←⊃[3 2]R
      P
  1  5
  2  6
  3  7
  4  8

  A  E
  B  F
  C  G
  D  H

 PA IT
 MA UP
 WE ON
 BY HI
```

***Shapes of Items Differ:*** If items of *R* are scalar or have different shapes, each is padded to a shape that represents the greatest length along each axis of all items in *R*; that is, after padding the shape of each item is `↑⌈/(,⍴¨R)~⊂⍳0`.

Each item's corresponding fill item is used for its new positions.

```
      Q←(⍳3) 'JUMP'
      N←⊃[1]Q
      N
1 J
2 U
3 M
0 P

      E←(⍳5) 'JUMP'
      J←⊃[1]E

      J
1 J
2 U
3 M
4 P
5
```

```
      S←(2 6⍴'ABCDEFGHIJKL') (3 4⍴⍳12)
      S
 ABCDEF    1  2  3  4
 GHIJKL    5  6  7  8
           9 10 11 12
      ⍴S
2
      ⍴¨S
 2 6  3 4
      ⌈/(⍴¨S)~⊂⍳0
 3 6
      D←⊃[2 3]S
      ⍴D
2 3 6
      D
A  B  C  D E F
G  H  I  J K L
(2 rows of blanks)
1  2  3  4 0 0
5  6  7  8 0 0
9 10 11 12 0 0
```

***Empty Axis Needed:*** If all items of R are scalars, X must be empty.

```
      T←⊂¨'ONE' 'FOUR' 'THREE'
      ≡¨T
2 2 2
      ⊃[⍳0]T
 ONE FOUR THREE
```

***Relationship to Enclose with Axis:*** Disclose with axis is the left inverse of enclose with axis:

```
      R ←→ ⊃[X]⊂[X]R
```

# ÷ Divide

`Z←L÷R` Divides `L` by `R`.

`L`, `R`, and `Z`: Numeric

*Scalar Function*

Divide is the arithmetic division function.

```
      3÷2
1.5
      9 4 7 10÷.25
36 16 28 40
```

```
      0J12÷4
0J3
      .3 5 1÷0J1 ¯2 1
0J¯0.3 ¯2.5 1
```

If `R` is 0, `L` must also be `0`. The expression `0÷0` is defined in APL2 to be `1`.

```
      0÷5
0
      0÷0
1
```

```
      5÷0
DOMAIN ERROR
      5÷0
      ^^
```

## ↓ Drop

`Z←L↓R` Removes subarrays from the beginning or end of the `I`th axis of `R`, according to whether `L[I]` is positive or negative.

`L`: Simple scalar or vector, integer
`Z`: Nonscalar array

```
 ρZ   ←→  0⌈(ρR)-|L
ρρZ   ←→  (ρ,L)⌈ρρR
```

***Specifying the Amount to Drop:*** If `L` is a scalar, it is treated as a one-item vector; if `R` is a scalar, it is treated as an array of shape `(ρL)ρ1`. Then:

For `L[I]>0`, drop removes `L[I]` subarrays from the beginning of the `I`th axis of `R`.

For `L[I]<0`, drop removes `|L[I]` subarrays from the end of the `I`th axis of `R`.

For `L[I]=0`, no subarrays are removed from the `I`th axis.

```
      3↓12 31 45 10 57       |       ¯3↓12 31 45 10 57
10 57                        |      12 31
```

***Nonscalar Right Argument:*** For nonscalar `R`, `L` must have the same number of items as `R` has rank: `(ρ,L) = ρρR`.

```
      A←3 5ρ'STRIPERODEPLANT'
      B←'STOREFIRSTMIGHTHATER'
      B←'SHEETTHEREMETROERASE'
      B←3 4 5ρB,'BREADOTHERANVILEVADE'
```

```
      A                  |         B          |        C←¯1 2 ¯2
STRIP                    |   STORE            |
ERODE                    |   FIRST            |   (means drop the
PLANT                    |   MIGHT            |   last plane and
      1 2↓A              |   HATER            |   first two rows
ODE                      |                    |   and last two
ANT                      |   SHEET            |   columns from the
                         |   THERE            |   remaining planes)
                         |   METRO            |
                         |   ERASE            |         C↓B
                         |                    |    MIG
                         |   BREAD            |    HAT
                         |   OTHER            |
                         |   ANVIL            |    MET
                         |   EVADE            |    ERA
```

The number of subarrays dropped does not affect the rank of the result.

```
      K←3 2 4⍴'ABCDEFGH',(⍳8),'abcdefgh'
```

```
      K
A B C D
E F G H

1 2 3 4
5 6 7 8

a b c d
e f g h
```

```
      Z←2 1 3↓K
      Z
h
      ⍴Z
1 1 1
```

***Dropping None:*** If L[I] is zero, no subarrays are removed from the Ith axis.

```
      0↓'INTACT'
INTACT
```

```
      0 2↓3 5⍴⍳15
 3  4  5
 8  9 10
13 14 15
```

***Overdrop:*** If L[I] equals or exceeds the length of the Ith axis, the resulting shape has an Ith axis whose length is zero.

```
    W←5↓23 41 73 26
    ⍴W
0
```

```
      H←2 3⍴'ABCDEF'
      Y←3 1↓H
      ⍴Y
0 2
      M←2 3↓H
      ⍴M
0 0
```

***Scalar Right Argument:*** For scalar R, L may have any length. The length of ,L determines the rank of the result.

```
      J←0↓4
      J
4
      ⍴J
1
```

```
      K←0 0 0↓4
      K
4
      ⍴K
1 1 1
```

***Effect on Depth:*** Drop does not affect the depth of any selected item. The depth of the result is less than or equal to the depth of the argument, except when the right argument is a simple scalar.

```
      D←'A' 'AN'('ANT' 'ANTE')
      D
 A  AN    ANT ANTE
      ⍴D
3
      ≡D
3

      S←¯1↓D
      S
 A  AN
      ≡S
2
      T←¯2↓D
      T
A
      ⍴T
1
      ≡T
1
```

***Selective Specification:*** Drop can be used for selective specification:

```
      U←'ABCDE'
      (2↓U)←⍳3
      U
AB 1 2 3
```

```
      V←3 4⍴'ABCDEFGHIJKL'
      V
ABCD
EFGH
IJKL
      (1 ¯1↓V)←2 3⍴⍳6
      V
A B C D
1 2 3 H
4 5 6 L
```

# ↓[ ] Drop with Axis

`Z←L↓[X]R` Removes subarrays from the beginning or end of the `X[I]`th axis of `R`, according to whether `L[I]` is positive or negative.

`L`: Simple scalar or vector, integer
`R` and `Z`: Nonscalar array
`X`: Simple scalar or vector; nonnegative integers: `X∊⍳⍴⍴R`; or empty

Implicit argument: `⎕IO`

```
(⍴Z)[,X]  ↔  0⌈(⍴R)[,X]-|L
     ⍴⍴Z  ↔  ⍴⍴R
```

Drop with axis is similar to drop except that subarrays are removed only from the axes indicated by `X`. The shape along axes not selected by `X` remains unchanged.

***Drop with Axis Compared with Drop:*** The following identity states the relationship between drop and drop with axis:

```
L↓R ↔ L↓[⍳⍴⍴R]R
```

```
      A←3 4⍴'FOLDBEATRODE'
      A
FOLD
BEAT
RODE
      1↓[1]A
BEAT
RODE
      1↓[2]A
OLD
EAT
ODE
```

```
      1 0↓A
BEAT
RODE
      0 1↓A
OLD
EAT
ODE
```

***Permitted Axes:*** Multiple axes indicated by `X` need not be in ascending order; however, no axis may be repeated. `L[I]` defines the number of subarrays to drop from the `X[I]`th axis.

```
      Q←3 2 4⍴'ABCDEFGH',(⍳8),'abcdefgh'
```

```
      Q
A B C D
E F G H

1 2 3 4
5 6 7 8

a b c d
e f g h
```

```
      1 ¯1↓[2 3]Q
E F G

5 6 7

e f g
      1 ¯1↓[3 2]Q
B C D

2 3 4

b c d
```

***Effect on Depth:*** Drop with axis does not affect the depth of any selected item. The depth of the result is less than or equal to the depth of the argument.

```
      T←'W' 'WE'('WEE' 'WEED')'B' 'BE'('BEE' 'BEEP')
```

```
      U←2 3⍴T

      U
 W WE   WEE WEED
 B BE   BEE BEEP
      ≡U
3

      Q←1↓[1]U
      ≡Q
3
      Q
 B BE   BEE BEEP
      ⍴Q
1 3
```

```
      M←¯1↓[2]U
      ≡M
2
      M
 W WE
 B BE
      ⍴M
2 2
      N←¯2↓[2]U
      ≡N
1
      N
W
B
      ⍴N
2 1
```

***Selective Specification:*** Drop with axis can be used for selective specification:

```
      V←3 4⍴'ABCDEFGHIJKL'
      V
ABCD
EFGH
IJKL
      (1↓[1]V)←2 4⍴⍳8
      V
A B C D
1 2 3 4
5 6 7 8
```

## ¨ Each (Dyadic)

`Z←L LO¨ R` Applies the function *LO* between corresponding pairs of items of *L* and *R*.

*LO*: Dyadic function

```
 ρZ   ↔  ρR or ρL
ρρZ   ↔  ρρR or ρρL
```

***Conformability of Arguments:*** Either *L* and *R* must have the same shape, or one may be a scalar or one-item vector. A scalar or a one-item vector argument is applied against each item.

If *R* is not empty:

`I⊃Z ↔ (I⊃L) LO I⊃R`

for every scalar *I* for which `I⊃L` and `I⊃R` are defined.

```
      Z←4 6ρ¨'ME' 'YOU'
      Z
 MEME YOUYOU
      ρZ
2
      ρ¨Z
 4  6
      ≡Z
2
```

```
      'SET',¨'HES'
 SH EE TS
```

***Each and a Scalar Argument:*** The conformability for each means that if one argument is a scalar and the other is not, each pairs its operand (*LO*) with the item inside the scalar and each item of the nonscalar argument. This fact can be used to pair any array (*A*) with each item of another array (*B*) by enclosing *A*.

```
      (⊂A)F¨B
```

applies *F* with *A* as the left argument and each item of *B*, in turn, as the right argument.

```
      2ρ¨3 4 5
3 3  4 4  5 5
      (⊂2 3)ρ¨4 6
4 4 4  6 6 6
4 4 4  6 6 6
```

***Each and Primitive Dyadic Scalar Functions:*** Applied to the primitive dyadic scalar functions, the operator each has no effect; that is:

`L LO¨ R ↔ L LO R`

The primitive scalar functions are listed in Figure 9 on page 51.

***Each Substitutes for Looping:*** Each has an effect similar to the DO loop in other programming languages. It can be used to eliminate most looping in APL2 functions. For an example, see "¨ Each (Monadic)" on page 109.

***Empty Argument:*** If `L` or `R` is empty, the function `LO` is not applied. Instead, a related function called the *fill function* of `LO` is applied.

Either `L` or `R` or both can be empty. If one argument is not empty, it must be a scalar item and the first (`↑`) of that scalar is presented to the fill function as an argument. An empty argument is presented to the fill function as `↑L` or `↑R` (the prototype). That is, if either `L` or `R` or both are empty:

For `Z←L LO¨ R`, `Z` is `Sρ⊂(↑L) FF (↑R)`.

Where:

`S` is the shape of the empty argument.

`FF` is the fill function of `LO`.

For example:

```
      Z←5↑¨0ρ⊂0 0 0
      ρZ
0
      ρ↑Z
5
```

Figure 20 on page 110 gives all the fill functions for the primitive functions and defined operations.

Some functions derived by inner product or reduction may not have fill functions. An attempt to apply such a function to each item of an empty array generates a `DOMAIN ERROR`.

## ¨ Each (Monadic)

`Z←LO¨R` Applies the function `LO` to *each* item of `R`.

`LO`: Monadic function

```
 ρZ   ←→  ρR
ρρZ   ←→  ρρR
```

If `R` is not empty:

`I⊃Z ←→ LO I⊃R`

For every scalar `I` for which `I⊃R` is defined.

```
      Z←ρ¨'TOM' 'DICK'
      Z
 3  4
      ρZ
2
      ≡Z
2
```

```
      W←⍳¨1 2 3 4
      W
 1  1 2  1 2 3  1 2 3 4
      ρW
4
      ≡W
2
```

***Each and Primitive Monadic Scalar Functions:*** Applied to the primitive monadic scalar functions, the operator each has no effect; that is:

`LO¨ R ←→ LO R`

The primitive scalar functions are listed in Figure 9 on page 51.

***Each Substitutes for Looping:*** Each has an effect similar to the DO loop in other programming languages. It can be used to eliminate most looping in APL2 functions. For example, the loop shown below applies the function `F` to each item of a vector `V` and accumulates the results in a vector. This loop can be replaced with an application of the operator each:

```
⋮
Z←0ρV
L1:→(0=ρV)/L1X
Z←Z,⊂F↑V
V←1↓V
→L1
L1X:
⋮
```

The above loop can be replaced by:

```
Z←F¨ V
```

***Empty Argument:*** If *R* is empty, the function *LO* is not applied. Instead, a related function called the *fill function* of *LO* is applied with argument ↑*R* (the prototype of *R*). This result is used as a prototype of the empty array of ⍴*R* .

The identity is:

*LO*¨*R* ←→ (⍴*R*)⍴⊂ *fill fn* ↑*R*

where:

| | |
|---|---|
| *LO* | Is any function for which a fill function is defined |
| *fill fn* | Is its related fill function |

```
      DISPLAY ⌹¨0⍴⊂2 3⍴0
.⊖------.
| .→--. |
| ↓0 0| |
| |0 0| |
| |0 0| |
| '∼--' |
'∊------'
```

Figure 20 gives expressions that are the fill functions. All defined fill functions are given below. Remember that the *prototypes* of the arguments of the function become the arguments of the fill function. The result of the fill function becomes the prototype of the result of the application of the function or derived function.

| | |
|---|---|
| Scalar Functions | *Z*←(*R*)≠(*L*) |
| Matrix Inverse | *Z*←⍉*R* |
| Matrix Divide | *Z*←((1↓⍴*R*),1↓⍴*L*)⍴0 |
| Other Primitive Functions | The function itself |
| Defined Operations | *Z*←*R* (the identity function) |

*Figure 20. Fill Functions*

A function derived by each or outer product has the same fill function as its operand, if the operand has a fill function.

Some functions derived by inner product or reduction may not have fill functions. An attempt to apply such a function to each item of an empty array generates a *DOMAIN ERROR*.

# ⊂ Enclose

`Z←⊂R` Creates a scalar array whose only item is `R`.

`Z`: Scalar array

```
 ⍴Z   ←→  ⍳0
⍴⍴Z   ←→  ,0
```

If `R` is a simple scalar, `⊂R` is `R`. If `R` is not a simple scalar, the depth of `⊂R` is `1+≡R`.

```
      A←2 3 4⍴⍳24
      A
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
      ⍴A
2 3 4
      ⍴⍴A
3
      ≡A
1
```

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

 13 14 15 16
 17 18 19 20
 21 22 23 24
      ⍴Z
(empty)
      ⍴⍴Z
 0
      ≡Z
 2
```

***Compared to Vector Notation:*** For `A`, `B`, and `C`:

`(A B C) ←→ ((⊂A),(⊂B),(⊂C))`

```
      D←'ON' 'UP' 'BY'
      ⍴D
3
      ≡D
2
```

Enclose is used to create a scalar whose only item is *R*. This scalar can replace a scalar subarray selected by indexing. It is also subject to scalar extension as an argument of a scalar function or of an each-derived function.

```
      S←15 0 29
      ρS
3
      ≡S
1
      S[2]←'NONE'
RANK ERROR
      S[2]←'NONE'
      ^    ^
      S[2]←⊂'NONE'
      S
 15 NONE 29
      ρS
3
      ≡S
2
```

```
      T←3 5+⊂0 1
      T
 3 4  5 6
      ρT
2

      Q←'LOUIS' 'CROIX'
      Z←(⊂'ST. '),¨Q
      Z
 ST. LOUIS ST. CROIX
```

Enclose and ravel can be used to create a nested one-item vector:

```
      W←,⊂15 0 29
      W
 15 0 29
      ρW
1
```

Ravel and enclose can be used to create a scalar containing a one-item vector.

```
      Y←⊂,5
      ρρY
0
      ≡Y
2
```

***Relationship to Disclose:*** Disclose is the left inverse of enclose:

```
R ←→ ⊃⊂R
```

## ⊂[ ] Enclose with Axis

*Z*←⊂[*X*]*R* Yields an array whose items are the contiguous subarrays along the set of axes indicated by *X*. That is, the set of axes indicated by *X* is enclosed.

*X*: Simple scalar or vector, nonnegative integer.

If *X* is nonempty, *X*∊⍳ρρ*R*.

Implicit argument: ⎕*IO*

```
 ρZ  ←→ (ρR)[(⍳ρρR)~X]  ρ↑Z  ←→ (ρR)[,X]  ρρZ  ←→
(ρρR)-ρ,X
```

```
      A←2 3ρ⍳6
      A
1 2 3
4 5 6

      Z←⊂[1]A
      Z
 1 4  2 5  3 6
      ρZ
3
      ρ¨Z
2  2  2
      ≡Z
2

      Y←⊂[2]A
      Y
 1 2 3  4 5 6
      ρY
2
      ρ¨Y
 3  3
      ≡Y
2
```

```
      B←3 4ρ'PINEODORDATA'
      B
PINE
ODOR
DATA
      X←⊂[1]B
      X
 POD IDA NET ERA
      ρX
4
      ρ¨X
3  3  3  3
      ≡X
2

      W←⊂[2]B
      W
 PINE ODOR DATA
      ρW
3
      ρ¨W
 4  4  4
      ≡W
2
```

***Empty Axis:*** An empty axis has no effect on R if R is a simple array. If R is nested, an empty axis increases the depth of R by enclosing each item without affecting its shape: ⊂[⍳0]R ←→ ⊂¨R.

```
      C←2 3⍴⍳6
      V←⊂[⍳0]C
      V
1 2 3
4 5 6
      ⍴V
2 3
      ≡V
1

      Q←2 3⍴'CAT' 'DOG' 'FOX' 'COW' 'BAT' 'YAK'
      Q
 CAT DOG FOX
 COW BAT YAK
      ⍴Q
2 3
      ≡Q
2
      H←⊂[⍳0]Q
      H
  CAT   DOG   FOX
  COW   BAT   YAK
      ⍴H
2 3
      ≡H
3
```

***Order of Axes:*** The order in which the axes are listed in X affects the shape of each item of Z.

```
      S←2 3 4⍴'LESSSOMENONEMOREMANYMOST'
      S
LESS
SOME
NONE

MORE
MANY
MOST

      P←⊂[2 3]S
      P
 LESS  MORE
 SOME  MANY
 NONE  MOST
      ⍴P
2
      ⍴¨P
 3 4  3 4
```

```
      ≡P
2

      Q←⊂[3 2]S
      Q
 LSN  MMM
 EOO  OAO
 SMN  RNS
 SEE  EYT
      ⍴Q
2
      ⍴¨Q
 4 3  4 3
      ≡Q
2
```

If all the axes of *R* are included in *X*, then:

`⊂(⍋X)⍉R ↔ ⊂[X]R`

If *X* is `⍳⍴⍴R`, then:

`⊂R ↔ ⊂[X]R`

```
      T←2 3⍴⍳24
      T
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
      ⍴T
2 3 4

      J←⊂[1 2 3]T
      J
  1  2  3  4
  5  6  7  8
  9 10 11 12

 13 14 15 16
 17 18 19 20
 21 22 23 24
      ≡J
2
```

```
      ⊂[1 3 2]T
  1  5  9
  2  6 10
  3  7 11
  4  8 12

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

      (⊂[1 3 2]T)≡⊂(⍋ 1 3 2)⍉T
1
```

***Relationship to Disclose with Axis:*** Disclose with axis is the left inverse of enclose with axis:

`R ↔ ⊃[X] ⊂[X] R`

# ⊤ Encode

`Z←L⊤R`   Yields the representation of `R` in the number system whose radices are `L`.

`L`, `R`, and `Z`: Simple numeric array

```
 ⍴Z   ←→ (⍴L),⍴R
⍴⍴Z   ←→ (⍴⍴L)+⍴⍴R
```

***Representation of a Base-10 Number:*** For radices `L` having positive integer items, encode has an inverse relationship to decode, as follows:

```
L⊥(L⊤R) ←→ (×/L)|R
```

Thus, encode can be used to determine the representation of the base-10 number in a number system whose radices are defined by the vector `L`.

```
      2 2 2 2⊤15           │         24 60 60⊤8592
1 1 1 1                    │   2 23 12
```

The example in the second column is a mixed radix encoding of the number of hours, minutes, and seconds in 8592 seconds.

For `L<10`, `10⊥L⊤R` displays the base-`L` representation of `R` as a single number.

```
      2 2 2 2⊤15           │         10⊥2 2 2 2⊤15
1 1 1 1                    │   1111
```

The number of digits present for the encoding of `R` depends on the shape of `L`. If `L` has greater shape than needed, the result has leading `0`'s. If `L` has less shape than needed, the result is an incomplete representation.

```
        2 2 2 2 2⊤15       │         2 2 2⊤15
  0 1 1 1 1                │   1 1 1
```

For a single-base encoding, the expression `⌊1+L⍟(|R)+R=0` can be used to determine how many items `L` should contain for a complete representation of the scalar `R`.

```
      ((⌊1+2⍟135)⍴2)⊤135
1 0 0 0 0 1 1 1
```

No simple expression exists for predetermining the number of places in the result for a mixed-radix encoding. However, if `1↑L` is zero, the first item of the result captures any overflow if `L` is not long enough for a complete representation of `R`.

```
      24 60 60⊤162507
21 8 27
```

```
      0 24 60 60⊤162507
1 21 8 27
```

***General Encode:*** The basic definition of `L⊤R` concerns a vector `L` and a scalar `R` and yields a result with the shape of `L`. Encode is defined formally in terms of the function residue (`|`) but with `⎕CT←0`, as shown in the defined function `ENCODE` below:

```
     ∇ Z←L ENCODE R;I;⎕CT
[1]    ⎕CT←0
[2]    Z←0×L
[3]    I←⍴L
[4]    GO:→(I=0)/0
[5]    Z[I]←L[I]|R
[6]    →(L[I]=0)/0
[7]    R←(R-Z[I])÷L[I]
[8]    I←I-1
[9]    →GO
     ∇
```

For arguments of other ranks:

```
      Z← ⊃[1] (⊂[1]L) ∘.ENCODE R
```

```
      10 10 10⊤215 345 7
2 3 0
1 4 0
5 5 7
```

```
      L←4 2⍴8 2
      L
   8 2
   8 2
   8 2
   8 2
      L⊤15
   0 1
   0 1
   1 1
   7 1
```

# ∊ Enlist

Z←∊R  Creates a simple vector whose items are the simple scalars in R.

Z: Simple vector

 ⍴Z ←→ Number of simple scalars in R
⍴⍴Z ←→ ,1

The result of enlist is always a simple vector.

```
      C←'ALE' 'BEER' 'STOUT'
      Z←∊C
      Z
ALEBEERSTOUT
      ⍴Z
12
      ≡Z
1
```

The example below shows how enlist selects items from a nested array to form a simple vector.

```
      H←(2 2⍴⍳4)(2 2⍴(5 6(2 2⍴7 8 9 10)11))'ABCD'
      DISPLAY H
.→---------------------------.
| .→--. .→----------. .→---. |
| ↓1 2| ↓  5     6  | |ABCD| |
| |3 4| |           | '----' |
| '~--' | .→---.    |        |
|       | ↓7  8| 11 |        |
|       | |9 10|    |        |
|       | '~---'    |        |
|       '∊----------'        |
'∊---------------------------'

      ∊H
1 2 3 4 5 6 7 8 9 10 11 ABCD
      ⍴∊H
15
      ≡∊H
1
```

***Compared to Ravel:*** **,** Ravel, page 202, creates a vector from the items in *R*. If *R* is simple, the results of enlist and ravel are equivalent:

```
,R ←→ ∊R
```

***Selective Specification:*** Enlist can be used for selective specification.

```
      A←(10 20 30) 'AB'
      (∊A)←⍳5
      A
 1 2 3  4 5
```

# ⍎ Execute

Z←⍎R  Evaluates the statement represented by the character vector R.

R: Simple character scalar or vector

 ⍴Z ←→ Data dependent
⍴⍴Z ←→ Data dependent

R is taken to represent a valid statement which is evaluated.

```
      ⍎'⍳4'
1 2 3 4

      1+⍎'⍳4'
2 3 4 5
      ⍎'195÷5×13'
3
```

```
      ⍎'MATRIX←3 3⍴⍳9'
      MATRIX
1 2 3
4 5 6
7 8 9
      ⍎'''AGNES'''
AGNES
```

The last example in the right column shows that it takes three sets of quotation marks to specify a character vector for execution.

***Valueless Expression:*** If R is empty or represents a defined function or operator without explicit result, ⍎R has no value.

```
      ∇F X
[1]   Z←3×X
[2]   ∇

      S←⍎'F 2'
VALUE ERROR+
      S←⍎'F 2'
       ^^
```

```
      2×⍎''
VALUE ERROR+
      2×⍎''
        ^
```

***Conditional Execution:*** The statement R may be executed conditionally.

```
      V←⍳0
      ⍎(0=⍴V)/'''EMPTY'''
EMPTY
```

```
      CTR←0
      ⍎(1=CTR)/'124'
No value (i.e., not empty)

      CTR←1
      ⍎(1=CTR)/'124'
124
```

***Error Message:*** If the statement R results in an error, the error message includes lines showing the content of R and where the error occurred in R.

```
      ⍎'3 4×'
SYNTAX ERROR+
      3 4×
      ^   ^
      ⍎'3 4×'
      ^
```

**Execute with Branch Statements:** Execute applies to a branch statement only if the execute is the leftmost primitive in a statement and is applied without an operator. Here are two examples of illegal execute statements:

```
      L1←3

      2+⍎'→L1'       ⍝ NOT LEFTMOST PRIMITIVE
SYNTAX ERROR+
      →L1
      ^
      2+⍎'→L1'       ⍝ NOT LEFTMOST PRIMITIVE
       ^^

      ⍎¨'2+2' '→L1'  ⍝ APPLIED WITH OPERATOR
DOMAIN ERROR
      →L1
      ^
      ⍎¨'2+2' '→L1'  ⍝ APPLIED WITH OPERATOR
      ^^
```

# \ Expand (from Backslash)

`Z←LO\R` Expands the last axis of `R` under the control of the Boolean vector `LO`.

`LO`: Simple Boolean scalar or vector
`Z`: Nonscalar array

```
¯1↓ρZ  ↔  ¯1↓ρR
¯1↑ρZ  ↔  ρ,LO
  ρρZ  ↔  ρρR
```

Positions in `Z` that correspond to ones in `LO` are filled with items of `R`. Positions in `Z` that correspond to `0`'s in `LO` are filled with the fill item (`↑0ρ⊂↑R`).

```
      1 0 1 0 0 1\1 2 3
1 0 2 0 0 3

      H←(1 2) (3 4 5) 6
      1 0 1 1 0\H
 1 2  0 0  3 4 5  6  0 0
```

```
      1 0 1 0 0 1\'ABC'
A B  C

      K←1 (2 3) (4 5 6)
      1 0 1 1 0\K
 1 0  2 3  4 5 6  0
```

When applied to multidimensional arrays, expand treats each subarray along the last axis as a vector and expands it with a fill item appropriate for that subarray. For example:

```
      R←1 2 3 4 'A' 4 'C' 2 6
      R←R, 'X' 7 'Y' 1 'D' 'E'
      R←5 4ρR,5 'F' 'G' 'H' 'I'
      R
1 2 3 4
A 4 C 2
6 X 7 Y
1 D E 5
F G H I

      1 0 0 1 1 0 1\R
1 0 0 2 3 0 4
A     4 C   2
6 0 0 X 7 0 Y
1 0 0 D E 0 5
F     G H   I
```

***Conformability:*** If `¯1↑⍴R` is not `1`, it must be equal to `+/LO`. For scalar `R` or if `¯1↑⍴R` is 1, the following extensions are applied before the function is evaluated:

- If `R` is a scalar, it is treated as a one-item vector.
- If `¯1↑⍴R` is `1`, `R` is replicated along the last axis `+/LO` times.

```
      1 0 0\5
5 0 0
```

```
      S←3 1⍴7 8 9
      0 1 0\S
0 7 0
0 8 0
0 9 0
```

***Compared with Replicate:*** Expand is similar to replicate with negative `LO`. The following identities also exist for Boolean `LO`:

```
LO\R ←→ (LO-LO=0)/R

LO\R ←→ (¯1+2×LO)/R
```

```
      W←7 8 9
      1 0 0 1 0 1\W
7 0 0 8 0 9
```

```
      1 ¯2 1 ¯1 1/W
7 0 0 8 0 9
```

***Empty Arrays:*** If `LO` is empty, `R` must be a scalar or the shape along the last axis (`¯1↑⍴R`) must be `0` or `1`. If `R` is empty with a zero last axis, `LO` must consist entirely of `0`'s. If `R` is empty with a nonzero last axis, `+/,LO` must be `¯1↑⍴R`.

```
      Z←(⍳0)\2 0⍴0
      ⍴Z
2 0

      B←1 0 1\0 2⍴0
      B
      ⍴B
0 3
```

```
      A←(⍳0)\,[⍳0]6 7 8
      ⍴A
3 0

      C←0 0 0\2 0⍴0
      ⍴C
2 3
      C
0 0 0
0 0 0
```

***Selective Specification:*** Expand can be used for selective specification:

```
      M←'ABC'
      (1 0 1 0 1\M)←⍳5
      M
1 3 5
```

```
      N←2 3⍴⍳6
      N
1 2 3
4 5 6

      T←2 4⍴'ABCDEFGH'
      (1 0 1 1\N)←T
      N
ACD
EGH
```

## \[ ] ⍀[ ]Expand with Axis (from Backslash)

`Z←LO\[X]R` Expands the *X*th axis of *R* under the control of the Boolean vector *LO*.

*LO*: Simple Boolean scalar or vector
*R* and *Z*: Nonscalar array
*X*: Simple scalar or one-item vector, integer: `X∊⍳⍴⍴R`

Implicit Argument: `⎕IO`

```
(⍴Z)[,X]  ←→ ⍴,LO
     ⍴⍴Z  ←→ ⍴⍴R
```

Expand with axis is similar to expand, except that expansion occurs along the *X*th axis.

```
      R←2 3 4⍴⍳24
      ((,R)[1 3 14 16])←'ACDE'

      R
 A  2  C  4
 5  6  7  8
 9 10 11 12

13  D 15  E
17 18 19 20
21 22 23 24

      1 1 0 1\[2]R
 A  2  C  4
 5  6  7  8
    0     0
 9 10 11 12

13  D 15  E
17 18 19 20
 0     0
21 22 23 24
```

```
      F←2 2 2⍴⊂[2]8 2⍴⍳16
      F
 1 2    3 4
 5 6    7 8

 9 10   11 12
 13 14  15 16

      1 0 1\[2]F
 1 2    3 4
 0 0    0 0
 5 6    7 8

 9 10   11 12
 0 0    0 0
 13 14  15 16
```

```
      G←2 2 2ρ1 (2 3) 4 (5 6) 7 (8 9) 10 (11 12)
```

```
      G
  1   2 3
  4   5 6

  7   8 9
 10   11 12
```

```
      1 0 1\[2]G
  1   2 3
  0   0 0
  4   5 6

  7   8 9
  0   0 0
 10   11 12
```

***Conformability:*** If `(ρR)[X]` is not `1`, it must be equal to `+/LO`. For scalar `R` or if `(ρR)[X]` is `1`, the following extension is applied before the function is evaluated:

If `(ρR)[X]` is 1, `R` is replicated along the `X`th axis `+/LO` times.

```
      T←2 1 3ρι6
      T
1 2 3

4 5 6
```

```
      1 0 0 1\[2]T
1 2 3
0 0 0
0 0 0
1 2 3

4 5 6
0 0 0
0 0 0
4 5 6
```

***Applied to First Axis:*** The symbol `⍀` is an alternate symbol for `\[1]`. However, if `⍀` is followed by an axis (`⍀[X]`), it is treated as `\[X]`.

```
      M←3 4ρ'A' 'B' 1 'C' 2 3 4 5 6 7 8 9
```

```
      M
A B 1 C
2 3 4 5
6 7 8 9

      1 0 0 1 1⍀M
A B 1 C
    0
    0
2 3 4 5
6 7 8 9
```

```
      1 0 0 1 1\[1]M
A B 1 C
    0
    0
2 3 4 5
6 7 8 9
```

***Selective Specification:*** Expand with axis (from backslash) can be used for selective specification:

```
      M←3 2⍴⍳6
      M
1 2
3 4
5 6
  (1 1 0 0 1\[1]M)←5 2⍴-⍳10
      M
¯1  ¯2
¯3  ¯4
¯9 ¯10
```

# * Exponential

*Z*←**R* Determines the *R*th power of the base of the natural logarithms *e*, where *e* is approximately 2.7182818284590452.

*R* and *Z*: Numeric

*Scalar Function*

The exponential function is equivalent to *e***R*.

```
      *1
2.718281828
      *0
1
```

```
      *0J1
0.5403023059J0.8414709848
      *○0J1
¯1
```

# ! Factorial

`Z←!R` For positive integer *R*, yields the product of all positive integers through *R*.

For all numbers but negative integers, factorial yields the Gamma function of `R+1`.

*R*: Numeric, except for negative integers
*Z*: Numeric

*Scalar Function*

```
      !4
24
      !1 2 3 4 5
1 2 6 24 120
```

```
      ! 3J2
¯3.011540037J1.770168194
      !.05 ¯.05
0.9735042656 1.031453317
```

***Gamma Function:*** Factorial approximates the gamma function of (*n*+1):

$$\Gamma(n) = \int_0^\infty e^{-x} x^{n-1} dx$$

$$\Gamma(n+1) = n\Gamma(n) \qquad \text{if } n > 0$$

# ⍷ Find

*Z*←*L*⍷*R* Yields a Boolean array that maps to *R*. An item of *Z* is 1, where the pattern *L* begins in the corresponding position of *R*. Otherwise, an item of *Z* is 0.

**Note:** See the discussion of the )*PBS* command on page 444 for alternate ways to enter this character.

*Z*: Simple Boolean array

Implicit argument: ⎕*CT*

```
 ⍴Z   ←→  ⍴R
⍴⍴Z   ←→  ⍴⍴R
```

```
      'AB'⍷'ABABABABA'
1 0 1 0 1 0 1 0 0
      H←4 5⍴'ABCABA'
      H
ABCAB
AABCA
BAABC
ABAAB
```

```
      1 2 3⍷1 2 3 4 1 2 3
1 0 0 0 1 0 0
      K←2 3⍴'BCAABC'
      K
BCA
ABC
      K⍷H
0 1 0 0 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
```

***Rank of** L **Smaller Than Rank of** R*: If *L* has smaller rank than *R*, the search is performed along the last ⍴⍴*L* axes of *R*. That is, *L* is treated as being reshaped as ((*D*⍴1),⍴*L*)⍴*L*, where *D* is the difference in ranks: (⍴⍴*R*)-⍴⍴*L*.

With *H* specified as above:

```
      'BA'⍷H
0 0 0 0 0
0 0 0 0 0
1 0 0 0 0
0 1 0 0 0
```

***Rank of L Greater Than Rank of R:*** If L has larger rank than R, the pattern L cannot be found in R and all items of Z are 0.

```
      Q←2 3⍴'ABCABB'
      Q
ABC
ABB
      Q⍷'ABCABB'
0 0 0 0 0 0
```

***Nested Arrays:*** The pattern being searched for is found only if an exact match in structure and data (within comparison tolerance) exists:

```
      S←'GO' 'ON' 'GO' 'TO'
      S
 GO ON GO TO
      ⍴S
4
      ≡S
2
      'GO' 'TO'⍷S
0 0 1 0
      'GOTO'⍷S
0 0 0 0
```

***Deleting Multiple Blanks:*** ⍷ can be used to delete multiple blanks as follows:

```
      S←'AB   DEF'
      (~'  '⍷S)/S
AB DEF
```

# ↑ First

`Z←↑R` Selects the first item of *R* taken in row major order. If *R* is empty, yields the prototype of *R*.

`ρZ` ←→ Depends on shape of the first item
`ρρZ` ←→ Depends on rank of the first item

```
      A←'DO' 'RE' 'ME'
      Z←↑A
      Z
DO
      ≡Z
1
      ρZ
2

      Y←'ABCDE'
      ≡Y
1
      W←↑Y
      W
A
      ≡W
0
```

```
      B←(2 3) ((4 5 6) 7)
      ≡B
3
      J←↑B
      J
2 3
      ≡J
1

      C←⌽B

      C
  4 5 6  7   2 3
      S←↑C
      S
 4 5 6  7
      ρS
2
      ≡S
2
      ≡C[1]
3
```

***Empty Argument:*** If *R* is empty, first yields the prototype of *R*; that is:

`↑R ←→ ↑0ρ⊂↑R`.

```
      ↑⍳0
0
      D←0 3ρ(2 3ρ0) 0
      ρD
0 3
      T←↑D
      T
0 0 0
0 0 0
```

```
        ↑''
(a blank)
      H←0 2ρ(0 0) (0 0 0)
      H
      ρH
0 2
      U←↑H
      U
0 0
```

***Compared with Pick and Enclose:*** ⊃ Pick, page 195, selects any item from an array:

`↑R ←→ (⊂(⍴⍴R)⍴1)⊃R` (for nonempty `R`)

⊂ Enclose, page 111, creates a nested scalar whose only item is the argument array:

`R ←→ ↑⊂R`

***Selective Specification:*** First can be used for selective specification:

```
      K←'RED' 'WHITE' 'BLUE'
      K
 RED WHITE BLUE
      ⍴K
3
      (↑K)←'YELLOW'
      K
 YELLOW WHITE BLUE
      ≡K
2
```

# ⌊ Floor

| | |
|---|---|
| `Z←⌊R` | For real numbers, yields the largest integer that does not exceed `R` (within the comparison tolerance).<br>For complex numbers, depends on the relationship of the real and imaginary parts of `R`. |

`R` and `Z`: numeric
Implicit Argument: `⎕CT`

*Scalar Function*

The magnitude of the difference of a number and its floor is always less than 1. The examples below show floor applied to real `R`.

```
      ⌊2.3
2
```

```
      ⌊¯2.7 3 .5
¯3 3 0
```

For complex `R` of the form `A+0J1×B` (where `A` and `B` are real), the result depends on the relationship of the real (`A`) and imaginary parts (`0J1×B`) of `R` as follows:

| **If** | **Then `Z` Is** |
|---|---|
| `1>(A-⌊A)+B-⌊B` | `(⌊A)+0J1×⌊B` |
| `1≤(A-⌊A)+(B-⌊B)` *and* `(A-⌊A)≥B-⌊B` | `(1+⌊A)+0J1×⌊B` |
| `1≤(A-⌊A)+B-⌊B` *and* `(A-⌊A)<B-⌊B` | `(⌊A)+0J1×1+⌊B` |

Figure 21 illustrates the floor of a complex number. Any number within the rectangle has point `B` as its floor.

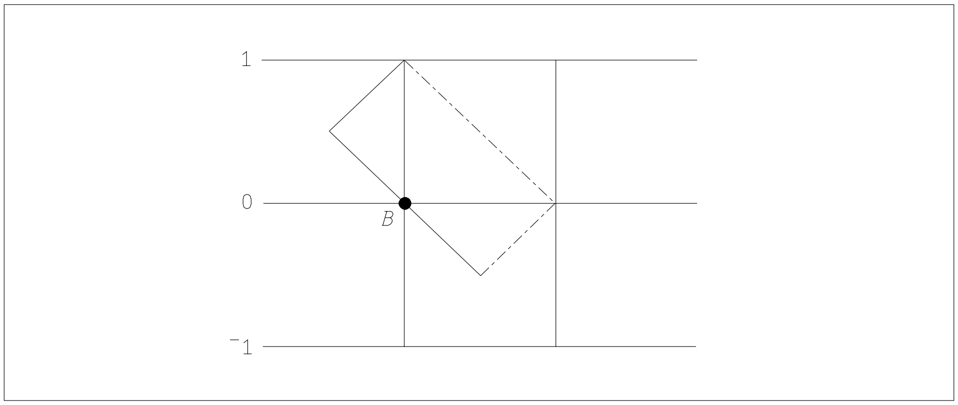


*Figure 21. The Shape of the Complex Floor Area*

The rectangle of sides `2*.5` by `.5*.5` is oriented so that the center of one long side is coincident with a lattice point `B`, and with the ends of the opposite long side coincident with the lattice points above and to the right of `B`. The points within the rectangle all have `B` as floor. The two edges of the rectangle associated with `B` as floor are the bottom one, on which `B` lies, and the one to the left, as shown by the darker lines in the figure.

The examples below show floor applied to nonreal *R*.

```
      ⌊1.5J2.5
2J2
```

```
      ⌊1J2 1.2J2.5 ¯1.2J¯2.5
 1J2 1J2 ¯1J¯3
```

## ⍕ Format (Default)

Z←⍕R Creates a simple character array whose appearance is the same as the display of R (if ⎕PW is set sufficiently wide.)

Z: Character array

Implicit argument: ⎕PP

⍴Z ←→ See below
⍴⍴Z ←→ ,1⌈⍴⍴R if R is simple
⍴⍴Z ←→ ,1; or ,2 if R is nested
(see below)

Z is a character array that includes the character representation of all data in R plus all leading, intermediate, and interdimensional blanks in the display of R.

```
      R←2 3⍴'ONE' 1 1 'TWO' 2 22

      R
 ONE 1  1
 TWO 2 22

      Z←⍕R
      Z
 ONE 1  1
 TWO 2 22

      ⍴Z
2 10
```

To illustrate the format created by ⍕, the display below substitutes carets for blanks in Z.

```
^ONE^1^^1^
^TWO^2^22^
```

For additional information, see “Display of Numbers” on page 12, “Display of Characters” on page 14, and “Display of Arrays” on page 17.

***Printing Width:*** Numeric and character arrays are displayed differently when they are too wide for the printing width, as shown below:

```
      ⎕PW←30
      T←34559898 449449449 13981 93891293
      T
34559898 449449449 13981
      93891293

      U←⍕T
      U
34559898 449449449 13981 93891
      293
```

A numeric array may not be displayed to the allowed printing width because numbers are not usually split for display. For the vector *T*, the number `93891293` is too large for the line so it is displayed on the second line. A character array, such as *U*, is displayed to the allowed printing width.

Because of this difference in the display of numeric and character arrays, the result of `⍕R` may not appear to be the same as the display of *R*.

`⎕PW` and its effect on display are discussed on page 318.

***Simple Character Array Argument:*** If *R* is a simple character array, *Z* and *R* are the same. If *R* is a simple numeric array, `¯1↓ρZ ←→ ¯1↓ρR`.

```
      M←'3  5  SIX'
      M
3  5  SIX

      N←⍕M
      N
3  5  SIX

      N≡M
1
```

```
      S←4×⍳4
      S
4 8 12 16

      Y←⍕S
      Y
4 8 12 16

      S≡Y
0

      ρS
4
      ρY
9
```

***Nested Arrays:*** When *R* is a nested array, *Z* is a vector if all items of *R* at any depth are scalars or vectors. Otherwise, *Z* is a matrix. Two examples are shown.

*Example 1*: All items of *B* are scalars or vectors and the result of `⍕B` is a vector.

```
      B←(2 3 4) 5 (⊂7 8 (9 10 11))
      ρB
3
      ≡B
4
      C←⍕B
      ρC
27
      B
 2 3 4  5   7 8  9 10 11
      C
 2 3 4  5   7 8  9 10 11
```

The display below substitutes carets for the blanks in *C*:

```
^2^3^4^^5^^^7^8^^9^10^11^^^
```

*Example 2*: One item of *D* is a rank-3 array. The result of ⍕*D*, therefore, is a matrix.

```
      D←(1 2) (3 4 5) 6 (2 2 3⍴6+⍳12)
      D
 1 2  3 4 5  6     7  8  9
                  10 11 12

                  13 14 15
                  16 17 18

      E←⍕D
      ⍴E
5 27

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

                  13 14 15
                  16 17 18
```

In the display below, carets are substituted for the blanks in *E*.

```
^1^2^^3^4^5^^6^^^^^7^^8^^9^
^^^^^^^^^^^^^^^^^^10^11^12^
^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^13^14^15^
^^^^^^^^^^^^^^^^^^16^17^18^
```

***Display Rules:*** The display that ⍕ creates follows certain rules. Generally, rows and columns are formatted independently, and rectangular nesting and hierarchy are displayed. Figure 22 on page 138 presents the rules formally.

- There is one column each of leading and trailing blanks.
- Character scalar and vector items in columns containing numeric scalars are right-justified.
- Character scalar and vector items in columns not containing numeric scalars are left-justified.
- Row and column spacing is determined by the context of adjacent items. The spacing increases with the rank of the items. The number of embedded blanks is one less for character items than for other items.

The definition of the default format function is applied recursively so that nested items within a nested array appear with a leading and trailing blank.

The formal rules listed below for default formatting of nested arrays use the function `NOTCHAR`. `NOTCHAR` returns a `1` if `R` is not a simple character array and a `0` otherwise:

```
     ∇ Z←NOTCHAR R
[1]    Z←1
[2]    →(1<≡R)/0
[3]    Z←' '∨.≠,↑0ρ⊂R
     ∇
```

For `Z←⍕R`, where `R` is a nested array:

- `Z` has single left and right blank pad spaces.
- `Z` has `S` intermediate blank spaces between horizontally adjacent items `A` and `B`, where:

  `S←((ρρA)+NOTCHAR A)⌈(ρρB)+NOTCHAR B`
- `Z` has `LN` intermediate blank lines between vertically adjacent items `C` and `D`, where:

  `LN←0⌈¯1+(ρρC)⌈ρρD`
- If the rank of `R` is three or more, `Z` can contain blank lines for the interdimensional spacing.

*Figure 22. Formal Rules for Default Formatting*

***Effect of Printing Precision:*** Because the result of default format has the appearance of the displayed argument, the printing precision (`⎕PP`, page 315), influences the result. For example:

```
      ⎕PP←5
      H←÷3 6
      H
0.33333 0.16667
      I←⍕H
      I
0.33333 0.16667

      ⎕PP←8
      H
0.33333333 0.16666667
      I
0.33333 0.16667
```

# ⍕ Format by Example

`Z←L⍕R` Transforms `R` to a character array that is displayed according to format model `L`. `L` includes control characters, which show where digits can appear in the result, and decorators, such as $ + PAID, which can accompany the display of a number.

`L`: Simple character vector
`R`: Simple real numeric array
`Z`: Simple character array

Implicit argument: `⎕FC[⍳5]`

```
¯1↓⍴Z   ←→  ¯1↓⍴R
  ⍴⍴Z   ←→  ,1⌈⍴⍴R
```

The left argument `L` provides a model for each column of `Z`. It consists of one or more *fields*. A field is a sequence of characters containing at least one digit and bounded by either blanks or a special field boundary mark (the digit 6). The spaces are significant and are retained as column dividers in `Z`. A sequence of characters that does not contain a digit is considered a decoration. `5`'s define the numeric pattern, except where special handling is desired.

In the following example, `@`, `$`, and `EA` are decorators; the dot defines the position of the decimal point; and the blanks define spaces between items. The first field (`55@`) defines the format of the first column and displays the positive numbers `0` through `99`, following each number with the symbol `@`. The second field (`$55.50 EA`) defines the format of the second column and displays positive numbers in dollars-and-cents format, following each amount with `EA`.

```
      L←' 55@  $55.50 EA'
      R←3 2⍴3 4.99 7 7.45 12 .5
      R
 3 4.99
 7 7.45
12 0.5

      Z←L⍕R
      Z
  3@  $ 4.99 EA
  7@  $ 7.45 EA
 12@  $   .50 EA

      ⍴Z
3 15
      ⍴L
15
```

***Conformability:*** For conformability, `L` must have either one field, which is then applied to each column, or as many fields as `R` has columns. Each field of `L` then applies to a corresponding column of `R`.

If `L` has `¯1↑⍴R` fields:

```
(¯1↑⍴Z) ←→ ⍴L
```

If `L` has one field:

```
(¯1↑⍴Z) ←→ (⍴L)×¯1↑⍴R
```

***Specifying the Left Argument:*** `L` can contain two kinds of characters:

1. *Control characters*—the character digits `'1234567890'`, the period (`.`), and the comma (`,`) - that specify:
   - Where numbers in `R` can appear in `Z` and the display pattern for the numbers.
   - Where decimals, controlled commas (thousands indicator), and floating decorators in `L` appear in `Z`. (Note that the display generated by `'.'` and `','` depends on the setting of `⎕FC[1 2]`).
2. Any character, including the space, that is not a control character is a *decorator*. Decorators can be:
   - *Simple*, always appearing in `Z` as they appear in `L`.

     Simple decorators can be used to indicate the meaning of the number being displayed:

     ```
           EXPR←234.67 456.23 987.65 34.23
           'TOTAL ORDER COST:  $5,555.50'⍕+/EXPR
     TOTAL ORDER COST:  $1,712.78
     ```

     It is a good idea to precede each field by one or more spaces to ensure that at least one blank separates numbers.
   - *Controlled*, appearing in `Z` according to the control characters in `L`.
   - *Floating*, appearing next to a number in `Z`, according to the control characters in the pattern for the number.

***Effect of Format Control:*** The format control system variable (`⎕FC`), page 291, is an implicit argument of picture format:

- `⎕FC[1]` specifies the character for the decimal indicator. This character prints wherever a `'.'` is specified in `L`. The default setting is `'.'`.
- `⎕FC[2]` specifies the character used as a decorator to mark thousands. This character prints wherever a `','` is specified in `L`. The default setting is `','`.
- `⎕FC[3]` specifies the fill character whenever `'8'` is specified in `L`. The default is `'*'`.

- ⎕FC[4] can be '0' or an overflow character. For ⎕FC[4] equal to '0', a *DOMAIN ERROR* is generated if *L* specifies a pattern that is too small for the corresponding column of *R*. If ⎕FC[4] is not '0', its value is printed in the field having an overflow. The default is '0'.

```
      ' 55.55'⍕345 .6789
DOMAIN ERROR
      ' 55.55'⍕345 .6789
      ^       ^

      ⎕FC[4]←'?'
      ' 55.55'⍕345 .6789
 ?????   .68
```

- ⎕FC[5] specifies a “print-as-blank” character. It is used in *L* to specify that a blank should separate the digits of a number. The default is '_'.

```
      '555_555_5555'⍕8324632190
832 463 2190
```

  The print-as-blank character is useful to break up a long string of numbers such as a charge card number or to print on a form that has vertical rules.

***Effects of Left Argument:*** The effects of the control characters and decorators are defined and illustrated in Figure 23. In the figure, the control character '5', the decorators ',' and '.', and the control character '0' are presented first because they are the most commonly used. The other digits are presented in numeric order. All examples use the default format control ⎕FC settings.

*Figure 23 (Page 1 of 2). Picture Format Control Characters*

<table>
<tr><th>n</th><th>Effect</th><th>Example</th></tr>
<tr><td>'5'<br>and<br>'.'<br>and<br>','</td><td><p>Perform normal formatting, observing APL2 rules for removing leading and trailing 0's. Display blanks for a value of 0. (See “Display of Numbers” on page 12.)</p><p>Fractional numbers are rounded to the specified number of decimal places.</p><p>Print ⎕FC[1] wherever a '.' appears and ⎕FC[2] wherever a ',' appears.</p><p><b>Note:</b> 5 alone does not allow display of negative values. Use 1 and 2 to control the display of signed numbers.</p></td><td><pre>
      ' 55.55'⍕.10 1.1 1.01 10.019 .11
   .1   1.1   1.01 10.02   .11

      ' 55.55'⍕2 2.2 0 2.22
  2     2.2         2.22
</pre></td></tr>
<tr><td>'0'</td><td>Pad with 0's to the position of the 0. If the value of the corresponding item of <i>R</i> is 0, the position is filled with 0's.</td><td><pre>
      ' 055.50'⍕.3 33.2 0 300
 000.30 033.20 000.00 300.00
</pre></td></tr>
<tr><td>'1'</td><td>Float the decorator against the number only if the value is negative.</td><td><pre>
      ' -55.10'⍕¯3.4 0 4.5 ¯2.12
  -3.40    .00   4.50  -2.12

      ' (55.10)'⍕¯3.4 0 4.5 ¯2.12
  (3.40)    .00     4.50   (2.12)
</pre></td></tr>
</table>

*Figure 23 (Page 2 of 2). Picture Format Control Characters*

| n | Effect | Example |
|---|---|---|
| `'2'` | Float the decorator against the number only if the value is positive. | `      ' +552.50'⍕¯4 40 ¯400`<br>`    4.00  +40.00  400.00`<br><br>`      ' -551.20CR'⍕¯4 40 ¯400`<br>`   -4.00     40.00CR -400.00` |
| `'3'` | Float the decorator against the number. `1` or `2` must also be used if a number may be negative.<br>**Note:** If only one of the characters `1`, `2`, or `3` appears within a given pattern in *L*, it applies to both right and left floating decorators. If more than one appears, each applies to its respective side. | `     ' $555.50'⍕3.1 32.23 324`<br>`$   3.10 $ 32.23 $324.00`<br><br>`     ' $553.50'⍕3.1 32.23 324`<br>`  $3.10  $32.23 $324.00` |
| `'4'` | Counteract the effect of a `1`, `2`, or `3`, to prevent it from affecting the other side of the decimal. Any decorator on the same side of the decimal as the `4` displays as entered. | `     ' -551.20CR'⍕¯1 10 ¯100`<br>`  -1.00     10.00CR -100.00`<br><br>`     ' -551.40CR'⍕¯1 10 ¯100`<br>`  -1.00CR   10.00CR -100.00CR` |
| `'6'` | The decorator to the right marks the end of this field; treat it as though there were a blank between the fields, but display the decorator. | `      '0006/06/06 06:06'⍕5↑⎕TS`<br>`1991/12/17 12:35` |
| `'7'` | The next nonnumeric character to the right is the symbol to be used for scaled form (*E-format*). | `     '  ¯1.7000E¯01'⍕¯25.784 .0034`<br>` ¯2.5784E 01   3.4000E¯03` |
| `'8'` | Fill empty portions of the field with the character defined by `⎕FC[3]`. The default character is `*`. This specification is sometimes called *check protection* because it can be used to print fill characters on checks. | `     '  85555.50'⍕17.3 56.43`<br>` ***17.30  ***56.43`<br><br>`     ⎕FC[3]←'∘'`<br>`     '  85555.50'⍕17.3 56.43`<br>`-∘∘∘17.30  ∘∘∘56.43`<br><br>`     ' -85555.10'⍕¯17.3 56.43`<br>`-∘∘∘17.30  ∘∘∘56.43` |
| `'9'` | Pad with `0`'s to the position of the `9`. If the value of the corresponding item of *R* is `0`, the position is all blanks. | `     '  9995.59'⍕14.7 0 56.43`<br>` 0014.70          0056.43`<br><br>`     '  9995.19-'⍕¯17.3 0 56.43`<br>` 0017.30-          0056.43` |

# ⍕ Format by Specification

`Z←L⍕R` Transforms *R* to a character array that displays according to column specifications *L*. Each pair of *L* corresponds to a column. The first of the pair sets column width; the second sets display precision and format – either conventional or scaled.

A single pair of integers extends the specification to all columns. A single integer is interpreted as `(0,L)`.

*L*: Simple integer vector
*R*: Array of depth 2 or less, whose items are simple real scalars or simple character scalars or vectors
*Z*: Simple character array

Implicit argument: `⎕FC[1 4 6]`

```
¯1↓⍴Z   ←→  ¯1↓⍴R
  ⍴⍴Z   ←→  1⌈⍴⍴R
```

*L* controls the column width (first integer of pair) and the precision and format of the display of numbers in a column (second integer of pair). For example:

```
      R←3 2⍴1 .468987 2 57.276 3 27963
      R
1     0.468987
2    57.276
3 27963

      4 2 12 ¯5⍕R
1.00   4.6899E¯1
2.00   5.7276E1
3.00   2.7963E4

      4 0 10 2⍕R
   1       .47
   2     57.28
   3  27963.00
```

***Specifying the Integer Pair:*** The first integer of a pair specifies the width in *Z* of the corresponding column of *R*.

The first integer can be either:

- `0` to specify that column width should be determined automatically by the number of positions in the largest item in the corresponding column of *R*, allowing a one-column space leading each column.

- Positive to specify overall column width. It must be large enough to include:
  - The sign (if necessary)
  - The digits
  - The decimal indicator
  - The number of positions specified for precision

  **Note:** If you want *Z* to be displayed at the left margin, use `0` to get the minimum readable format.

The second of the pair of integers specifies the precision and format of the display of the numeric simple scalar items in *R*. It can be:

- Positive to specify the number of digits to be displayed after the decimal in the corresponding column of *R*. Decimal positions not filled by digits of *R* are padded with `0`'s.

  If a number has more decimal positions than specified, the number is rounded to the specified number of decimal positions.

- Zero to indicate integer formatting. No decimal point is used.

  If a number of *R* is fractional, it is rounded to an integer.

- Negative to specify scaled form and the number of digits to be displayed in the mantissa in the corresponding column of *R*.

A number that is displayed in scaled form can be displayed by ⍕ in conventional form by appropriate specification of *L*. For example:

```
      2*70              |         22 0⍕2*70
1.180591621E21          |   1180591620717411303424
```

The character representation is an exact reflection of the numeric value to the requested number of digits. In an implementation, not all numbers are represented exactly.

***Effect of*** `⎕FC[1]`: `⎕FC[1]` specifies the decimal position indicator to be used. The default is the point (`.`).

***Effect of*** `⎕FC[4]`: `⎕FC[4]` specifies an overflow character to be used if the number being formatted exceeds the column width set by *L*.

```
      ⎕FC[4]←'?'
      10 0⍕2*70
??????????
```

**Note:** The default for `⎕FC[4]` is `'0'`, which causes a *DOMAIN ERROR* to be generated in overflow cases.

***Effect of*** `⎕FC[6]`: `⎕FC[6]` specifies the negative number indicator to be used. The default is `'¯'`.

***Conformability:*** *L* can have one of the following forms:

1. Pair of integers for each column of *R*, that is, `(⍴L) ←→ 2×¯1↑⍴R` (as shown in previous examples).
2. Single pair of integers, applying to all columns of *R*:

```
      S←3 2⍴⍳6
      7 2⍕S
   1.00   2.00
   3.00   4.00
   5.00   6.00
```

3. Single integer, interpreted as the single pair (`0,L`) and applying to all columns of *R*:

```
      3⍕S
 1.000 2.000
 3.000 4.000
 5.000 6.000
```

***Alignment of Data:*** All columns are right-justified and numbers are aligned on the decimal point. If a column of *R* contains character data only, the corresponding column in *Z* is left-justified.

```
      A←4 2⍴'AMT' 'PERCENT' 5 26.31 6 31.5 8 42.11
      A
AMT  PERCENT
  5    26.31
  6    31.5
  8    42.11

      3 0 9 2⍕A
AMT  PERCENT
  5    26.31
  6    31.50
  8    42.11

      0⍕A
AMT PERCENT
  5      26
  6      32
  8      42

      D←'ITEM' 'PENS' 'BOOKS' 'PAPER',A
      D
 ITEM  AMT  PERCENT
 PENS    5    26.31
 BOOKS   6    31.5
 PAPER   8    42.11
```

```
      5 0 5 0 9 2⍕D
ITEM   AMT  PERCENT
PENS     5    26.31
BOOKS    6    31.50
PAPER    8    42.11
```

***Nested Arrays:*** With format by specification, each item of *R* must be a simple numeric scalar or simple character scalar or vector. Thus, *R* may have a depth no greater than 2. The precision setting applies only to simple numeric scalars of *R*.

```
      2⍕1(2 3)
DOMAIN ERROR
      2⍕1(2 3)
      ^^
```

Use the each operator (¨) to extend precision and format display to vector items.

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

      2⍕¨B
  1.00 2.00  3.00 4.00 5.00
  6.00       7.00
  8.00 9.00  10.00
```

## ⍒ Grade Down

`Z←⍒R` Yields a vector of integers (a permutation of `⍳1↑⍴R`) that puts the subarrays along the first axis of `R` in descending order.

`R`: Simple nonscalar numeric array
`Z`: Simple vector, nonnegative integers

Implicit argument: `⎕IO`

```
 ⍴Z   ←→  1↑⍴R
⍴⍴Z   ←→  ,1
```

```
      ⎕IO←1

      ⍒23 11 13 31 12
4 1 3 5 2
```

```
      ⎕IO←0

      ⍒23 11 13 31 12
3 0 2 4 1
```

***To Sort the Array:*** `R` is sorted in descending order if it is indexed by the result of grade down: `R[⍒R]`.

```
      ⎕IO←1
      A←23 11 13 31 12
      A[⍒A]
31 23 13 12 11
```

***Identical Subarrays:*** The indexes of any set of identical subarrays in `R` occur in `Z` in ascending order of their occurrence in `R`. In other words, their order in relation to one another is unchanged.

```
      ⍒23 14 23 12 14
1 3 2 5 4
```

***Rank of Right Argument Is Two or More:*** If *R* is not a vector, the subarrays are ordered with the first position being the high-order position.

```
      B←5 3⍴ 4 16 37 2 9 26 5 11 63 3 18 45 5 11 54
      B
4 16 37
2  9 26
5 11 63
3 18 45
5 11 54

      ⍒B
3 5 1 4 2

      B[⍒B;]
5 11 63
5 11 54
4 16 37
3 18 45
2  9 26

      C←4 23 54 28 2 11 51 26
      C←C,4 29 17 43 3 19 32 41
      C←3 2 4⍴C,4 23 54 28 1 25 31 16
      C
4 23 54 28
2 11 51 26

4 29 17 43
3 19 32 41

4 23 54 28
1 25 31 16

      ⍒C
2 1 3

      C[⍒C;;]
4 29 17 43
3 19 32 41

4 23 54 28
2 11 51 26

4 23 54 28
1 25 31 16
```

# ⍒ Grade Down (with Collating Sequence)

`Z←L⍒R` Yields a vector of integers (a permutation of `⍳1↑⍴R`) that puts the subarrays along the first axis of `R` in descending order according to the collating sequence `L`.

`L`: Simple nonempty nonscalar character array
`R`: Simple nonscalar character array
`Z`: Simple vector nonnegative integers

Implicit argument: `⎕IO`

```
 ⍴Z  ←→ 1↑⍴R
⍴⍴Z  ←→ ,1
```

Collation works by searching in `L` (in row-major order) for each item in `R` and then attaching a significance to each according to where the item was first found.

```
      ⎕IO←1
      'ABCDE'⍒'BEAD'
2 4 1 3
```

```
      ⎕IO←0
      'ABCDE'⍒'BEAD'
1 3 0 2
```

The significance depends on both the location and the rank of `L`. The last axis of `L` is the most significant for collating, and the first axis of `L` is the least significant.

```
      ⎕IO←1
      A←5 4⍴'DEADBADECEDEBEADDEED'
      A
DEAD
BADE
CEDE
BEAD
DEED

      'ABCDE'⍒A
5 1 3 4 2

      C←'FACE$'
      B←'@$&ABCDEF'
      B⍒C
1 4 3 2 5
      C[B⍒C]
FECA$
```

In the following example, differences in spelling have higher significance than differences in case, and lowercase letters have more significance than their uppercase counterparts.

```
      K←5 4⍴'dealDealdeadDeadDEED'
      K
deal
Deal
dead
Dead
DEED

      H←2 12⍴'abcdefghijklABCDEFGHIJKL'
      H
abcdefghijkl
ABCDEFGHIJKL

      Z←H⍒K
      K[Z;]
DEED
Deal
deal
Dead
dead
```

A collating sequence is provided as the variable *DCS* in the *EXAMPLES* workspace distributed with APL2.

*DCS* is discussed on page 156, and shown in Figure 24 on page 157.

```
      DCS⍒'AVENUE'
2 5 4 3 6 1

      H←'YZOMMXA'
      DCS⍒H
2 1 6 3 4 5 7

      H[DCS⍒H]
ZYXOMMA

      Q←5 4⍴'SENT ZAPDOWNALSOBOA'
      Q
SENT
 ZAP
DOWN
ALSO
BOA

      DCS⍒Q
1 3 5 4 2
```

```
      Q[DCS⍒Q;]
SENT
DOWN
BOA
ALSO
 ZAP

      K
deal
Deal
dead
Dead
DEED

      DCS⍒K
5 1 2 3 4

      K[DCS⍒K;]
DEED
deal
Deal
dead
Dead

      S←⊃'X1' 'X10' 'X2' 'X21' 'X3' 'X9' 'X11' 'x3'
      S
X1
X10
X2
X21
X3
X9
X11
x3

      DCS⍒S
4 7 2 8 6 5 3 1

      S[DCS⍒S;]
X21
X11
X10
x3
X9
X3
X2
X1
```

***Identical Subarrays:*** The indexes of any set of identical subarrays in *R* occur in *Z* in ascending order (according to collating sequence *L*) of their occurrence in *R*. In other words, their order in relation to one another is unchanged.

```
      'ABCDE'⍒'DABBED'
5 1 6 3 4 2
```

***Items Not in Collating Sequence:*** Items of *R* not found in *L* have collating sequence as if they were found immediately past the end of *L*. They are assigned indexes in ascending order of their occurrence in *R*.

```
      Q←'BLEAT'
      W←'ABCDE'⍒Q
      W
2 5 3 1 4
      Q[W]
LTEBA
```

## ⍋ Grade Up

`Z←⍋R` Yields a vector of integers (a permutation of `⍳1↑⍴R`) that puts the subarrays along the first axis of *R* in ascending order.

*R*: Simple nonscalar numeric array
*Z*: Simple vector nonnegative integers

Implicit argument: `⎕IO`

```
 ⍴Z   ←→  1↑⍴R
⍴⍴Z   ←→  ,1
```

```
      ⎕IO←1
      ⍋23 11 13 31 12
2 5 3 1 4
```

```
      ⎕IO←0
      ⍋23 11 13 31 12
1 4 2 0 3
```

***To Sort Right Argument:*** *R* is sorted in ascending order if it is indexed by the result of grade up: `R[⍋R]`.

```
      ⎕IO←1
      A←23 11 13 31 12
      A[⍋A]
11 12 13 23 31
```

***Identical Subarrays:*** The indexes of any set of identical subarrays in *R* occur in *Z* in ascending order of their occurrence in *R*. In other words, their order in relation to one another is unchanged.

```
      ⍋23 14 23 12 14
4 2 5 1 3
```

***Rank of R is Two or More***: If R is not a vector, the subarrays are ordered with the first position being the most significant position.

```
      B←5 3⍴4 16 37 2 9 26 5 11 63 3 18 45 5 11 54
      B
4 16 37
2  9 26
5 11 63
3 18 45
5 11 54

      ⍋B
2 4 1 5 3

      B[⍋B;]
2  9 26
3 18 45
4 16 37
5 11 54
5 11 63

      C←4 23 54 28 2 11 51 26
      C←C,4 29 17 43 3 19 32 41
      C←3 2 4⍴C,4 23 54 28 1 25 31 16
      C
4 23 54 28
2 11 51 26

4 29 17 43
3 19 32 41

4 23 54 28
1 25 31 16

      ⍋C
3 1 2

      C[⍋C;;]
4 23 54 28
1 25 31 16

4 23 54 28
2 11 51 26

4 29 17 43
3 19 32 41
```

## ⍋ Grade Up (with Collating Sequence)

*Z*←*L*⍋*R* Yields a vector of integers (a permutation of ⍳1↑⍴*R*) that puts the subarrays along the first axis of *R* in ascending order according to the collating sequence *L*.

*L*: Simple nonempty nonscalar character array
*R*: Simple nonscalar character array
*Z*: Simple vector, nonnegative integers

Implicit argument: ⎕*IO*

```
 ⍴Z   ←→  1↑⍴R
⍴⍴Z   ←→  ,1
```

Collation works by searching in *L* (in row-major order) for each item in *R* and then attaching a significance to each according to where it was first found.

```
      ⎕IO←1
      'ABCDE'⍋'BEAD'
3 1 4 2
```

```
      ⎕IO←0
      'ABCDE'⍋'BEAD'
2 0 3 1
```

The significance depends on both the location and rank of *L*. The last axis of *L* is the most significant for collating, and the first axis of *L* is the least significant.

```
      ⎕IO←1
      A←5 4⍴'DEADBADECEDEBEADDEED'
      A
DEAD
BADE
CEDE
BEAD
DEED

      'ABCDE'⍋A
2 4 3 1 5

      Q←'FACE$'
      S←'@$&ABCDEF'
      S⍋Q
5 2 3 4 1
      Q[S⍋Q]
$ACEF
```

In the following example, differences in spelling have higher significance than differences in case, and lowercase letters have more significance than uppercase letters.

```
      K←5 4⍴'dealDealdeadDeadDEED'
      K
deal
Deal
dead
Dead
DEED

      H←2 12⍴'abcdefghijklABCDEFGHIJKL'
      H
abcdefghijkl
ABCDEFGHIJKL

      Z←H⍋K
      Z
3 4 1 2 5

      K[Z;]
dead
Dead
deal
Deal
DEED
```

A collating sequence is provided as the variable *DCS* in the *EXAMPLES* workspace distributed with APL2.

*DCS*, which is shown in Figure 24 on page 157, sorts an alphanumeric array in the following order:

```
     ' AA̲aBB̲bCC̲cDD̲dEE̲eFF̲fGG̲gHH̲hII̲iJJ̲jKK̲kLL̲lMM̲m
      OO̲oPP̲pQQ̲qRR̲rSS̲sTT̲tUU̲uVV̲vWW̲wXX̲xYY̲yZZ̲z0123456789'
```

As a result of the structure of *DCS*, numeric integer suffixes in rows of a matrix can be sorted in numeric order.

*DCS* has a shape of `10 2 28`. The first column of each row is a blank. Each plane is a matrix of shape 2 28, where all nonprintable characters are blanks.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0
A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲

                          1
abcdefghijklmnopqrstuvwxyz

                          2

                          3

                          4

                          5

                          6

                          7

                          8

                          9
```

*Figure 24. Collating Sequence Array*

```
      DCS⍋'AVENUE'
1 3 6 4 5 2

      H←'LWLOIBY'
      DCS⍋H
6 5 1 3 4 2 7

      H[DCS⍋H]
BILLOWY

      K←5 4⍴'SENT ZAPDOWNALSOBOA '
      K
SENT
 ZAP
DOWN
ALSO
BOA
      DCS⍋K
2 4 5 3 1

      K[DCS⍋K;]
 ZAP
ALSO
BOA
DOWN
SENT
```

```
      K←5 4⍴'dealDealdeadDeadDEED'
      K
deal
Deal
dead
Dead
DEED
      DCS⍋K
4 3 2 1 5

      K[DCS⍋K;]
Dead
dead
Deal
deal
DEED

      S←⊃'X1' 'X10' 'X2' 'X21' 'X3' 'X9' 'X11' 'x3'
      S
X1
X10
X2
X21
X3
X9
X11
x3
      DCS⍋S
1 3 5 6 8 2 7 4

      S[DCS⍋S;]
X1
X2
X3
X9
x3
X10
X11
X21
```

***Identical Subarrays:*** The indexes of any set of identical subarrays in R occur in Z in ascending order (according to collating sequence L) of their occurrence in R. In other words, their order in relation to one another is unchanged.

```
      'ABCDE'⍋'DABBED'
2 3 4 1 6 5
```

***Items Not in Collating Sequence:*** Any items of *R* not found in *L* have collating sequence as if they were found immediately past the end of *L*. They are assigned indexes in ascending order of their occurrence in *R*:

```
      W←'ABCDE'⍋'EXACT'
      W
3 4 1 2 5
      'EXACT'[W]
ACEXT
```

# ⌷ Index

*Z*←*L*⌷*R*   This function selects cross-sections of *R* using a list of index arrays *L*.

**Note:** See the discussion of the )*PBS* command in “)*PBS*—Query or Set the Printable Backspace Character (APL2/370 Only)” on page 444 for alternate ways to enter this character.

*L*: Scalar or vector of nonnegative integers of depth no greater than 2
*R*: Any array
*Z*: An array cross-section of *R*

Implicit Argument: ⎕IO

```
ρZ   ←→ ⊃,/ρ¨L
ρρZ  ←→ ,+/∊ρ¨ρ¨L
```

Index is similar in function to bracket index. For example, to index a 3-dimensional array *A* with page, row, and column index arrays *I*, *J*, and *K*:

```
I J K ⌷ A   ←→   A[I;J;K]
```

The length of the left argument must be equal to the rank of the right argument.

```
ρ,L  ←→  ρρR
```

Index, unlike bracket index, can be used to index a scalar with an empty left argument.

```
(⍳0) ⌷ Scalar  ←→  Scalar
```

***When a Vector Is Indexed:*** If V is a vector, a single-item vector or scalar left argument is required.

```
      ⎕IO←1
      V←2 2.3 ¯5 999 .01
      3⌷V
¯5
      (⊂3 4)⌷V
¯5 999
      (⊂2 3ρ1 2 1 4 1 2)⌷V
  2 2.3 2
999 2   2.3
```

***When a Matrix Is Indexed:*** If M is a matrix, a two-item vector left argument is required.

```
      ⎕IO←1
      M←3 4⍴⍳12
      M
1  2  3  4
5  6  7  8
9 10 11 12
      3 1⌷M
9
      3(1 3)⌷M
9 11
      (2 3)4⌷M
8 12
      (2 3)(,4)⌷M
 8
12
      ⍴(1 2)(3 4⍴3)⌷M
2 3 4
      ⍴(⍳0)(⍳0)⌷M
0 0
```

## ⍳ Index Of

`Z←L⍳R` Yields the first occurrence in `L` of items in `R`.

`L`: Vector
`Z`: Nonnegative integers

Implicit arguments: `⎕IO`, `⎕CT`

```
 ⍴Z  ←→  ⍴R
⍴⍴Z  ←→  ⍴⍴R
```

The following expression is equivalent to index of:

```
L⍳R ←→ ⎕IO++/∧\~R∘.≡L
```

```
      ⎕IO←1
      8 4 2 7 3⍳3 8 4
5 1 2

      'SPORT'⍳'TOP'
5 3 2
```

```
      ⎕IO←0
      8 4 2 7 3⍳3 8 4
4 0 1

      'SPORT'⍳'TOP'
4 2 1
      ⎕IO←1
      A←(2 3) (⍳0) 'ME'
      A⍳'ME' (⍳0)
3 2
```

***Item Not Found:*** If an item of `R` is not found in `L`, the corresponding item in `Z` is `⎕IO+⍴L`.

```
      ⎕IO←1
      8 9 5⍳2 5 8
4 3 1

      L←'OH' 'NO' 'I'
      L⍳'NO' 'ON'
2 4

      'OHNOI'⍳'NO' 'ON'
6 6
```

```
      'WIZARD'⍳'OZ'
7 3


      6 7 4⍳4 7 (⍳0)
3 2 4
```

***Item Recurs:*** If an item of `R` occurs several times in `L`, the corresponding item in `Z` is the index of its first occurrence.

```
      5 5 8 8 9⍳8 9 5
3 5 1
```

```
      'BANANA'⍳'BANANA'
1 2 3 2 3 2
```

## ⌷[ ] Index with Axis

`Z←L⌷[X]R` This function selects cross-sections of *R* using a list of index arrays *L*, which correspond to axes *X*.

**Note:** See the discussion of the `)PBS` command in "`)PBS`—Query or Set the Printable Backspace Character (APL2/370 Only)" on page 444 for alternate ways to enter this character.

*L*: Scalar or vector of nonnegative integers of depth no greater than 2.
*R*: Any array.
*X*: Simple scalar or vector; nonnegative integers: `X∊⍳⍴⍴R`
*Z*: An array cross-section of *R*.

Implicit Argument: `⎕IO`

Index with axis is similar in function to bracket index with elided positions. For example, to index a 3-dimensional array *A* with page and column index arrays *I* and *J* and select all rows:

```
I J ⌷[⎕IO+0 2] A   ←→   A[I;;J]
```

The length of the left argument must be equal to the number of axes mentioned.

```
⍴,L   ←→   ⍴,X
```

Index with axis compared with bracket index:

```
      ⎕IO←1
      A←2 3 4⍴⍳24
      A
 1  2  3  4
 5  6  7  8
 9 10 11 12

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

      2⌷[1]A
13 14 15 16
17 18 19 20
21 22 23 24

      A[2;;]
13 14 15 16
17 18 19 20
21 22 23 24

      (1 3)4⌷[2 3]A
 4 12
16 24

      A[;1 3;4]
 4 12
16 24
```

# . Inner Product (from Array Product)

| | |
|---|---|
| `Z←L LO.RO R` | Combines the subarrays along the last axis of `L` with subarrays along the first axis of `R` by applying an `RO` outer product. An `LO`-reduction is then applied to each item of that result. |

`LO`: Dyadic function
`RO`: Dyadic function

```
 ρZ   ←→  (¯1↓ρL),1↓ρR
ρρZ   ←→  ,0⌈¯2+(ρρL)+ρρR
```

Formally, for nonscalar arguments, inner product is defined in origin `1` as:

```
LO/¨ (⊂[ρρL]L)∘.RO ⊂[1]R
```

For a scalar argument, the enclose with axis (`⊂[ ]`) in the above expression is replaced by enclose.

The primary definition of inner product is in terms of matrix arguments. For matrixes `L` and `R` and result `Z`:

```
Z[I;J] ←→ ⊂LO/L[I;] RO R[;J]
```

Figure 25, for example, depicts the calculation of a `+.×` inner product.

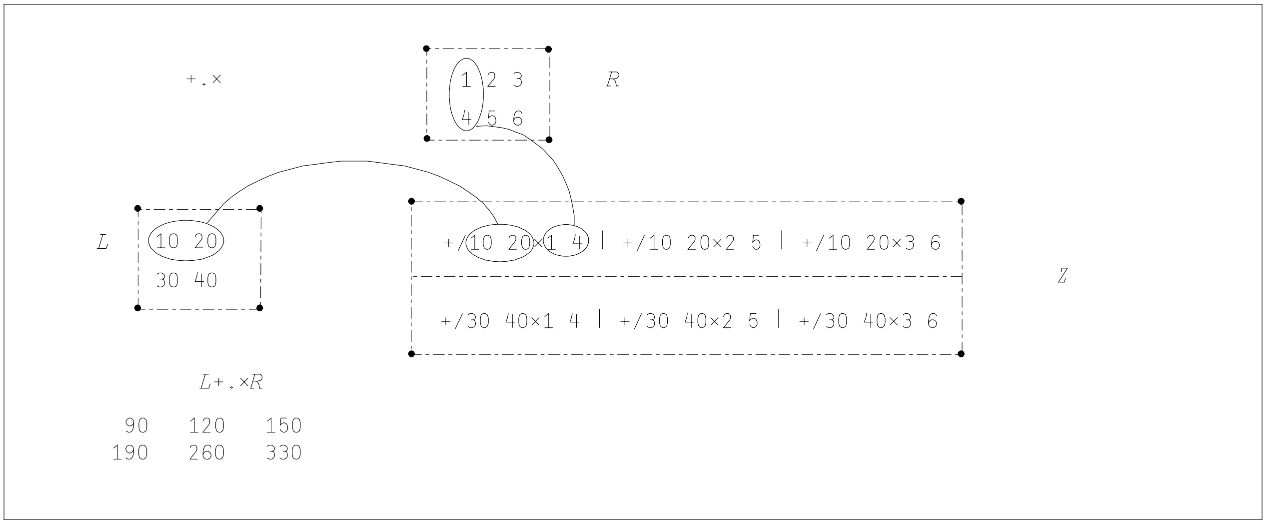


*Figure 25. Calculation of an Inner Product*

The `+.×` inner product is the same function as the matrix product used in matrix algebra.

Informally, for matrix arguments, inner product is defined in terms of reduction and outer product as:

*LO*/¨(rows of *L*) ∘.*RO* (columns of *R* .)

```
      M←4 4⍴1 1 1 1 0 1 1 1 0 0 1 1 0 0 0 1
      M
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1

      M∧.=M
0 0 0 1
0 0 0 0
0 0 0 0
0 0 0 0

      K←3 8⍴'SATURDAY7/04/99 JULY 4  '
      K
SATURDAY
7/04/99
JULY 4

      K+.∊'0123456789'
0 5 1

      (2 3⍴⍳6)⌊.⌈3 4⍴⍳12
1 2 3 4
4 4 4 4
      J←3 2⍴⍳6
      J
1 2
3 4
5 6

      P←2 2⍴(.1×J)
      P
0.1 0.2
0.3 0.4

      J,.+P
  1.1 2.3   1.2 2.4
  3.1 4.3   3.2 4.4
  5.1 6.3   5.2 6.4
```

```
      S←3 5⍴'SANDYBETTYGRACE'
      S
SANDY
BETTY
GRACE
      S∧.='SANDY'
1 0 0
      'SANDY'∧.=⍉S
1 0 0
```

***Empty Argument(s):*** If an empty argument is presented to the outer product portion of the inner product calculation, the related fill function is applied as discussed in "∘. Outer Product (from Array Product)" on page 186. If an empty argument is presented to the reduction portion of the inner product calculation, the related identity function is applied as discussed in "Reduce (from Slash)", on page 209.

```
      U←(0 2⍴0)+.×2 0⍴0
      ⍴U
0 0
```

```
      Q←(2 0⍴0)+.×0 4⍴5
      Q
0 0 0 0
0 0 0 0
      ⍴Q
2 4
```

***Derived Functions of Special Interest:*** The following functions derived from the inner product operator have wide application:

- Matrix product (+.×)
- Count (+.∊)
- Outer product of vectors (requires simple array arguments) (,.*RO*)
- Match two lists (∧.=)

## ⍳ Interval

`Z←⍳R` Produces `R` consecutive ascending integers, beginning with `⎕IO`.

`R`: Simple scalar or one-item vector, nonnegative integer
`Z`: Simple vector, nonnegative integers

Implicit argument: `⎕IO`

```
⍴Z   ←→  ,R
⍴⍴Z   ←→  ,1
```

```
      ⎕IO←1
      ⍳6
1 2 3 4 5 6
```

```
      ⎕IO←0
      ⍳6
0 1 2 3 4 5
```

***Zero Argument:*** The expression `⍳0` produces an empty vector and is a common method of creating or indicating an empty vector.

```
      Z←⍳0
      Z
(empty)
      ⍴Z
0
```

***Arithmetic Progressions:*** Interval is used to create arithmetic progressions.

```
      ⎕IO←0
      ⍳5
0 1 2 3 4
      10+⍳5
10 11 12 13 14
      .1×10+⍳5
1 1.1 1.2 1.3 1.4
```

## ,[] Laminate

`Z←L,[X]R`  Joins `L` and `R` by forming a new axis of length 2, which is filled with `L` and `R`.

`Z`: Nonscalar
`X`: Simple scalar fraction between `¯1+⎕IO` and `⎕IO+(⍴⍴L)⌈⍴⍴R`

Implicit argument: `⎕IO`

```
 ⍴Z   ←→ Case dependent; see below.
⍴⍴Z   ←→  1+(⍴⍴L)⌈⍴⍴R
```

`X` defines the position of the new axis: between two existing axes, before the first or after the last, as follows:

`X<⎕IO`—creates a new first axis

`X>⎕IO+(⍴⍴L)⌈⍴⍴R`—creates a new last axis

For other `X`—creates a new axis between the `⌊X`th and the `⌈X`th axes.

If both arguments are scalars, `L,[X]R ←→ L,R` where `X<⎕IO` and `X>⎕IO-1`.

```
      A←'FOR'
      B←'AXE'

      Z←A,[.5]B
      Z
FOR
AXE
      ⍴Z
2 3
      ≡Z
1


      W←A,[1.1]B
      W
FA
OX
RE
      ⍴W
3 2
      ≡W
1
```

```
      H←(1 2) (3 4)
      K←'AB' 'CD'
      Y←H,[.5]K
      Y
  1 2   3 4
  AB    CD
      ⍴Y
2 2
      ≡Y
2


      V←H,[1.1]K
      V
  1 2   AB
  3 4   CD
      ⍴V
2 2
      ≡V
2
```

***Conformability:*** The arguments of laminate must have the same shape and rank or one must be a scalar.

If one argument is a scalar, it is reshaped to match the nonscalar argument. After scalar extension, the shape of the result is:

```
ρZ ←→ (2,ρL)[⍋X,⍳ρρL]
```

```
      Q←3 3ρ'STYHIMRED'
      Q
STY
HIM
RED
      'A',[.1]Q
AAA
AAA
AAA

STY
HIM
RED
```

```
      'A',[1.1]Q
AAA
STY

AAA
HIM

AAA
RED
      'A',[2.1]Q
AS
AT
AY

AH
AI
AM

AR
AE
AD
```

# ⍟ Logarithm

`Z←L⍟R` Determines the base `L` logarithm of `R`.

`L` and `R`: Numeric, nonzero
`Z`: Numeric

*Scalar Function*

Logarithm is defined in terms of the natural logarithm:

`L⍟R ←→ (⍟R)÷⍟L`

Because `⍟1` is `0`, this definition implies that if `L` is `1`, `R` must also be `1`.

```
      2⍟256
8
      10⍟100 500 1000
2 2.698970004 3
```

```
      2⍟0J2
1J2.266180071
      1⍟1
1
```

# | Magnitude

`Z←|R` Yields the distance between `0` and `R`.

`R`: Numeric
`Z`: Numeric, real

*Scalar Function*

For real `R`, `|R ←→ R⌈-R`.

For complex `R` of the form `A+0J1×B` (where `A` and `B` are real):
`|R ←→ (+/A B*2)*.5`

For all `R`, `|R ←→ (R×+R)*.5`.

```
      |¯4.2
4.2
      |2 ¯2 .3 ¯.3
2 2 0.3 0.3
```

```
      |2J¯3
3.605551275
      |0J1 2J¯2 4J3
1 2.828427125 5
```

## ≡ Match

*Z*←*L*≡*R* Yields a 1 if the arguments are the same in structure and data, and a 0 otherwise.

**Note:** See “)*PBS*—Query or Set the Printable Backspace Character (APL2/370 Only)” on page 444 for alternate ways to enter this character.

*Z*: Boolean

Implicit argument: ⎕*CT*

```
 ⍴Z   ←→ ⍳0
⍴⍴Z   ←→ ,0
```

```
      'TO' 'ME'≡'TO ME'
0
      1 2 3 4≡1 2 3 4
1
```

```
      'TO' 'ME'≡'TO' 'ME'
1
      1 2 3 4=1 2 3 4
1 1 1 1
```

***Empty Arrays:*** Empty arrays are the same if they have the same structure and prototype.

```
      ''≡⍳0
0

      (0 2⍴0)≡(0 2⍴' ')
0
```

```
      ''≡''
1

      (0 2⍴0)≡(0 2⍴0)
1
```

## ⌹ Matrix Divide

`Z←L⌹R` Yields the solution of a system of linear equations or other algebraic or geometric results, according to the values and shapes of `L` and `R`.

`L` and `R`: Simple numeric array of rank 2 or less
`Z`: Simple numeric

```
 ⍴Z   ↔  (1↓⍴R),1↓⍴L
⍴⍴Z   ↔  ,1⌈¯2+(⍴⍴L)+⍴⍴R
```

***Conformability:*** The definition of matrix divide assumes that `L` and `R` are matrixes. If either `L` or `R` is a vector, it is treated as a one-column matrix. If either `L` or `R` is a scalar, it is treated as a matrix of shape `1 1`.

After these extensions, `L` and `R` must have the same number of nonzero rows. `L⌹R` is executed only if all the following are true:

- `L` and `R` have the same number of rows
- The columns of `R` are linearly independent
- `R` does not have more columns than rows.

If `Z←L⌹R` is executable, `Z` is determined to minimize the value of the least squares expression:

```
+/,(L-R+.×Z)*2
```

Various interpretations of the results for different arguments are discussed below.

***Solving Systems of Linear Equations:*** If `R` is a nonsingular matrix and `L` is a vector, `Z` is the solution of the system of linear equations expressed conventionally as *Ax=b*, where *A*(`R`) represents the coefficients of variables in a system of linear equations in several variables, and *b*(`L`) is a constant, and *x*(`Z`) is the unknown.

If `L` is a matrix, `Z` is the solution of the system of linear equations for each *column* of `L`. For either a vector or matrix `L`:

`L ←→ R+.×Z`

```
      R←2 3⍴1 0 0 2
      R
1 0
0 2
      L←2 2⍴1 2 4 8
      L
1 2
4 8
      1 4⌹R
1 2

      L⌹R
1 2
2 4
```

```
      R←2 2⍴ 0J1 0 0 2
      R
0J1 0
0   2

      1 4⌹R
0J¯1 2

      L⌹R
0J¯1 0J¯2
2    4
```

Geometrically, if `R` is a matrix and `L` is a vector, `R+.×L⌹R` is a point closest to the point `L` in the space spanned by the column vectors of `R`. That is, `R+.×L⌹R` is the projection of `L` on the space spanned by the columns of `R`.

***Curve Fitting:*** The least squares approximation to a numeric function `F` can be determined as follows. If `X` is a vector and `Y←F X` is executed, `P←⌽Y⌹X∘.*0,⍳D` is the vector of the coefficients of the polynomial of degree `D` (constant term last) which best fits the function `F` at points `R`.

For example, the sequence in Figure 26 computes and evaluates successively close polynomial approximations to the gamma function where `X⊥P` evaluates polynomial `P` at point `X`.

```
      ⎕PP←8
      V←1 1.2 1.4 1.6 1.8 2
      L←!V
      L
1 1.1018025 1.2421693 1.4296246 1.6764908 2
      1.6⊥⌽L⌹V∘.*0,⍳2
1.434011
      1.6⊥⌽L⌹V∘.*0,⍳3
1.4289585
      1.6⊥⌽L⌹V∘.*0,⍳4
1.4295805
      1.6⊥⌽L⌹V∘.*0,⍳5
1.4296246
```

*Figure 26. Polynomial Approximations of the Gamma Function*

***Compared to Matrix Inverse:*** For all nonsingular matrixes, `I⌹R` ←→ `⌹R`, where `I` is the `(⍴R)` identity matrix.

***Algorithm for Matrix Divide:*** Matrix divide (as well as matrix inverse) uses the Lawson and Hanson Algorithm[1], which is an extension of the Golub and Businger Algorithm[2], to handle undetermined cases.

---

[1] C.L. Lawson and R.J. Hanson, *Solving Least Squares Problems* (New Jersey: Prentice-Hall, 1974).

[2] G.H. Golub and P. Businger, “Linearly Least Squares Solutions by Householder Transformations” *Numerische Mathematik*, Vol. 7, (1965): pp. 269-276.

# ⌹ Matrix Inverse

`Z←⌹R` Yields the inverse of a nonsingular matrix. Results for other matrixes, vectors, and scalar `R` are discussed below.

`R` and `Z`: Simple numeric array of rank 2 or less

```
 ρZ   ←→  ⌽ρR
ρρZ   ←→  ρρR
```

The result of `⌹R` depends on the nature of `R` as follows:

| **If** `R` **is** | **Then** `⌹R` **is** |
|---|---|
| Nonsingular matrix | Inverse of `R` |
| Matrix such that `R` has more rows than columns | Pseudo-inverse of `R`, in the least squares sense |

`R` cannot have more columns than rows.

***Nonsingular Matrix:*** If `R` is a nonsingular matrix, `Z` is the matrix inverse of `R` and:

```
I ←→ R+.×⌹R
```

where `I` is a `ρR` identity matrix:

```
(I ←→ (⍳↑ρR)∘.=⍳↑ρR)
```

**Note:** Rounded off and poorly conditioned arguments can cause inaccurate results.

```
      R←3 3ρ1 0 0 0 2 0 2 0 4
      R
1 0 0
0 2 0
2 0 4

      Z←⌹R
      Z
 1    0    0
 0    0.5  0
¯0.5  0    0.25

      Z+.×R
1 0 0
0 1 0
0 0 1
```

```
      ⎕PP←4

      R←3 3⍴1 2 3 2 4 5 3 5 6
      R
1 2 3
2 4 5
3 5 6

      Z←⌹R
      Z
 1 ¯3  2.000E0
¯3  3 ¯1.000E0
 2 ¯1 ¯1.604E¯16

      R+.×Z
 1.000E0   2.22E¯16  0.00E0
¯8.882E¯16 1.00E0    2.22E¯16
¯1.554E¯15 2.224E¯16 1.00E0
```

Numbers that are smaller than `1E¯15` in the result array can be considered as approximating 0.

***Matrix with More Rows Than Columns:*** If `R` is a matrix with more rows than columns (`>/⍴R`), `Z` is a *pseudo-inverse* of `R` that minimizes the expression:

`+/(,I-R+.×Z)*2`

where `I` is the `(2⍴1↑⍴R)` identity matrix.

The matrix `Z` is a left inverse of `R`; that is, `Z+.×R` produces a `(2⍴¯1↑⍴R)` identity matrix.

```
      R←4 3⍴1 0 0 0 2 0 2 0 2 0 1 4
      R
1 0 0
0 2 0
2 0 2
0 1 4

      ⎕PP←5
      Z←⌹R
      Z
 0.24706   0.094118  0.37647  ¯0.18824
 0.047059  0.49412  ¯0.023529  0.011765
¯0.058824 ¯0.11765   0.029412  0.23529

      Z+.×R
 1.0000E0   ¯5.5511E¯17 8.3267E¯17
¯7.8063E¯18  1.0000E0   0.0000E0
 5.2042E¯18 ¯1.3878E¯17 1.0000E0
```

***Vector:*** If *R* is a vector, *Z* is its image obtained by inversion in the unit circle (or sphere).

```
      ⌹3 4
0.12 0.16
```

***Scalar:*** If *R* is a scalar, *Z* is `÷`*R*.

```
      ⌹3
0.3333333333
```

***Compared to Matrix Divide:*** For all nonsingular matrixes, `⌹`*R* `←→` *I*`⌹`*R*, where *I* is the `(⍴`*R*`)` identity matrix.

***Algorithm for Matrix Inverse:*** Matrix inverse (as well as matrix divide) uses the Lawson and Hanson Algorithm[3], which is an extension of the Golub and Businger Algorithm[4], to handle undetermined cases.

---

[3] C.L. Lawson and R.J. Hanson, *Solving Least Squares Problems* (New Jersey: Prentice-Hall, 1974).

[4] G.H. Golub and P. Businger, “Linearly Least Squares Solutions by Householder Transformations” *Numerische Mathematik*, Vol. 7, (1965)**:** pp. 269-276.

# ⌈ Maximum

`Z←L⌈R`   Returns the larger of `L` and `R`.

`L`, `R`, and `Z`: Numeric, real

*Scalar Function*

```
      3⌈4
4
      5⌈4 5 7
5 5 7
```

```
      ¯2⌈¯3
¯2
      3.3 0 ¯6.7⌈3.1 ¯4 ¯5
3.3 0 ¯5
```

# ∊ Member

`Z←L∊R` Yields a Boolean array `Z` with the same shape as `L`. An item of `Z` is 1 if the corresponding item of `L` can be found anywhere in `R`. An item of `Z` is 0 otherwise.

`Z`: Simple Boolean array

Implicit argument: `⎕CT`

```
 ⍴Z   ←→ ⍴L
⍴⍴Z   ←→ ⍴⍴L
```

The Boolean array `Z` maps to `L`, following this identity:

```
Z ←→ ∨/L∘.≡,R
```

```
      'BANANA'∊'AN'
0 1 1 1 1 1

      5 1 2∊6 5 4 1 9
1 1 0
```

```
      A←2 3⍴8 3 5 8 4 8
      A
8 3 5
8 4 8
      A∊1 8 9 3
1 1 0
1 0 1
```

***Nested Arrays:*** An item of `L` is found in `R` only if an item in `R` matches that item exactly in structure and data (within comparison tolerance):

```
      B←'AH' 'HA' 'AH' 'NO'
```

```
      B
 AH HA AH NO
      ⍴B
4
      ≡B
2
      B∊'AH'
0 0 0 0
      B∊⊂'AH'
1 0 1 0
```

```
      C←(1 2) (⍳0) (3 4)
      C
 1 2    3 4
      ⍴C
3
      ≡C
2
      C∊(1 2) (3 5) (⍳0)
1 1 0
```

***Empty Right Argument:*** If `R` is empty, `Z` is `(⍴L)⍴0`.

```
      8 9 7 3∊⍳0
0 0 0 0
```

***Mathematical Membership:*** The expression `(⊂L)∊R` determines whether `L` as a unit is contained in `R`.

# ⌊ Minimum

*Z*←*L*⌊*R*    Returns the smaller of *L* and *R*.

*L*, *R* and *Z*: Numeric, real

*Scalar Function*

```
      3⌊4
3
      5⌊4 5 7
4 5 5
```

```
      ¯2⌊¯3
¯3
      3.3 0 ¯6.7⌊3.1 ¯4 ¯5
3.1 ¯4 ¯6.7
```

# × Multiply

`Z←L×R` Multiplies `L` by `R`.

`L`, `R`, and `Z`: Numeric

*Scalar Function*

Multiply is the arithmetic multiplication function.

```
      3×4
12
      3×0 ¯2 5 .7
0 ¯6 15 2.1
```

```
      1J2×3J4
¯5J10
      1 ¯3 .8×1 .5 ¯.2
1 ¯1.5 ¯0.16
```

# ⍟ Natural Logarithm

`Z←⍟R` Determines the logarithm of `R` to the base of the natural logarithms *e*, where *e* is approximately 2.7182818284590452.

`R`: Numeric, nonzero
`Z`: Numeric

*Scalar Function*

```
       ⍟1
0
       ⍟¯1
0J3.141592654
```

```
       ⍟2.7182818284
1
       ⍟0J1
0J1.570796327
```

# - Negative

`Z←-R`    Reverses the sign of `R`.

`R` and `Z`: Numeric

*Scalar Function*

If `R` is positive, `Z` is negative. If `R` is negative, `Z` is positive. If `R` is 0, `Z` is 0. For complex numbers, the signs of both the real and imaginary parts are changed.

Subtract and negative are related as follows:

`-R ←→ 0-R`

```
      -5
¯5
      -3 ¯1 .6 7
¯3 1 ¯0.6 ¯7
```

```
      -2J4
¯2J¯4
      -0J1 ¯3J4 ¯2J¯1
0J¯1 3J¯4 2J1
```

## ∘. Outer Product (from Array Product)

`Z←L ∘.RO R` Applies the function `RO` between pairs of items, one from `L` and one from `R`, in all combinations.

`RO`: Dyadic function
`LO`: The left operand must be the jot symbol (∘)

```
 ρZ   ←→ (ρL),ρR
ρρZ   ←→ (ρρL)+ρρR
```

For any scalar `I` and `J` for which `I⊃L` and `J⊃R` is defined:

```
(I,¨J)⊃Z ←→ (I⊃L) RO J⊃R
```

This identity defines the way the familiar addition and multiplication tables of elementary arithmetic are built. The column and row headings are added to demonstrate the operation.

```
      (ι10)∘.×ι10


 ×  |  1  2  3  4  5  6  7  8  9  10
------------------------------------
 1  |  1  2  3  4  5  6  7  8  9  10
 2  |  2  4  6  8 10 12 14 16 18  20
 3  |  3  6  9 12 15 18 21 24 27  30
 4  |  4  8 12 16 20 24 28 32 36  40
 5  |  5 10 15 20 25 30 35 40 45  50
 6  |  6 12 18 24 30 36 42 48 54  60
 7  |  7 14 21 28 35 42 49 56 63  70
 8  |  8 16 24 32 40 48 56 64 72  80
 9  |  9 18 27 36 45 54 63 72 81  90
10  | 10 20 30 40 50 60 70 80 90 100
```

Outer product can be used to construct such a table for any dyadic function.

```
      (ι4)∘.+ι5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9

      10 20∘.,1 2 3
 10 1  10 2  10 3
 20 1  20 2  20 3
```

```
      (ι4)∘.=ι4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

      R←'⎕⎕⎕⎕' 'ΔΔΔΔΔ'
      3 4∘.↑R
 ⎕⎕⎕  ΔΔΔ
 ⎕⎕⎕⎕ ΔΔΔΔ
```

If *L* or *R* or both are matrixes or arrays of higher rank, each item in *L* is still matched with each item of *R*.

```
       S←3 4ρ'THEYWANTRAIN'
       S
THEY
WANT
RAIN

      'AT'∘.=S
0 0 0 0
0 1 0 0
0 1 0 0

1 0 0 0
0 0 0 1
0 0 0 0
```

***Empty Argument(s):*** If either argument is empty, *RO* is not applied. Instead, the related fill function is applied as described with “Each (dyadic)”, page 107. Fill functions for the primitive functions are given in Figure 20 on page 110.

## ⊂ Partition

`Z←L⊂R` Partitions *R* into an array of vectors specified by *L*.

*L*: Simple scalar or vector of nonnegative integers
*R*: Nonscalar
*Z*: Array of vectors

`ρZ ←→ (¯1↓ρR),+/2</0,L` (after left scalar extended)
`ρρZ ←→ ρρR`
`≡Z ←→ 1+≡R`

This function partitions its right argument at break points specified by its left argument. The result is an array of vectors made up of non-overlapping contiguous segments taken from vectors of the right argument along the last axis. The left argument is a simple vector or scalar of nonnegative integers. New items are created in the result whenever the corresponding item in *L* is greater than the previous item in *L*. If an item from *L* is `0` then the corresponding items from *R* are not included in the result. The first items in the result are created when the first nonzero in *L* is encountered.

The length of the left argument and the size of the last axis of the right argument must match, unless the left argument is a scalar or one-item vector, in which case it is extended. For empty arrays the prototype is `↑R`.

For Boolean vector or scalar *B*:

`B/R ←→ ⊃,/B⊂R`

For *L* containing no zeroes:

`R ←→ ⊃,/L⊂R`

For any appropriate *L* and *R*:

`(0≠L)/R ←→ ⊃,/L⊂R`

The following is an annotated set of examples.

Partition a string into substrings:

```
       DISPLAY 1 1 2⊂'ABC'
 .→---------.
 | .→-. .→. |
 | |AB| |C| |
 | '--' '-' |
 '∊---------'
```

Partition and delete:

```
       DISPLAY 1 0 1⊂'ABC'
 .→--------.
 | .→. .→. |
 | |A| |C| |
 | '-' '-' |
 '∊--------'
```

Lengths of the arguments must match:

```
       DISPLAY 1 0 1⊂'ABCD'
LENGTH ERROR
       DISPLAY 1 0 1⊂'ABCD'
       ^            ^
```

Partition a numeric vector into pieces:

```
       DISPLAY 2 1 2⊂10 20 30
 .→-------------.
 | .→----. .→-. |
 | |10 20| |30| |
 | '~----' '~-' |
 '∊-------------'
```

Partition adds a level of nesting:

```
      OTB←'ONE' 'TWO' 'BUCKLE MY SHOE'
      DISPLAY OTB
.→-----------------------------.
| .→--. .→--. .→-------------. |
| |ONE| |TWO| |BUCKLE MY SHOE| |
| '---' '---' '--------------' |
'∊-----------------------------'
```

```
     DISPLAY 1 1 2⊂OTB
.→-------------------------------------.
| .→------------. .→-----------------. |
| | .→--. .→--. | | .→-------------. | |
| | |ONE| |TWO| | | |BUCKLE MY SHOE| | |
| | '---' '---' | | '--------------' | |
| '∊------------' '∊-----------------' |
'∊-------------------------------------'
```

Examples with blank delimiters between words:

```
      X←'  A STITCH  IN TIME   '
      DISPLAY X
.→--------------------.
|  A STITCH  IN TIME   |
'----------------------'
```

Partition and discard blank delimiters:

```
      DISPLAY (' '≠X)⊂X
.→-------------------------.
| .→. .→-----. .→-. .→---. |
| |A| |STITCH| |IN| |TIME| |
| '-' '------' '--' '----' |
'∊-------------------------'
```

Keep delimiters on the ends:

```
      DISPLAY (1+' '≠X)⊂X
.→-------------------------------------.
| .→-. .→-. .→-------. .→--. .→------. |
| |  | |A | |STITCH  | |IN | |TIME   | |
| '--' '--' '--------' '---' '-------' |
'∊-------------------------------------'
```

Keep delimiters on the beginnings:

```
      DISPLAY (1+~' '≠X)⊂X
.→-------------------------------------.
| .→--. .→------. .→---. .→----. .→--. |
| |  A| | STITCH| |  IN| | TIME| |   | |
| '---' '-------' '----' '-----' '---' |
'∊-------------------------------------'
```

Partition a matrix at blank columns:

```
 M←3 12⍴'1   10 3.1422  100 6.2833 1000 9.425'
      DISPLAY M
.→-----------.
↓1   10 3.142|
|2  100 6.283|
|3 1000 9.425|
'------------'

      DISPLAY (~∧⌿' '=M)⊂M
.→-------------------.
↓ .→. .→---. .→----. |
| |1| |  10| |3.142| |
| '-' '----' '-----' |
| .→. .→---. .→----. |
| |2| | 100| |6.283| |
| '-' '----' '-----' |
| .→. .→---. .→----. |
| |3| |1000| |9.425| |
| '-' '----' '-----' |
'∊-------------------'
```

## ⊂[ ] Partition with Axis

`Z←L⊂[X]R`    Partitions *R* into an array of vectors specified by *L* along axis *X*.

*L*: Simple scalar or vector of nonnegative integers
*R*: Nonscalar
*Z*: Array of vectors
*X*: Simple scalar or one-item vector;
nonnegative integer: `X∊⍳⍴⍴R`

Implicit argument: `⎕IO`

```
X⊃⍴Z   ←→   +/2</0,L
 ⍴⍴Z   ←→   ⍴⍴R
  ≡Z   ←→   1+≡R
```

Partition with axis is similar to partition except that the vectors are selected along axis *X*. The shape of the result is the same as the shape of the right argument except for axis *X*.

For Boolean vector or scalar *B*:

```
B/[X]R ←→ ⊃[X],/[X]B⊂[X]R
```

For *L* containing no zeroes:

```
R ←→ ⊃[X],/[X]L⊂[X]R
```

For any appropriate *L* and *R*:

```
(0≠L)/[X]R ←→ ⊃[X],/[X]L⊂[X]R
```

Partition with axis for a high rank *R* is based on partition defined on a vector *R* as follows:

```
L ⊂[I] R  ←→  ⊃[I] (⊂L) ⊂¨ ⊂[I]R
```

```
      DISPLAY N←4 3⍴⍳12
.→-------.
↓ 1  2  3|
| 4  5  6|
| 7  8  9|
|10 11 12|
'~-------'

      DISPLAY 1 0 1 1⊂[1]N
.→--------------------.
↓ .→.    .→.    .→.    |
| |1|    |2|    |3|    |
| '~'    '~'    '~'    |
| .→---. .→---. .→---. |
| |7 10| |8 11| |9 12| |
| '~---' '~---' '~---' |
'∊--------------------'
```

---

# ○ Pi Times

`Z←○R` Multiplies any number by π (approximately `3.1415926535897933`).

`R` and `Z`: Numeric

*Scalar Function*

```
      ○1
3.141592654
      ○3J2
9.424777961J6.283185307
```

```
      ○¯2
¯6.283185307
      ○2.75*2
23.75829444
```

**Note:** The last expression in the right column calculates the area of a circle whose radius is 2.75 by using the formula $\pi r^2$

# ⊃ Pick

`Z←L⊃R`    Selects an item of `R` as specified by the path indexes `L`.

`L`: Scalar or vector whose depth is `≤2`; integer or empty

Implicit argument: `⎕IO`

```
 ρZ    ←→  Depends on the shape of the selected item
ρρZ    ←→  Depends on the rank of the selected item
```

Pick enables you to select any item at any depth from an array. The *N*th item of *L* specifies an index to one item at depth *N* in *R*. The depth at which the selection is made depends on *L*, as explained in the following sections.

***Scalar or One-Item Left Argument:*** If *L* is a scalar or one-item vector, the item selected is from the outermost structure.

*Example 1*:

```
      R←'FOUR' 'TO' 'GO'
      ≡R
2
      ⎕IO←1             │        ⎕IO←0
      Z←2⊃R             │        1⊃R
      Z                 │  TO
TO                      │
      ρZ                │        ⎕IO←1
2                       │
      ≡Z                │
1                       │
```

*Example 2*:

```
      A←'S' 'SI' ('SIR' 'SIRE')
      ≡A
3

      W←2⊃A             │        X←3⊃A
      W                 │        X
SI                      │   SIR SIRE
      ρW                │        ρX
2                       │  2
      ≡W                │        ≡X
1                       │  2
```

To select from the outermost structure of a matrix or higher rank array, *L* must be a one-item vector or scalar whose only item is a vector. Each item of the vector in *L* corresponds to an axis of *R*.

```
      C←2 2⍴'ONE' 'TWO' 'BUCKLE' ('MY' 'SHOE')
      DISPLAY C
.→-------------------------.
↓ .→--.    .→--.           |
| |ONE|    |TWO|           |
| '---'    '---'           |
| .→-----. .→-----------.  |
| |BUCKLE| | .→-. .→---. | |
| '------' | |MY| |SHOE| | |
|          | '--' '----' | |
|          '∊------------' |
'∊-------------------------'
```

```
      ⍴C
2 2

      ≡C
3
```

```
      Y←(⊂2 2)⊃C

      Y
 MY SHOE
      ⍴Y
2

      ≡Y
2
```

```
      D←C,[.5]2 2⍴'THREE' 'FOUR' 'SHUT' ('THE' 'DOOR')
      DISPLAY D
.→---------------------------.
↓↓ .→--.    .→--.            |
|| |ONE|    |TWO|            |
|| '---'    '---'            |
|| .→-----. .→-----------.   |
|| |BUCKLE| | .→-. .→---. |  |
|| '------' | |MY| |SHOE| |  |
||          | '--' '----' |  |
||          '∊------------'  |
||                           |
|| .→----.  .→---.           |
|| |THREE|  |FOUR|           |
|| '-----'  '----'           |
|| .→---.   .→------------.  |
|| |SHUT|   | .→--. .→---. | |
|| '----'   | |THE| |DOOR| | |
||          | '---' '----' | |
||          '∊-------------' |
''∊--------------------------'
```

```
      ⍴D
2 2 2

      ≡D
3
```

```
      Q←(⊂2 1 2)⊃D
      Q
FOUR

      ⍴Q
4
      ≡Q
1
```

**Specifying the Left Argument:** The shape of *L* can be no greater than the depth of the item from which the selection is being made. Successive items of *L* penetrate deeper into the structure. For example:

```
Q←'FLY' 'PAPER'

2 4⊃Q
▲ ▲
│ └───Selects 'E' from 'PAPER'
│
│
└───Selects 'PAPER' from 'FLY' 'PAPER'
```

If the right argument is a multidimensional array, the first item of *L* must be a vector whose length is the rank of the right argument. For example:

```
      S←2 3⍴'AB' 'CD' 'EF' 'GH' 'IJ' 'KL'
      S
 AB CD EF
 GH IJ KL

      (1 3) 2⊃S
       ▲     ▲
       │     └───(2)—Selects 'F' from 'EF'
       └─────────(1 3)—Selects 'EF' (first row, third column) from S
```

If *L* is empty, selection is from *R* at depth 0. Therefore, all of *R* is selected:

```
L⊃R ←→ R

      M←'B' 'BA' ('BAT' 'BATH')
```

```
      M
 B BA    BAT BATH
      ≡M
3
      N←(⍳0)⊃M
      N
 B BA    BAT BATH
      ≡N
3
```

```
      A←(⍳0)⊃0 2⍴0
      ⍴A
0 2
      ≡A
1
```

Some additional examples follow. Each uses the *DISPLAY* function to show the structure of the array from which the selection is being made.

```
      H←2 2⍴'BUCKS' 'TWANG' 'LYMPH' 'FROZE'
      DISPLAY H
.→----------------.
↓ .→----. .→----. |
| |BUCKS| |TWANG| |
| '-----' '-----' |
| .→----. .→----. |
| |LYMPH| |FROZE| |
| '-----' '-----' |
'∊----------------'
      ≡H
2

      S←(2 1) 4⊃H
      S
P
      ≡S
0

      G←'I' 'AM' ('FOR' 'APL2')
      DISPLAY G
 .→------------------------.
|   .→-. .→-------------.  |
| I |AM| | .→--. .→---. | |
| - '--' | |FOR| |APL2| | |
|        | '---' '----' | |
|        '∊-------------' |
'∊------------------------'
      ≡G
3

      T←3 2⊃G
      T
APL2
      ⍴T
4
      ≡T
1

      3 2 1⊃G
A
```

```
      E←2 3ρ'CRY' 'VOX' 'KID' 'JAB' (2 3ρι6) ('LEG' 'NTH')
      DISPLAY E
.→------------------------------.
↓ .→--. .→--.   .→--.           |
| |CRY| |VOX|   |KID|           |
| '---' '---'   '---'           |
| .→--. .→----. .→------------. |
| |JAB| ↓1 2 3| | .→--. .→--. | |
| '---' |4 5 6| | |LEG| |NTH| | |
|       '~----' | '---' '---' | |
|               '∊------------' |
'∊------------------------------'
      (2 2) (2 3)⊃E
6
      ρE
2 3
      ≡E
3
```

```
      U←(2 3) 2⊃E
      U
NTH
      ≡U
1
```

```
      J←(2 3) 2 3⊃E
      J
H
      ≡J
0
```

```
      K←'ELM' 'TAX' 'SPY' 'JOB' 'WIN'
      K←2 3ρK,(2 2ρ'QUE' 'ZiG' 'HaD' 'FoR')
      DISPLAY K
.→----------------------------.
↓ .→--. .→--. .→--.           |
| |ELM| |TAX| |SPY|           |
| '---' '---' '---'           |
| .→--. .→--. .→------------. |
| |JOB| |WIN| ↓ .→--. .→--. | |
| '---' '---' | |QUE| |ZiG| | |
|             | '---' '---' | |
|             | .→--. .→--. | |
|             | |HaD| |FoR| | |
|             | '---' '---' | |
|             '∊------------' |
'∊----------------------------'
      ρK
2 3
      ≡K
3
```

```
      P←(2 3)(1 2)⊃K
      P
ZI̲G
      ≡P
1
```

```
      I←(2 3)(1 2) 3⊃K
      I
G
      ≡I
0
```

***Compared with First:*** ↑ First, page 131, selects the first item of $R$ taken in row major order:

`↑R ←→ (⊂(ρρR)ρ1)⊃R` (for nonempty $R$)

***Selective Specification:*** Pick can be used for selective specification:

```
      B←'P' 'PI' ('PIE' 'PIER')
      (2⊃B)←'MY'

      B
 P MY   PIE PIER

      (2 1⊃B)←'TR'
      B
 P   TR Y    PIE PIER

      (3 2 1⊃B)←'T'
      B
 P   TR Y    PIE TIER
```

## * Power

`Z←L*R`    Raises the base *L* to the *R*th power.

*L*, *R* and *Z*: Numeric

*Scalar Function*

Power is the algebraic exponentiation function. *L* and *R* may be any number; however, if *L* is `0`, *R* must be a nonnegative real number.

If *R* is a nonnegative integer, `Z ↔ ×/R⍴L`. This identity has two implications: if *R* is `0`, *Z* is `1`; if *R* is `1`, *Z* is *L*.

```
      4*3
64
      10*0
1
```

```
      2*0 1 2 3 4 5
1 2 4 8 16 32
      10*1
10
```

Power is generalized to nonpositive, noninteger, and nonreal numbers in order to preserve the relation:

`L*A+B ↔ (L*A)×L*B`

Familiar consequences of this extension are that:

- `L*-R` is the reciprocal of `L*R`.
- `L*÷R` is the *R*th root of *L*. In particular, the square root of *L* is `L*÷2` or `L*.5`. In cases where there are multiple roots, the result is the one with the smallest nonnegative angle in the complex plane. The odd root of a nonreal number is a nonreal number.

```
      5*¯2
0.04
      ¯16*÷2
0J4
      ¯125*÷3
2.5J4.330127019
```

```
      16*÷2
4
      125*÷3
5
      0J2*3
0J¯8
```

# , Ravel

`Z←,R` Creates a vector from the items in `R`, taken in row-major order.

`Z`: Vector

```
 ⍴Z   ←→  ,×/⍴R
⍴⍴Z   ←→  ,1
```

Ravel is related to reshape (⍴), page 225, as follows: `,R ←→ (×/⍴R)⍴R`

```
      A←3 3⍴⍳9
      A
1 2 3
4 5 6
7 8 9
      Z←,A
      Z
1 2 3 4 5 6 7 8 9
      ⍴Z
9
```

```
      B←2 2 4⍴'BAD FOG GO  SLOW'
      B
BAD
FOG

GO
SLOW
      ⍴B
2 2 4
      M←,B
      M
BAD FOG GO  SLOW
      ⍴M
16
```

***Ensure Vector Argument:*** Ravel can be used to ensure that an argument is a vector.

```
      C←4
      ⍴C
(empty)
      ≡C
0
```

```
      W←,C
      ⍴W
1
      ≡W
1
```

***Compared with Enlist:*** ∊ Enlist, page 118, creates a simple vector whose items are the simple scalars in *R*. If all items of *R* are simple scalars, **,***R* ←→ ∊*R*.

***Selective Specification:*** Ravel can be used for selective specification:

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

      S
 1 2  3 4
 5 6  7 8
      ≡S
2

      (,S)←'ABCD'
      S
AB
CD
      ⍴S
2 2
      ≡S
1
```

## ,[ ] Ravel with Axis

`Z←,[X]R` Creates an array that contains the items of `R` reshaped according to axes `X`: If `X` is a fraction, a new axis of length 1 is formed; if `X` is an integer, the `X` axes of `R` are combined.

`X`: Simple scalar fraction *or* simple scalar or vector of nonnegative integers *or* empty

Implicit argument: `⎕IO`

```
 ρZ    ↔ Depends on the value of X
ρρZ    ↔ Depends on the value of X
```

Ravel with axis has three cases, based on the value of `X`: fractional, integer, or empty.

***When*** `X` ***Is a Fraction***: `⌈X` is at least one, but less than or equal to `1+ρρR`. A new axis of length 1 is created before the `⌈X`th axis. The rank of the result is one greater than the rank of `R`:

```
ρρZ ↔ 1+ρρR
```

The shape of the result is:

```
ρZ ↔ (1,ρR)[⍋X,⍳ρρR]
 Z ↔ (ρZ)ρR
```

```
      A←2 3ρ'TENSIX'
      A
TEN
SIX
      Z←,[.1]A
      Z
TEN
SIX
      ρZ
1 2 3
      Y←,[1.1]A
      Y
TEN

SIX
      ρY
2 1 3
```

```
      W←,[2.1]A
      W
T
E
N

S
I
X
      ρW
2 3 1

      B←10 15 20
      V←,[1.1]B
      V
10
15
20
      ρV
3 1
```

***When X Is an Integer***: X must be a simple scalar or vector of nonnegative integers. If X is a scalar, Z is R.

If X is a vector, it must contain contiguous axes in ascending order of R. For example, for a rank-3 array, X may be 1 2 *or* 2 3 *or* 1 2 3. The axes indicated by X are combined to form a new array whose rank is `1+(⍴⍴R)-⍴,X`.

```
      C←3 2 4⍴⍳24
      C
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24

      P←,[2 3]C
      P
 1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
      ⍴P
3 8

      J←,[1 2]C
      J
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
      ⍴J
6 4

      A←'ANT' 'BOAR' 'CAT' 'DOG' 'ELK' 'FOX' 'GNU'
      B←'HEN' 'IBEX' 'JIRD' 'KITE' 'LAMB' 'MICE'
      C←'NENE' 'OX' 'PIG' 'QUAIL' 'RAT' 'SEAL'
      D←4 3 2⍴A,B,C,'TITI' 'VIPER' 'WOLF' 'YAK' 'ZEBRA'
      D
 ANT  BOAR  CAT
 DOG  ELK   FOX

 GNU  HEN   IBEX
 JIRD KITE  LAMB

 MICE NENE  OX
 PIG  QUAIL RAT

 SEAL TITI  VIPER
 WOLF YAK   ZEBRA
```

```
      ⍴D
4 2 3
      ≡D
2

      M←,[1 2]D
      M
 ANT  BOAR  CAT
 DOG  ELK   FOX
 GNU  HEN   IBEX
 JIRD KITE  LAMB
 MICE NENE  OX
 PIG  QUAIL RAT
 SEAL TITI  VIPER
 WOLF YAK   ZEBRA
      ⍴M
8 3
      ≡M
2
```

Ravel, page 202, is equivalent to ravel with axis when *X* includes all axes of *R*: `,R ←→ ,[⍳⍴⍴R]R`.

**When *X* Is Empty**: When *X* is empty, a new last axis (columns) of length 1 is created. The rank of the result is one greater than the rank of *R*, and the shape of the result is `(⍴R),1`.

For vectors only:

`,[⍳0]R ←→ ,[1.1]R`

```
      H←2 3⍴⍳6
      N←,[⍳0]H
      N
1
2
3

4
5
6

      ≡N
1
```

```
      K←'PRUNE' 'PEAR' 'FIG'
      ⍴K
3
      ≡K
2
      I←,[⍳0]K
      I
 PRUNE
 PEAR
 FIG
      ⍴I
3 1
      ≡I
2
```

***Turning an Array into a Matrix:*** The following expression can be used to turn any array *R* into a matrix:

`,[⍳⍴⍴R],[.5]R`

For example:

```
      E←3 2 5⍴⍳30
      ,[⍳⍴⍴E],[.5]E
 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
```

```
      G←'JIM' 'ED' 'MIKE'
      ⍴G
3
      F←,[⍳⍴⍴G],[.5]G
      F
 JIM ED MIKE
      ⍴F
1 3
```

***Selective Specification:*** Ravel with axis can be used for selective specification.

```
      Q←2 3 4⍴⍳24
      (,[2 3]Q)←2 12⍴-⍳24
      Q
 ¯1  ¯2  ¯3  ¯4
 ¯5  ¯6  ¯7  ¯8
 ¯9 ¯10 ¯11 ¯12

¯13 ¯14 ¯15 ¯16
¯17 ¯18 ¯19 ¯20
¯21 ¯22 ¯23 ¯24
      ⍴Q
2 3 4
```

# ÷ Reciprocal

`Z←÷R`  Divides 1 by `R`.

`R` and `Z`: Numeric, nonzero

*Scalar Function*

Reciprocal is the arithmetic reciprocal function:

`÷R ←→ 1÷R`

```
      ÷4
0.25
      ÷1 .2 ¯3
1 5 ¯0.3333333333
```

```
      ÷2J2
0.25J¯0.25
      ÷0J1 0J¯1
0J¯1 0J1
```

## / Reduce (from Slash)

`Z←LO/ R` Has the effect of placing the function `LO` between adjacent pairs of items along the last axis of `R` and evaluating the resulting expression for each subarray.

`LO`: Dyadic function

```
 ρZ   ←→  ¯1↓ρR
ρρZ   ←→  0⌈¯1+ρρR
```

If `R` is the vector `A B C`, the `LO`-reduction is defined as follows:

```
LO/A B C ←→ ⊂ A LO B LO C
```

If `LO` is a scalar function, the reduction of a simple vector is a simple scalar. If the right argument is nested, the depth of the result is the same as that of the right argument.

```
      +/1 2 3 4 5
15

      Z←+/(1 2)(3 4)(5 6)
      Z
 9 12
      ρZ
(empty)
      ≡Z
2
```

```
      ∨/0 0 1 1 0
1

      W←,/'AB' 'CD' 'EF'
      W
 ABCDEF
      ρW
(empty)
      ≡W
2
```

If `R` is a matrix or array of higher rank, the subarrays along the last axis are treated as vectors, and the function is applied between adjacent items along the last axis, so that for a matrix `R`:

```
      Z[I]←LO/R[I;]
```

For a rank-3 array:

```
      Z[I;J]←LO/R[I;J;]
```

Higher-rank arrays follow a similar pattern. In general for all nonscalars:

`LO/R ←→ ⊃LO/¨⊂[⍴⍴R]R`

```
      M←3 4⍴⍳12
      M
1  2  3  4
5  6  7  8
9 10 11 12
      +/M
10 26 42
```

```
      R←3 2⍴'ACEGIK',¨'BDFHJL'
      R
 AB  CD
 EF  GH
 IJ  KL
      Y←,/R
      ⍴Y
3
      ≡Y
2
```

*R* ***Is a Scalar or Its Last Axis Is One***: If *R* is a scalar, *Z* is *R*. If the last axis of *R* (`¯1↑⍴R`) is one, *Z* is `(¯1↓⍴R)⍴R`. The function *LO* is not applied in either case.

```
      =/15
15
```

```
      N←4 1⍴2 4 6 8
      ÷/N
2 4 6 8
```

***Empty*** *R*: If the last axis of *R* is 0, the function *LO* is not applied. Instead, a related function called the *identity* function is applied with argument `↑R` (prototype of *R*). The result returned is `(¯1↓⍴R)⍴⊂I`, where *I* is the value produced by the identity function for the function *LO*. Figure 27 on page 211 and Figure 28 on page 212 show the identity function for each primitive function that has one.

```
      +/⍳0
0
```

```
      ×/2 3 0⍴⊂0 0
1 1  1 1  1 1
1 1  1 1  1 1
```

The identity function related to a defined function cannot be specified, and an attempt to reduce an empty argument with a defined function generates a *DOMAIN ERROR*.

***Dyadic Scalar Function Identities::*** The identity function for each dyadic scalar function is defined as:

```
Z←SR⍴⊂R+F/⍳0
```

where *R* is the prototype of the right argument and *SR* is the shape of the result.

**Note:** In Figure 27, *A* is the array satisfying the identity and *M* is `7.2370055773322621E75`.

*Figure 27. Identity Items for Dyadic Scalar Functions*

| Function | F | Identity<br>F/⍳0 | Left/<br>Right | Identity<br>Restriction |
|---|---|---|---|---|
| Add | + | 0 | L R | |
| Subtract | - | 0 | R | |
| Multiply | × | 1 | L R | |
| Divide | ÷ | 1 | R | |
| Residue | \| | 0 | L | |
| Minimum | ⌊ | M | L R | |
| Maximum | ⌈ | -M | L R | |
| Power | * | 1 | R | |
| Logarithm | ⍟ | | none | |
| Circular | ○ | | none | |
| Binomial | ! | 1 | L | |
| And | ∧ | 1 | L R | ∧/∊A∊0 1 |
| Or | ∨ | 0 | L R | ∧/∊A∊0 1 |
| Less | < | 0 | L | ∧/∊A∊0 1 |
| Not Greater | ≤ | 1 | L | ∧/∊A∊0 1 |
| Equal | = | 1 | L R | ∧/∊A∊0 1 |
| Not Less | ≥ | 1 | R | ∧/∊A∊0 1 |
| Greater | > | 0 | R | ∧/∊A∊0 1 |
| Not Equal | ≠ | 0 | L R | ∧/∊A∊0 1 |
| Nand | ⍲ | | none | |
| Nor | ⍱ | | none | |

***Dyadic Nonscalar Function Identities:*** In the definitions of the identity functions in Figure 28, *R* is the prototype of the right argument and *SR* is the shape of the result. *A* in the Identity Restriction column is the array satisfying the identity.

*Figure 28. Identity Functions for Primitive Dyadic Nonscalar Functions*

| **Function** | **F** | **Identity Function**<br>`Z←SRρ⊂....` | **Left/ Right** | **Identity Restriction** |
|---|---|---|---|---|
| Reshape | ρ | `ρR` | L | |
| Catenate | , | `((¯1↓ρR),0)ρ⊂((¯1↓ρR),0)ρR` | L R | `1≤ρρA` |
| Rotate | ⌽ | `(-1↓ρR)ρ0` | L | |
| Rotate | ⊖ | `(1↓ρR)ρ0` | L | |
| Transpose | ⍉ | `⍳ρρR` | L | |
| Pick | ⊃ | `⍳0` | L | |
| Drop | ↓ | `(ρρR)ρ0` | L | |
| Take | ↑ | `ρR` | L | |
| Without Matrix | ~ | `⍳0` | R | `1=ρρA` |
| Divide | ⌹ | `(⍳↑ρR)∘.=⍳↑ρR` | R | `1≤ρρA` |

***Derived Functions of Special Interest:*** The following reduction functions derived from the slash operator have wide application:

- Summation (∑) (`+/`)
- Alternating Sum (`-/`)
- Product (π) (`×/`)
- Alternating product (`÷/`)
- Smallest (`⌊/`)
- Largest (`⌈/`)
- Boolean vector contains at least one `1` (`∨/`)
- Boolean vector contains all `1`s (`∧/`)

The last two reductions are useful in determining the truth of various statements about a simple vector *R*. For instance:

- Every item of *R* is positive: `∧/R>0`
- Every item of *R* is odd: `∧/2|R`
- At least one item of *R* is even: `∨/~2|R`

## / Reduce N-Wise (from Slash)

`Z←L LO/ R`  Similar to reduce, except that *L* defines the number of items along the last axis to be considered in each application of the function to the subarrays along the last axis of *R*.

*LO*: Dyadic function
*L*: Simple scalar or one-item vector, integer

```
 ρZ   ←→ (¯1↓ρR),1+(¯1↑ρR)-|L
ρρZ   ←→ ρρR
```

The absolute value of *L* may be no more than one plus the length of the last axis of *R*:

```
      (|L)≤1+¯1↑ρR
```

*L* can be considered as a moving window for determining successive items of *Z*.

***Positive Left Argument:*** If *L* is positive, the window starts at the left of the subarray along the last axis and moves right. At each item of *R*, the window stops and the *LO*-reduction of the items in the window is taken.

To demonstrate, the examples below vary *L* for the vector *R*:

```
      R←1 2 3 4 5 6

      6+/R
21

      4+/R
10 14 18

      2+/R
3 5 7 9 11
```

```
      5+/R
15 20

      3+/R
6 9 12 15

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

Additional examples are shown below, including one with a nested right argument.

```
      2+/(1 2)(3 4)(5 6)
 4 6  8 10

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

      2+/M
 3  5  7
11 13 15
19 21 23
```

```
      2,/'ABCDEF'
AB BC CD DE EF

      B←3 3⍴'ABCDEFGHI'
      B
ABC
DEF
GHI

      2,/B
AB BC
DE EF
GH HI
```

***Negative Left Argument:*** If `L` is negative, the contents of the window are reversed just before the reduction.

```
      ¯2-/1 4 9 16 25
3 5 7 9

     ¯2,/'ABCDEF'
 BA CB DC ED FE
```

```
      2-/1 4 9 16 25
¯3 ¯5 ¯7 ¯9

      2,/'ABCDEF'
 AB BC CD DE EF
```

If `LO` is commutative (that is, `A LO B ↔ B LO A`), the sign of `L` does not affect the result.

```
      3×/⍳6
6 24 60 120
```

```
      ¯3×/⍳6
6 24 60 120
```

***Zero Left Argument:*** If `L` is 0, the identity function of `LO` is applied instead. See the discussion under “Reduce (from Slash)”, on page 209. The result is a `(⍴R)+(⍴⍴R)=⍳⍴⍴R` array of identity items for the primitive function `LO`. Identity items are listed in Figure 27 on page 211 and Figure 28 on page 212.

```
       0×/⍳5
1 1 1 1 1 1
```

`R` may be empty only if `L` is 0.

***Derived Functions of Special Interest:*** The following functions derived from reduce n-wise have wide application:

- First difference (`¯2-/R`)
- Yearly running total (`12+/R`)

## /[ ] ⌿[ ] Reduce N-Wise with Axis (from Slash)

`Z←L LO/[X]R`   Similar to reduce with axis except that `L` defines the number of items along the `X`th axis to be considered in each application of the function to the subarrays along the `X`th axis.

`LO`: Dyadic function
`L`: Simple scalar or one-item vector, integer
`X`: Simple scalar or one-item vector, integer: `X∊ιρρR`

Implicit argument: `⎕IO`

```
(ρZ)[X]   ←→  1+(ρR)[,X]-|L
     ρρZ   ←→  ρρR
```

The absolute value of `L` can be no more than one plus the length of the `X`th axis of `R`:

```
(|L)≤1+(ρR)[X]
```

`L` can be considered as a moving window for determining successive items of `Z`.

***Positive Left Argument:*** If `L` is positive, the window starts at the front of the subarray along the `X`th axis and moves backward. At each item of `R`, the window stops, and the `LO`-reduction of the items in the window is taken.

To demonstrate, the examples below vary `L` for the matrix `R`:

```
      R←3 4ρι12
      R
 1  2  3
 4  5  6
 7  8  9
10 11 12

      4+/[1]R
22 26 30

      2+/[1]R
 5  7  9
11 13 15
17 19 21
```

```
      3+/[1]R
12 15 18
21 24 27
      1+/[1]R
 1  2  3
 4  5  6
 7  8  9
10 11 12
```

The example below shows the application of n-wise reduce to a nested right argument.

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

      2×/[1]C
 5 12   21 32
 45 60  77 96
```

***Negative Left Argument:*** If *L* is negative, the contents of the window are reversed just before the reduction is applied.

```
      ¯2-/10 20 30 40
10 10 10
      ¯2-/10 8 20 ¯3
¯2 12 ¯23
```

If *LO* is commutative (that is, *A LO B* ←→ *B LO A*), the sign of *L* does not affect the result.

***Zero Left Argument:*** If *L* is 0, the identity function of *LO* is applied instead. See the discussion under “Reduce (from Slash),” on page 209. Identity items are listed in Figure 27 on page 211 and Figure 28 on page 212.

```
      0×/[1]R
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
```

*R* may be empty only if *L* is 0.

***Derived Functions of Special Interest:*** The derived functions listed under reduce n-wise (see “/ Reduce N-Wise (from Slash)” on page 213) apply to reduce n-wise with axis.

## /[ ] ⌿[ ] Reduce with Axis (from Slash)

`Z←LO/[X] R` Similar to reduce, except that the function `LO` is placed between adjacent pairs of items along the `X`th axis of `R`.

`LO`: Dyadic function
`X`: Simple scalar or one-item vector, integer: `X∊ιρρR`.

Implicit argument: `⎕IO`

```
 ρZ  ←→ (ρR)[(ιρρR)~X]
ρρZ  ←→ 0⌈¯1+ρρR
```

Reduce with axis is similar to reduce except that any axis, instead of only the last, may be specified:

`LO/[X]R ←→ ⊃LO/¨⊂[X]R`

and
`LO/[ρρR]R ←→ LO/R`

```
      M←3 4ρι12
      M
1  2  3  4
5  6  7  8
9 10 11 12
      +/[1]M
15 18 21 24


      ,/[1]2 3ρι6
 1 4  2 5  3 6
```

```
      N←2 3 4ρι24
      N
 1  2  3  4
 5  6  7  8
 9 10 11 12

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

      +/[1]N
14 16 18 20
22 24 26 28
30 32 34 36

      +/[2]N
15 18 21 24
51 54 57 60
```

***Applied to First Axis:*** The symbol `⌿` is an alternate symbol for `/[1]`. However, if `⌿` is followed by an axis (`⌿[X]`), it is treated as `/[X]`.

```
      ×/[1]M
45 120 231 384
```

```
      ×⌿M
45 120 231 384
```

***`X`th Axis of `R` Has Length One***: If the `X`th axis of `R` (`¯1↑⍴R`) has length one, `Z` is:

```
(⍴R)[(⍳⍴⍴R)~X]⍴R
```

```
      N←2 1 4⍴2×⍳8
      ÷/[2]N
 2  4  6  8
10 12 14 16
```

***Empty `R`***: If the `X`th axis of `R` is 0 (`0=¯1↑⍴R`), the function `LO` is not applied. Instead a related function called the *identity* function is applied with argument `↑R` (prototype of `R`). The result is:

```
(⍴R)[⍳(⍴⍴R)~X]⍴⊂I
```

where `I` is the value returned from the identity function for the function `LO`. Figure 27 on page 211 and Figure 28 on page 212 show the identity items for each primitive function.

```
      ÷/[2]2 0 3⍴0
1 1 1
1 1 1
```

The identity function related to a defined function cannot be specified, and an attempt to reduce an empty argument with a defined function generates a `DOMAIN ERROR`.

***Derived Functions of Special Interest:*** All the derived functions listed under reduce (from slash) (see "`/` Reduce (from Slash)" on page 209) apply to reduce with axis.

## < ≤ = ≥ > ≠ Relational Functions

| | |
|---|---|
| `Z←L<R` | Less than |
| `Z←L≤R` | Less than or equal |
| `Z←L=R` | Equal |
| `Z←L≥R` | Greater than or equal |
| `Z←L>R` | Greater than |
| `Z←L≠R` | Not equal |

`L` and `R` Numeric real for `< ≤ ≥ >`
`Z`: Boolean

Implicit Argument: `⎕CT`

*Scalar Functions*

Each relational function determines whether corresponding items of the arguments satisfy the relationship. The result is `1` if the relationship for corresponding items is true (within the comparison tolerance `⎕CT`), and `0` otherwise.

```
      'TRIAL'='TRAIL'
1 1 0 0 1
```

```
      8 ¯2 6 ¯4 0<0
0 1 0 1 0
```

Like other scalar functions, the relational functions apply corresponding items of an array throughout the entire structure. Scalar extension is performed as necessary for conformability. The example below uses the defined function `DISPLAY` to illustrate the result of a relational function.

```
      L←('IN' 'OUT') (9 5 6) (⊂2 2⍴⍳4)
      R←('IT' 'BUT') 6 (2 2⍴1 8 5 4)

      DISPLAY L=R
.→--------------------------------.
|.→-----------..→----..→---------.|
||.→--..→----.||0 0 1|↓.→--..→--.||
|||1 0||0 1 1||'~----'|↓1 0|↓0 0|||
||'~--''~----'|       ||0 0||0 0|||
|'∊-----------'       |'~--''~--'||
|                     |.→--..→--.||
|                     |↓0 0|↓0 0|||
|                     ||0 0||0 1|||
|                     |'~--''~--'||
|                     '∊---------'|
'∊--------------------------------'
```

# / Replicate (from Slash)

`Z←LO/R` Repeats each subarray along the last axis under the control of the vector `LO`.

`LO`: Simple scalar or vector, integer
`Z`: Nonscalar array

```
¯1↓⍴Z  ←→  ¯1↓⍴R
  ⍴⍴Z  ←→  ⍴⍴R
```

`LO` determines the pattern and type of replication of subarrays of `R`, as follows:

> If `LO[I]` (an item of `LO`) is positive, the corresponding subarray of `R` is replicated `LO[I]` times.
>
> If `LO[I]` is zero, the corresponding subarray is dropped from the result. (If `∧/LO=0`, `Z` has a zero shape for the last axis.)
>
> If `L[I]` is negative, the fill item of the corresponding subarray of `R` is replicated `|L[I]` fill items. The fill item is determined by the type of the first item in the `I`th subarray along the last axis.

```
      1 2 3 4/'ABCD'
ABBCCCDDDD

      R←3 2⍴'A' 8 7 6 5 4
      R
A 8
7 6
5 4
      2 ¯1 1 ¯2/R
A A   8
7 7 0 6 0 0
5 5 0 4 0 0
```

```
      1 2 ¯1 3 ¯2/6 7 8
6 7 7 0 8 8 8 0 0

      0 2 0 1/'SOAP'
OOP
```

***Conformability:*** If `¯1↑⍴R` is not `1`, it must be equal to `+/LO≥0`. For scalar `LO` or `R` or if `¯1↑⍴R` is `1`, the following extensions are applied before the replication is evaluated:

- If `LO` is a scalar or one-item vector, it is extended to `¯1↑1,⍴R`.
- If `R` is a scalar, it is treated as a one-item vector.
- If `¯1↑⍴R` is 1, `R` is replicated along the last axis `+/LO≥0` times.

If `LO` is not extended, `¯1↑⍴Z` is `+/|LO`.

```
      2/4 5
4 4 5 5

      S←,[⍳0]'TON'
      1 ¯2 2/S
T  TT
O  OO
N  NN
```

```
      1 ¯2 3/6
6 0 0 6 6 6
```

**Effect on Depth:** Replicate does not change the depth of any item; however, the depth of the result may be different from that of `R` if `LO[I]=0` should eliminate a nested item.

```
      W←'I' 'ID' ('IDE' 'IDEA')
```

```
      W
 I ID   IDE IDEA
      ≡W
3
      X←3 2 1/W
      X
 III ID ID   IDE IDEA
      ≡X
3
```

```
      P←1 2 0/W
      P
 I ID ID
      ≡P
2
```

# /[] ⌿[] Replicate with Axis (from Slash)

`Z←LO/[X]R` Repeats each subarray along the *X* axis under the control of the vector *LO*.

*LO*: Simple scalar or vector, integer or empty
*R* and *Z*: Nonscalar array
*X*: Simple scalar or one-item vector, integer: `X∊⍳⍴⍴R`

Implicit Argument: `⎕IO`

```
(⍴Z)[X]←→+/|LO
  ⍴⍴Z   ←→  ⍴⍴R
```

Replicate with axis is similar to replicate, except that replication occurs along the *X*th axis.

```
      R←3 2 4⍴⍳24
      R
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24
```

```
      2 ¯1 1/[2]R
 1  2  3  4
 1  2  3  4
 0  0  0  0
 5  6  7  8

 9 10 11 12
 9 10 11 12
 0  0  0  0
13 14 15 16

17 18 19 20
17 18 19 20
 0  0  0  0
21 22 23 24
```

***Conformability:*** The shape of *R* along the *X*th axis must be `1` or `+/LO≥0`. For scalar *LO* and *R* with a shape of `1` along the *X*th axis, the following extensions are applied before the function is evaluated:

- If *LO* is a scalar or one-item vector, it is extended to `(⍴R)[X]` items.
- If `(⍴R)[X]` is 1, *R* is replicated along the *X*th axis `+/LO≥0` times.

If *LO* is not extended, `(⍴Z)[X]` is `+/|LO` times.

```
      S←3 2 4⍴⍳24
      2/[2]S
 1  2  3  4
 1  2  3  4
 5  6  7  8
 5  6  7  8

 9 10 11 12
 9 10 11 12
13 14 15 16
13 14 15 16

17 18 19 20
17 18 19 20
21 22 23 24
21 22 23 24
```

```
      T←3 1 4⍴'ABCDEFGHIJKL'
      T
ABCD

EFGH

IJKL
      ⍴T
3 1 4

      ¯1 1/[2]T

ABCD


EFGH


IJKL

      ⍴¯1 1/[2]T
3 2 4
```

The symbol `⌿` is an alternative symbol for `/[1]`. However, if `⌿` is followed by an axis (`⌿[X]`), it is treated as `/[X]`.

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

      1 0 2 ¯1⌿M
1  2  3  4
9 10 11 12
9 10 11 12
0  0  0  0
```

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

***Effect on Nested Arrays:*** Replicate with axis does not change the depth of any item; however, the depth of the result may be different from that of *R* if *LO*[*I*]=0 eliminates a nested item.

```
      D←2 2 2ρ'HE' 'ME' 'WE' 'US' 'I' 'A' 'O' 'E'
      D
 HE ME
 WE US

 I  A
 O  E
      ρD
2 2 2
      ≡D
2

      J←0 2/[1]D
      J
IA
OE

IA
OE
      ≡J
1

      W←2 ¯1 1/[2]D
      W
 HE ME
 HE ME

 WE US

 I  A
 I  A

 O  E
     ρW
2 4 2
      ≡W
2
```

---

# ρ Reshape

`Z←LρR` Structures the items of *R* into an array of shape *L*.

*L*: Simple scalar or vector, not negative integers.

```
 ρZ  ←→ ,L
ρρZ  ←→ ρ,L
```

Items are selected from *R* in row-major order and placed into the result in row-major order.

```
      Z←2 3 4ρι24
      Z
 1  2  3  4
 5  6  7  8
 9 10 11 12

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

      ρZ
2 3 4
      ρρZ
3
```

```
      X←3 8ρZ
      X
 1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24

      ρX
3 8
      ρρX
2
```

If `(×/ρR) ≥ ×/L`, the first `×/L` items are used.

```
      3 5ρι24
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
```

If `(×/ρR) < ×/L`, items from *R* are repeated cyclically.

```
      B←'UP' 'ON' 'TO' 'BY'
      Y←2 3ρB
      Y
 UP ON TO
 BY UP ON

      5ρB
 UP ON TO BY UP
```

***Empty Argument:*** If R is empty, L must contain at least one zero. If L is empty, the result is a scalar whose only item is the first item of R. An empty character vector is treated like an empty numeric vector.

```
      S←0 2ρι0
      S
(empty)
      ρS
0 2
```

```
      V←5 6 7
      ρV
 3
      ρρV
 1
      H←(ι0)ρV or
      H←''ρV
      H
 5
      ρH
(empty)
      ρρH
 0
```

***Zero in*** **L**: If L contains at least one zero, Z is an empty array whose prototype is the prototype of R.

```
      M←3 0ρ0
      ρM
3 0
      ρρM
2
```

```
      N←2 0 6ρ5 3 2 1
      ρN
2 0 6
      ρρN
3
```

***Selective Specification:*** Reshape can be used for selective specification:

```
      T←'GROWTH'

      (2 3ρT)←2 3ρ-ι6
      T
¯1 ¯2 ¯3 ¯4 ¯5 ¯6
```

```
      (4ρT)←'ABCD'
      T
ABCD ¯5 ¯6

      (8ρT)←ι8
      T
7 8 3 4 5 6
```

# | Residue

`Z←L|R` For real positive `L` and `R`, the remainder from dividing `R` by `L`. For all numbers, `Z` is `R-L×⌊R÷L+L=0`.

**Note:** `⌊` is computed with a comparison tolerance of zero.

`L`, `R`, and `Z`: Numeric
Implicit Argument: `⎕CT`

*Scalar Function*

For real `L` and `R`:

If `L` is zero, `Z` is `R`.
If `R` is zero, `Z` is zero.
If `L` is positive, `Z≥0` and `Z<L`.
If `L` is negative, `Z≤0` and `Z>L`.

```
      10|17
7
      10|8 10 ¯4 4J3
8 0 6 4J3
```

```
      4J6|7J10
3J4
      ¯10 7J10 .3|17 5 10
¯3 ¯5J7 0.1
```

# ⌽ ⊖ Reverse

Z←⌽R  Creates an array with the items of R reversed along the last axis.

```
 ⍴Z   ←→  ⍴R
⍴⍴Z   ←→  ⍴⍴R
```

```
      A←'DESSERTS'
      ⌽A
STRESSED
```

```
      B←(1 2) (3 4) (5 6)
      ⌽B
 5 6  3 4  1 2
```

```
      C←3 5⍴'EMIT REGALTIDE '
      C
EMIT
REGAL
TIDE
      ⌽C
 TIME
LAGER
 EDIT
```

***Selective Specification:*** Reverse can be used for selective specification.

```
      D←3 4⍴⍳12

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

```
      (⌽D)←3 4⍴'STOPSPINODER'

      D
POTS
NIPS
REDO
```

⊖ means that reverse is applied along the first axis. For example:

```
      ⊖C
TIDE
REGAL
EMIT
```

# ⌽[ ] ⊖[ ] Reverse with Axis

`Z←⌽[X]R` Creates an array with items reversed along the *X*th axis.

*X*: Simple scalar or one-item vector, integer: `X∊⍳⍴⍴R`

Implicit argument: `⎕IO`

```
 ⍴Z   ←→  ⍴R
⍴⍴Z   ←→  ⍴⍴R
```

Reverse with axis is similar to reverse, except that reversal of items is done along the *X*th axis instead of along the last axis.

```
      A←2 3 1⍴'IN' 'OUT' 'UP' 'RIGHT' 'LEFT' 'DOWN'
      A
IN
OUT
UP

RIGHT
LEFT
DOWN

      ⌽[2]A
UP
OUT
IN

DOWN
LEFT
RIGHT

      ⌽[1]A
RIGHT
LEFT
DOWN

IN
OUT
UP
```

***Applied to First Axis:*** The symbol ⊖ is an alternate symbol for `⌽[1]`. If ⊖ is followed by an axis (`⊖[X]`), it is treated as `⌽[X]`.

***Selective Specification:*** Reverse with axis can be used for selective specification:

```
      B←3 4⍴⍳12

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

```
      (⌽[1]B)←3 4⍴-⍳12

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

# ? Roll

Z←?R Selects an integer at random from the population ⍳R.

R: Positive integer
Z: Integer in the set ⍳R

Implicit arguments: ⎕IO and ⎕RL

*Scalar Function*

Each integer in the population ⍳R has an equal chance of being selected.

The result depends on the value of ⎕RL. A side effect of roll is to change the value of ⎕RL (random link).

Both examples below show the value of ⎕RL prior to execution of the function. To duplicate these results, specify ⎕RL to be this value.

```
      ⎕IO←1
      ⎕RL
16807
      ?10
2

      ⎕RL
282475249
      ?10 10 10 10 10 10
8 5 6 3 1 7
```

```
      ⎕IO←0
      ⎕RL
16807
      ?10
1

      ⎕RL
282475249
      ?10 10 10 10 10 10
7 4 5 2 0 6
```

---

# ⌽ Rotate

`Z←L⌽R` Creates an array with items of `R` rotated `|L` positions along the last axis.

The sign of `L` determines the direction of the rotation.

`L`: Simple integer, either scalar or rank `¯1+⍴⍴R`

```
 ⍴Z   ←→  ⍴R
⍴⍴Z   ←→  ⍴⍴R
```

If `L` is a *nonnegative* scalar or one-item vector, `L` items are removed from the beginning of each vector along the last axis of `R` and appended to the same vector.

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

```
      B←2 5⍴'ANGLEASIDE'
      B
ANGLE
ASIDE
      2⌽B
GLEAN
IDEAS
```

If `L` is a *negative* scalar or one-item vector, `L` items are removed from the end of each vector along the last axis of `R` and prefixed to the same vector.

```
      ¯2⌽A
6 7 1 2 3 4 5
```

```
      D←2 4⍴'ACHEINKS'
      D
ACHE
INKS
      ¯1⌽D
EACH
SINK
```

If `L` is not a scalar or one-item vector, the rows of `R` are treated independently according to the corresponding items of `L`. To conform, `(⍴L)` must be `¯1↓⍴R`.

```
      H←3 3⍴'ATEEATTEA'
      H
ATE
EAT
TEA

      ¯1 0 1⌽H
EAT
EAT
EAT
```

```
      K←2 3⍴'CAT' 'BEAR' 'PONY' 'GNU' 'BIRD' 'FOX'
      K
 CAT BEAR PONY
 GNU BIRD FOX

      ⍴K
2 3

      ≡K
2

      1 2⌽K
 BEAR PONY CAT
 FOX  GNU  BIRD
```

The example below demonstrates how the left argument is formed for three-dimensional arrays. The rows of *L* correspond to the planes of *R* and the columns of *L* correspond to the rows of *R*. For example, `L[2;3]` specifies the rotation for the second plane, third row of *R*.

```
      S←2 3 5⍴'TARESSMARTEARTHSETONLAGERSHEAR'
      S
TARES
SMART
EARTH

SETON
LAGER
SHEAR

      ⍴S
2 3 5

      Q←2 3⍴4 0 ¯1 ¯2 5 1
      Q
 4 0 ¯1
¯2 5  1

      Q⌽S
STARE
SMART
HEART

ONSET
LAGER
HEARS
```

***Selective Specification:*** Rotate can be used for selective specification:

```
      W←'STRIPE'
      2⌽W
RIPEST
      (2⌽W)←'THERMO'
      W
MOTHER
```

## ⌽[ ] Rotate with Axis

| | |
|---|---|
| `Z←L⌽[X]R` | Creates an array with items of *R* rotated `∣L` positions along the *X*th axis.<br>The sign of *L* determines the direction of the rotation. |

*L*: Simple integer, scalar, or vector
*X*: Simple scalar or one-item vector, integer: `X∊⍳⍴⍴R`

Implicit argument: `⎕IO`

```
 ⍴Z   ←→  ⍴R
⍴⍴Z   ←→  ⍴⍴R
```

Rotate with axis is similar to rotate, except that removing items and appending or prefixing them is done along the *X*th axis instead of along the last axis.

```
      A←'BETTA' 'CARP' 'EEL' 'LOACH'
      B←'BAY' 'CEDAR' 'ELM' 'LARCH'
      C←3 4 1⍴A,B,'BOA' 'CAVY' 'ELAND' 'LION'
      C
BETTA
CARP
EEL
LOACH

BAY
CEDAR
ELM
LARCH

BOA
CAVY
ELAND
LION
```

```
     1⌽[1]C
BAY
CEDAR
ELM
LARCH

BOA
CAVY
ELAND
LION
BETTA
CARP
EEL
LOACH

      1⌽[2]C
CARP
EEL
LOACH
BETTA

CEDAR
ELM
LARCH
BAY

CAVY
ELAND
LION
BOA
```

***Applied to the First Axis:*** The symbol `⊖` is an alternate symbol for `⌽[1]`. However, if `⊖` is followed by an axis (`⊖[X]`), it is treated as `⌽[X]`.

```
      U←3 1ρ'ALFRED' 'THINK' 'QUICK'
      U
ALFRED
THINK
QUICK

      1⊖U
THINK
QUICK
ALFRED

      ¯1?U
QUICK
ALFRED
THINK
```

If *L* is not a scalar or one-item vector, `(⍴L)` must be `(⍴R)[(⍳⍴⍴R)~X]`.

```
      W←'abcdefghijklmnopqrst'
      W←W,(⍳20)
      W←3 4 5⍴W,'ABCDEFGHIJKLMNOPQRST'
      W
 a  b  c  d  e
 f  g  h  i  j
 k  l  m  n  o
 p  q  r  s  t

 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20

 A  B  C  D  E
 F  G  H  I  J
 K  L  M  N  O
 P  Q  R  S  T

      ⍴W
3 4 5

      V←2 5⍴0 1 ¯1 2 ¯2 ¯3 ¯1 1 3 0 1 0 2 ¯1 3
      V
 0  1 ¯1  2 ¯2
¯3 ¯1  1  3  0
 1  0  2 ¯1  3

      V⌽[2]W
 a  g  r  n  o
 f  l  c  s  t
 k  q  h  d  e
 p  b  m  i  j

 6 17  8 19  5
11  2 13  4 10
16  7 18  9 15
 1 12  3 14 20

 F  B  M  S  T
 K  G  R  D  E
 P  L  C  I  J
 A  Q  H  N  O
```

***Selective Specification:*** Rotate with axis can be used for selective specification.

```
      Y←3 4⍴⍳12
      (1 ¯1 2 ¯2⌽[1]Y)←3 4⍴'ABCDEFGHIJKL'
      Y
IFGL
AJKD
EBCH
```

## \ Scan (from Backslash)

`Z←LO\ R` The *I*th item along the last axis is determined by the *LO*-reduction of `I↑[⍴⍴R]R`.

*LO*: Dyadic function

```
 ⍴Z   ←→ ⍴R
⍴⍴Z   ←→ ⍴⍴R
```

If the length of the last axis is greater than `0`, the result is determined by:

`(1↑[⍴⍴R]R),(LO/[X]2↑[⍴⍴R]R), ... ,(LO/R)`

```
      +\1 2 3 4 5
1 3 6 10 15

      +\(1 2)(3 4)(5 6)
 1 2  4 6  9 12
```

```
      ∨\0 0 1 1 0
0 0 1 1 1

      ,\'AB' 'CD' 'EF'
 AB ABCD ABCDEF

      ,\2 3⍴⍳6
 1  1 2  1 2 3
 4  4 5  4 5 6
```

If the length along the last axis is zero, the result is *R*.

## \[ ] ⍀[ ] Scan with Axis (from Backslash)

`Z←LO\[X] R` The `I`th item along the `X`th axis is determined by the `LO`-reduction of `I↑[X]R`.

`LO`: Dyadic function
`X`: Simple scalar or one-item vector, integer: `X∊⍳⍴⍴R`

Implicit argument: `⎕IO`

```
 ⍴Z   ←→  ⍴R
⍴⍴Z   ←→  ⍴⍴R
```

If the length of the last axis is greater than `0`, the result is determined by:

`(1↑[X]R),(LO/2↑[X]R), ... ,(LO/[X]R)`

```
      M←3 4⍴⍳12
      +\[1]M
 1  2  3  4
 6  8 10 12
15 18 21 24
```

```
      ,\[1]2 3⍴⍳6
   1     2     3
 1 4   2 5   3 6
```

```
      N←2 3 4⍴⍳24
      +\[1]N
 1  2  3  4
 5  6  7  8
 9 10 11 12

14 16 18 20
22 24 26 28
30 32 34 36

      +\[2]N
 1  2  3  4
 6  8 10 12
15 18 21 24

13 14 15 16
30 32 34 36
51 54 57 60
```

***Applied to First Axis:*** The symbol `⍀` is an alternative symbol for `\[1]`. However, if `⍀` is followed by an axis (`⍀[X]`), it is treated as `\[X]`.

```
      ×\[1]M
 1   2   3   4
 5  12  21  32
45 120 231 384
```

```
      ×⍀M
 1   2   3   4
 5  12  21  32
45 120 231 384
```

If the length along the `X`th axis is `0`, the result is `R`.

## ρ Shape

`Z←ρR`   Yields the size of each axis of `R`.

`Z`: Simple nonnegative integer vector.

```
 ρZ   ↔  ρρR
ρρZ   ↔  ,1
```

In a character array, blanks (within quotation marks) are items:

```
      A←1 2 3 'A' 'B' 4
      ρA
6
      ρρA
1
```

```
      B←'STAND UP'
      ρB
8
```

As the last example in the first column shows, applying ρ twice yields the *rank* of an array.

The high-order axis is the first item of the shape vector.

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

      ρC
3 4
      ρρC
2
```

```
      D←2 3 4ρ(⍳12),-⍳12
      D
  1   2   3   4
  5   6   7   8
  9  10  11  12

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

      ρD
2 3 4
      ρρD
3
```

```
      H←'TOM' 'ED' 'HANK'
      ≡H
2
      ρH
3
      ρ¨H
 3  2  4


      Q←('ELSIE' 'TOM') 'HANK' ('ED' 'BOB' 'KIM')
      ≡Q
3
      ρQ
3
      ρρQ
1
      ρ¨Q
 2  4  3
      ρ¨¨Q
  5  3              2  3  3
         ↑
```

These four items are empty because the items of `'HANK'` are scalars.

***Scalar Argument:*** The shape of a scalar is empty and its rank is 0 because scalars have no axes. Shape demonstrates the difference between a scalar and a one-item vector.

**Scalar**

```
      ρ'A'
(empty)
      ρρ'A'
0

      S←⊂2 3ρι6
      S
 1 2 3
 4 5 6
      ≡S
2
      ρS
(empty)
```

**One-Item Vector**

```
      ρ,'A'
1
      ρρ,'A'
1

      T←,⊂2 3ρι6
      T
 1 2 3
 4 5 6
      ≡T
2
      ρT
1
```

# - Subtract

`Z←L-R`    Subtracts `R` from `L`.

`L`, `R`, and `Z`: Numeric

*Scalar Function*

Subtract is the arithmetic subtraction function.

```
      5-3
2
      6-8 .2 4J3
¯2 5.8 2J¯3
```

```
      3J4-1J2
2J2
      ¯4 .5 0-¯2 1.2 1J2
¯2 ¯0.7 ¯1J¯2
```

## ↑ Take

`Z←L↑R` Selects subarrays from the beginning or end of the `I`th axis of `R`, according to whether `L[I]` is positive or negative.

`L`: Simple scalar or vector, integer

```
 ⍴Z  ↔  |,L
⍴⍴Z  ↔  ⍴,L
```

***Specifying the Amount to Take:*** If `L` is a scalar, it is treated as being a one-item vector; and if `R` is a scalar, it is treated as being an array of shape `(⍴L)⍴1`. Then:

For `L[I]>0`, take selects `L[I]` subarrays from the beginning of the `I`th axis of `R`.

For `L[I]<0`, take selects `|L[I]` subarrays from the end of the `I`th axis of `R`.

For `L[I]=0`, no items are selected, and the resulting shape has an `I`th axis of length 0.

```
      3↑34 12 73 53 41
34 12 73
```

```
      ¯3↑34 12 73 53 41
73 53 41
```

***Nonscalar Right Argument:*** For nonscalar `R`, `L` must have the same number of items as `R` has rank:

```
(⍴,L) = ⍴⍴R
```

```
      Y←4 5⍴'TRIADFIELDMOOSEDINER'
      Y
TRIAD
FIELD
MOOSE
DINER
      ¯2 3↑Y
MOO
DIN
      W←3 3 4⍴'BEATMYTHANTETONEMEANHEREUPONWEEKDOES'
      W
BEAT
MYTH
ANTE

TONE
MEAN
HERE

UPON
WEEK
DOES
```

```
      V←¯1 ¯2 2
```

(*means take the last plane, last two rows, first two columns*)

```
      Z←V↑W
      Z
WE
DO
      ρZ
1 2 2
```

***Overtake:*** If |L[I] is greater than the length of the Ith axis, the extra positions in the result are filled with the fill item (the prototype of R←→↑0ρ⊂↑R).

```
      5↑21 33 52
21 33 52 0 0
      5↑'RED'
RED
   ↑
   Two blank characters
      U←2 3ρι6
      U
1 2 3
4 5 6
      H←4 4↑U
      ρH
4 4
      H
1 2 3 0
4 5 6 0
0 0 0 0
0 0 0 0

      N←(1 2) (3 4)
      4↑N
 1 2  3 4  0 0  0 0
```

```
      ¯5↑21 33 52
0 0 21 33 52
      ¯5↑'RED'
  RED
↑
Two blank characters
      ¯6↑'A' 1 'B' 2
  A 1 B 2

      ¯6↑1 'A' 2 'B'
0 0 1 A 2 B

      3↑ι0
0 0 0

      2 3↑0 2ρ⊂0 0 0
 0 0 0  0 0 0  0 0 0
 0 0 0  0 0 0  0 0 0
```

The last two examples in the right column show the effect of take with an empty right argument. A nonempty left argument results in an overtake, using ↑R as the fill item. The result is not empty.

***Scalar Right Argument:*** For scalar R, L may have any length. The length of ,L determines the rank of the result.

```
      E←1↑2
      ρE
1
      E
2
      G←1 1 1↑2
      G
2
      ρG
1 1 1
```

```
      F←(ι0)↑2
      F
 2
      ρF
(empty)
      2 3↑2
 2 0 0
 0 0 0
```

***Effect on Depth:*** Take does not affect the depth of any selected item. The depth of the result is less than or equal to the depth of the argument, except when the right argument is a simple scalar.

```
      T←'T' 'TO' ('TOT' 'TOTE')
```

```
      J←1↑T
      J
T
      ≡J
1
```

```
      K←2↑T
      K
 T TO
      ≡K
2
```

Recall that the fill item is determined by the first item.

```
      S←8 ((6 5) (4 3))
      S
 8   6 5  4 3

      ≡S
3
      3↑S
 8   6 5  4 3   0
```

```
      Q←⌽S
      Q
  6 5  4 3   8
      3↑Q
  6 5  4 3   8   0 0  0 0
```

***Selective Specification:*** Take can be used for selective specification:

```
      P←'ABCDE'
      (2↑P)←1 2
      P
1 2 CDE
```

```
      KY←3 4⍴'ABCDEFGHIJKL'
      KY
ABCD
EFGH
IJKL
      (¯2 1↑KY)←1 2
      KY
A BCD
1 FGH
2 JKL
```

# ↑[ ] Take with Axis

`Z←L↑[X]R` Selects subarrays from the beginning or end of the `X[I]`th axis of `R`, according to whether `L[I]` is positive or negative.

`L`: Simple scalar or vector, integer
`R` and `Z`: Nonscalar array
`X`: Simple scalar or vector; nonnegative integers: `X∊⍳⍴⍴R`; or empty

Implicit argument: `⎕IO`

```
(⍴Z)[,X]  ←→ |,L
     ⍴⍴Z  ←→ ⍴⍴R
```

*Take with axis* is similar to *take* except that the subarrays are selected only from the axes indicated by `X`. The shape along axes not selected by `X` remains unchanged.

***Take with Axis Compared to Take:*** The following identity states the relationship between *take* and *take with axis*:

```
L↑R ←→ L↑[⍳⍴⍴R]R

      A←3 5⍴'GIANTSTORETRAIL'
      A
GIANT
STORE
TRAIL

      2↑[1]A
GIANT
STORE

      2 5↑A
GIANT
STORE

      ¯3↑[2]A
ANT
ORE
AIL

      3 ¯3↑A
ANT
ORE
AIL
```

***Overtake:*** If `|L[I]` is greater than the length of the `X[I]`th axis, the extra positions in the result are filled with the fill item. The fill item depends on the subarray selected:

```
L↑[X]R  ←→  ⊃[X](⊂L)↑¨⊂[X]R
```

```
      B←2 3⍴⍳6
      B
1 2 3
4 5 6
      3↑[1]B
1 2 3
4 5 6
0 0 0


      H←2 3⍴'ABCDEF'
      H
ABC
DEF
      Z←¯4↑[1]H
      Z


ABC
DEF
      ⍴Z
4 3
```

```
      C←2 3⍴1'A' 3 4 5 6
      C
1 A 3
4 5 6
      4↑[1]C
1 A 3
4 5 6
0   0
0   0
```

***Permitted Axes:*** Multiple axes indicated by `X` need not be in ascending order; however, no axis may be repeated. `L[I]` defines the number of subarrays to take from the `X[I]`th axis.

```
      K←3 3 4⍴'HEROSHEDDIMESODABOARPARTLAMBTOTODAMP'
      K
HERO
SHED
DIME

SODA
BOAR
PART

LAMB
TOTO
DAMP
```

```
      ¯1 3↑[1 3]K
LAM
TOT
DAM

      ¯1 3↑[3 1]K
O
D
E

A
R
T

B
O
P
```

***Effect on Depth:*** Take with axis does not affect the depth of any selected item. The depth of the result is less than or equal to the depth of the argument, except when the right argument is a simple scalar.

```
      T←'D' 'DO'('DON' 'DONE')'M' 'ME'('MEN' 'MENE')
```

```
      S←2 3⍴T

      S
 D DO    DON DONE
 M ME    MEN MENE

      ≡S
3
```

```
      H←2↑[2]S
      H
 D DO
 M ME

      ≡H
2
      J←1↑[1]S
      J
 D DO    DON DONE
      ≡J
3
```

Recall that the fill item is the type of the first item (prototype) of each subarray along the *X*th axis.

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

```
      M
 1   2 3   4 5   6 7
 8   9 1   2 3   4 5
      ⍴M
2 3

      3↑[1]M
 1   2 3   4 5   6 7
 8   9 1   2 3   4 5
 0   0 0   0 0   0 0
```

```
      T←1⌽[2]M
      T
 2 3   4 5   6 7   1
 9 1   2 3   4 5   8

      3↑[1]T
 2 3   4 5   6 7     1
 9 1   2 3   4 5     8
 0 0   0 0   0 0     0
```

***Selective Specification:*** Take with axis can be used for selective specification:

```
      U←3 4⍴'ABCDEFGHIJKL'
      U
ABCD
EFGH
IJKL

      (¯2↑[2]U)←3 2⍴⍳6
      U
AB 1 2
EF 3 4
IJ 5 6
```

---

## ⍉ Transpose (General)

`Z←L⍉R` Case 1: `L` selects all axes of `R`. Creates an array similar to `R` but with the axes permuted according to `L`.

Case 2: `L` includes repetitions of axes. Creates an array with two or more axes of `R` mapped into a single axis of `Z`, which is then a diagonal cross section of `R`.

`L`: Simple scalar or vector, nonnegative integer

Implicit Argument: `⎕IO`

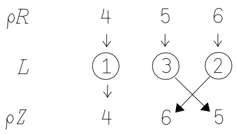
**Case 1**

```
 ⍴Z   ←→ (⍴R)[⍋L]
⍴⍴Z   ←→ ⍴⍴R
```

**Case 2**

```
I⊃⍴Z   ←→ ⌊/(L=I)/⍴R
  (for each I∊⍳⍴⍴Z)

 ⍴⍴Z   ←→ ,+/(L⍳L)=⍳⍴L
```

`L` ***Selects All Axes of*** `R`: All axes of `R` must be represented in `L`: `∧/(⍳⍴⍴R)∊L`. The axes of `R` map by position to axes of `Z` according to `L`. The diagram below shows the mapping of axes for `1 3 2⍉4 5 6⍴⍳120`:



The `I`th axis of `R` becomes the `L[I]`th axis of `Z`.

```
      A←2 3 4ρ'BEARLYNXDUCKPONYBIRDOXEN'
      A
BEAR
LYNX
DUCK

PONY
BIRD
OXEN
      ρA
2 3 4

      Z←1 3 2⍉A
      ρZ
2 4 3

      Z
BLD
EYU
ANC
RXK

PBO
OIX
NRE
YDN

      W←2 1 3⍉A
      ρW
3 2 4
      W
BEAR
PONY

LYNX
BIRD

DUCK
OXEN
```

```
      Y←3 1 2⍉A
      ⍴Y
3 4 2
      Y
BP
EO
AN
RY

LB
YI
NR
XD
DO
UX
CE
KN
```

⍉ Transpose (reversed axes), page 256, reverses the order of the axes for the transposition:

```
      ⍉R ←→ (⌽⍳⍴⍴R)⍉R
```

***Diagonal Cross Section of*** *R*: When there are repetitions in *L*, a diagonal cross section of *R* is selected. *L* must be constructed such that `∧/(⍳⌈/0,L)∊L`. For a matrix, `1 1⍉R` selects those items whose row and column indexes are the same and creates a vector from those items.

```
      B←4 4⍴⍳16
      B
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

      1 1⍉B
1 6 11 16

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

      1 1⍉C
1 6 11
```

```
      D←'ONE' 'FOR' 'ALL' 'HEAD' 'TO'
      D←3 3⍴D,'TOE' 'READY' 'SET' 'GO'
      D
 ONE   FOR ALL
 HEAD  TO  TOE
 READY SET GO

      V←1 1⍉D
      V
 ONE TO GO
      ⍴V
3
      ≡V
2
```

For higher rank arrays, the result is determined first by a selection of items whose indexes are the same in the duplicated axes indicated by `L`. For example, `2 1 2⍉R` selects all items whose first and third indexes are the same: `R[1;1;1]`, `R[1;2;1]`, and so forth. The selected items are then transposed by `((L⍳L)=⍳⍴L)/L`. The transpose for items selected for the `2 1 2⍉R`, for example, is `2 1`.

```
      H←2 3 4⍴'ABCDEFGHIJKL',⍳12
      H
A  B  C  D
E  F  G  H
I  J  K  L

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

      1 1 1⍉H
A 6

      1 1 2⍉H
A B C D
5 6 7 8

      2 2 1⍉H
A 5
B 6
C 7
D 8
```

```
      1 2 1⍉H
A E  I
2 6 10

      2 1 2⍉H
A  2
E  6
I 10

      1 2 2⍉H
A F  K
1 6 11

      2 1 1⍉H
A  1
F  6
K 11
```

***Effect of Index Origin:*** The index origin affects permissible values of `L`. For either origin: `∧/(⍳⍴⍴R)∊L`.

```
      ⎕IO←0
      K←3 2 4⍴⍳24
      K
 0  1  2  3
 4  5  6  7

 8  9 10 11
12 13 14 15

16 17 18 19
20 21 22 23
```

```
      1 0 2⍉K
 0  1  2  3
 8  9 10 11
16 17 18 19

 4  5  6  7
12 13 14 15
20 21 22 23
```

***Selective Specification:*** Either case of transpose can be used for selective specification.

```
      ⎕IO←1
      P←3 3⍴⍳9
      (1 1⍉P)←10 20 30
      P
10  2  3
 4 20  6
 7  8 30
```

## ⍉ Transpose (Reversed Axes)

```
Z←⍉R      Creates an array similar to R but with the order
          of the axes of R reversed.

 ⍴Z   ←→  ⌽⍴R
⍴⍴Z   ←→  ⍴⍴R
```

```
      A←4 3⍴'RAMONEATENET'
      A
RAM
ONE
ATE
NET
      ⍴A
4 3
      Z←⍉A
      Z
ROAN
ANTE
MEET

      ⍴Z
3 4

      B←2 3⍴(1 1)(1 2)(1 3)(2 1)(2 2)(2 3)
      B
 1 1  1 2  1 3
 2 1  2 2  2 3

      ⍴B
2 3

      ≡B
2

      X←⍉B
      X
 1 1  2 1
 1 2  2 2
 1 3  2 3

      ⍴X
3 2
```

```
      C←2 3 4⍴⍳24
      C
 1  2  3  4
 5  6  7  8
 9 10 11 12

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

      W←⍉C
      W
 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23

 4 16
 8 20
12 24

      ⍴W
4 3 2
```

***Selective Specification:*** Transpose with reversed axes can be used in selective specification:

```
      R←3 3⍴'STYPIEANT'
      R
STY
PIE
ANT
```

```
      (⍉R)←3 3⍴⍳9
      R
1 4 7
2 5 8
3 6 9
```

## ~ Without

`Z←L~R`    Yields the items in `L` that do not occur in `R`.

`L`: Scalar or vector
`Z`: Vector

Implicit argument: `⎕CT`

```
 ⍴Z   ←→  Depends on the contents of L and R
⍴⍴Z   ←→  ,1
```

The following identity holds:

```
   L~R ←→ (~L∊R)/L
```

```
      1 2 3 4 5~2 3 4    │        'RHYME'~'MYTH'
1 5                      │  RE
```

***Effect with Nested Arrays:*** An item of `L` is included in `Z` if an exact match (within the comparison tolerance) in structure and data does not exist in `R`.

```
      'GO' 'TO' 'IT'~'GOTO' 'IT'
 GO TO

      Z←4 5 (⍳0) 6 7~9 5 3 7
      Z
 4    6
      ⍴Z
3

      W←4 5 (⍳0) 6 7~9 5 3 7(⍳0)
      W
4 6
      ⍴W
2
```

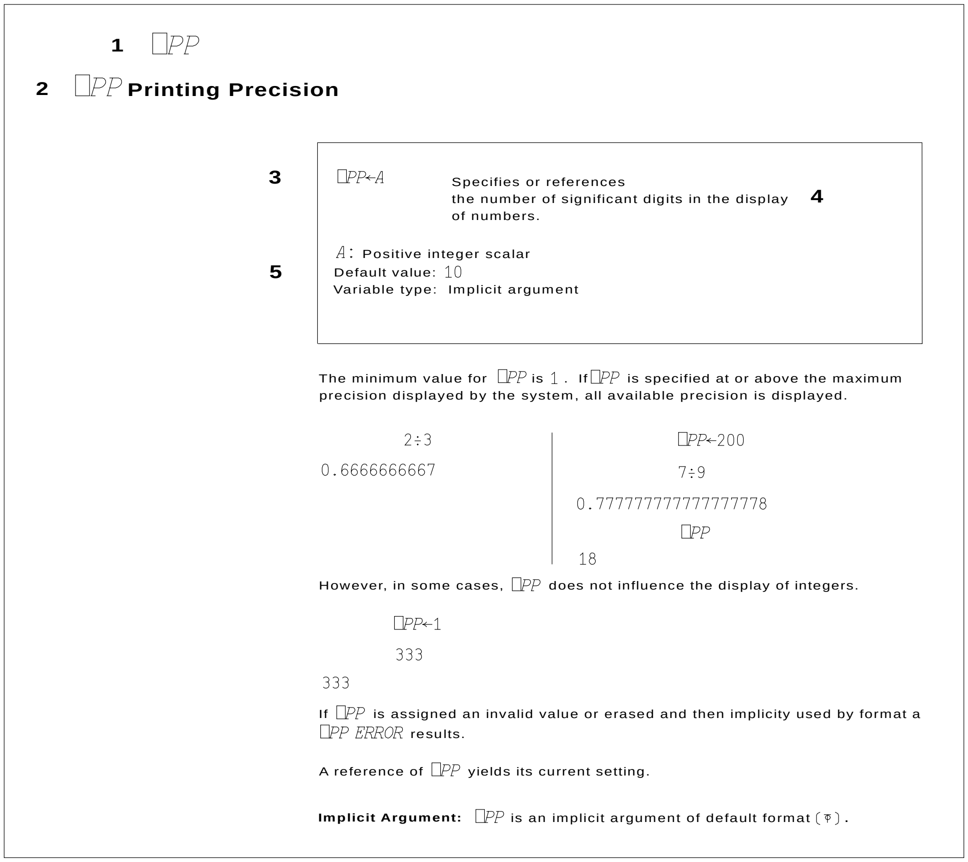
***Intersection of Two Vectors:*** The intersection of two vectors `L` and `R` (including any replication in `L`) may be obtained by the expression `L~L~R`.

```
      3 1 4 1 5 5~3 1 4 1 5 5~4 2 5 2 6
4 5 5
```

# Chapter 6. System Functions and Variables

This chapter describes all system functions and variables alphabetically. Each system function and variable description consists of a summary and several detailed sections. The organization of a system function description is similar to that of the primitive functions.

Figure 29 shows a sample page for a system variable description. The information in the summary at the top of the page is somewhat different from that in the summary for system functions and is discussed following the figure. The callouts on the figure correspond to that discussion.



⎕PP

**⎕PP Printing Precision**

⎕PP←A    Specifies or references the number of significant digits in the display of numbers.

A: Positive integer scalar
Default value: 10
Variable type: Implicit argument

The minimum value for ⎕PP is 1. If ⎕PP is specified at or above the maximum precision displayed by the system, all available precision is displayed.

```
      2÷3                         ⎕PP←200
0.6666666667                      7÷9
                        0.777777777777777778
                                  ⎕PP
                        18
```

However, in some cases, ⎕PP does not influence the display of integers.

```
      ⎕PP←1
      333
333
```

If ⎕PP is assigned an invalid value or erased and then implicity used by format a ⎕PP ERROR results.

A reference of ⎕PP yields its current setting.

**Implicit Argument:** ⎕PP is an implicit argument of default format (⍕).

*Figure 29. Sample Page of System Variables*

1. Variable name.
2. Variable name and description as it appears in the table of contents.
3. Variable syntax. If specifying the variable has an effect on its value, a specification is shown: *name*←A. If specifying or localizing the variable has no effect, only the name is shown.
4. Summary definition of the variable.

5. Properties of the variable:
   - Data. If specifying the variable has an effect, the type of data that is meaningful for the variable is listed.
   - Default value. The value the variable has in a CLEAR WS or the value it takes if you have not specified it. Note that the values you specify for ⎕NLT, ⎕PW, and ⎕TZ persist over a )CLEAR or a )LOAD.
   - Variable type. A classification of the variable by its general characteristics. Any of four characteristics of the variable can be highlighted.

     *Session variable*, if the variable is one of the three listed above as persisting over a )CLEAR or a )LOAD.

     *Debug variable*, if the variable is assigned by the system when an execution error occurs.

     *Implicit argument*, if the variable is used by a primitive function during the course of its execution.

     *Localizing or assigning a value has no effect* if the variable is respecified by the system.

Figure 30 displays the APL2 system functions and system variables.

*Figure 30 (Page 1 of 2). APL2 System Functions and System Variables*

| | | |
|---|---|---|
| **Control and Execution**<br>These functions and variables control execution and session I/O. | ⎕ | Evaluated Input/Output |
| | ⍞ | Character Input/Output |
| | ⎕DL | Delay |
| | ⎕EX | Expunge |
| | ⎕LX | Latent Expression |
| | ⎕NA | Name Association |
| | ⎕NLT | National Language Translation |
| | ⎕PR | Prompt Replacement |
| | ⎕PW | Printing Width |
| **Event Handling and Debugging**<br>These functions and variables provide for error handling and testing of conditions during an error situation. | ⎕EA | Execute Alternate |
| | ⎕EC | Execute Controlled |
| | ⎕EM | Event Message |
| | ⎕ES | Event Simulation |
| | ⎕ET | Event Type |
| | ⎕L | Left Argument |
| | ⎕LC | Line Counter |
| | ⎕R | Right Argument |
| **Workspace and System Information**<br>These functions and variables provide information about APL2 variables, functions and operators.<br>They provide general system information such as system time, user load, and available working space. | ⎕AF | Atomic Function |
| | ⎕AI | Account Information |
| | ⎕AT | Attributes |
| | ⎕AV | Atomic Vector |
| | ⎕NC | Name Class |
| | ⎕NL | Name List |
| | ⎕TC | Terminal Control Characters |
| | ⎕TS | Time Stamp |
| | ⎕TZ | Time Zone |
| | ⎕UCS | Universal Character Set |
| | ⎕UL | User Load |
| | ⎕WA | Workspace Available |

*Figure 30 (Page 2 of 2). APL2 System Functions and System Variables*

| | | |
|---|---|---|
| **Implicit Arguments**<br>These variables allow the user to alter the results returned by certain APL2 primitive functions. | ⎕*CT* | Comparison Tolerance |
| | ⎕*FC* | Format Control |
| | ⎕*IO* | Index Origin |
| | ⎕*PP* | Printing Precision |
| | ⎕*RL* | Random Link |
| **Transforming Data and Expressions**<br>These functions allow a user to convert character data to or from executable form. | ⎕*CR* | Character Representation |
| | ⎕*FX* | Fix |
| | ⎕*TF* | Transfer Form |
| **Sharing**<br>These functions and variables allow a user to share variables with other APL2 users and other auxiliary processors. | ⎕*SVC* | Shared Variable Control |
| | ⎕*SVE* | Shared Variable Event |
| | ⎕*SVO* | Shared Variable Offer |
| | ⎕*SVQ* | Shared Variable Query |
| | ⎕*SVR* | Shared Variable Retraction |
| | ⎕*SVS* | Shared Variable State |

## ⎕ Evaluated Input/Output

| | |
|---|---|
| ⎕ | Input: ⎕ presents a prompt for input (⎕:). The value of the expression entered replaces the quad, and the resulting expression is evaluated. |
| ⎕←*A* | Output: ⎕ displays the value of the expression to the right of the specification arrow. |

*A*: Any valid expression
Default value: None

The behavior of ⎕ is dependent on whether data is assigned to it (output), or it is referenced (input).

### Assignment

When ⎕ appears to the left of the specification arrow (the left arrow), the value of the expression to the right of the arrow is displayed.

```
      ⎕←4+6×5
34
```

Assignment of ⎕ allows the display of interim results, or the display of a variable's value in the expression in which it is specified.

```
      ρ⎕←ι3       |      A←4+⎕←5+6   |      ⎕←B←2÷3
1 2 3             |11                |0.6666666667
3                 |      A           |      B
                  |15                |0.6666666667
```

### Reference

When ⎕ appears and is not on the left of the specification arrow, a prompt (⎕:) is displayed, and the system waits for input under the control of default error or interrupt handling.

After the requested input is supplied and evaluated (by producing an array), error and interrupt handling reverts to whatever it was prior to the reference of ⎕.

```
      4+⎕×5                  |      A←4+⎕×5
⎕:                           |⎕:
      11                     |      ι3
59                           |      A
                             |9 14 19
```

***Error in Expression:*** If the evaluation of the expression in the response to `⎕:` generates an error, the error appears as it does in immediate execution mode.

When the expression in response to the prompt is in error, the prompt ⎕ is presented again. When, however, the error occurs after the value of the expression is substituted for ⎕, execution of the expression is suspended, no prompt is displayed, and the expression appears in the state indicator. Clearing the state indicator is discussed in “Clearing the State Indicator” on page 357.

```
      1+⎕
⎕:
      2 3×4 5 6
LENGTH ERROR
      2 3×4 5 6
      ^   ^
⎕:
      'A'
DOMAIN ERROR
      1+⎕
      ^^
      )SIS
*   1+⎕
    ^^
```

***Multiple Quads:*** More than one reference of quad can occur in an expression. The usual evaluation rules apply (see “Evaluating Expressions” on page 32).

```
      ⎕-⎕
⎕:
      8
⎕:
      3
¯5
```

***Escape:*** If the response to `⎕:` is the escape (→), execution ends and no result is returned.

```
      4+⎕+5+6
⎕:
      →
```

```
      FN[⎕]
⎕:
      →
```

A situation such as that shown in the right column occurs if you forget to enter a beginning del (∇) when trying to display a function or operator definition.

A system command can be entered when a `⎕:` is displayed. The system command and the system's response are not treated as responses to `⎕:`.

The following system commands end execution of the expression that referenced ⎕:

- `)CLEAR`
- `)LOAD`
- `)RESET`
- Some `)HOST` commands
- `)OFF`

For example:

```
      4+⎕+5+6
⎕:
      )WSID WORKOUT
WAS CLEAR WS
⎕:
      )RESET
```

# ⍞ Character Input/Output

| | |
|---|---|
| ⍞ | Input: The system waits for a response and treats the input as a character string. |
| ⍞←*A* | Output: Displays the value of the expression to the right of the specification arrow. The position of the cursor or print element carrier after output or before input depends on the situation, as described below. |

*A*: Any valid expression
Default value: None

The behavior of ⍞ is dependent on whether either data is assigned to it (output) or it is referenced (input).

## Assignment

When ⍞ appears to the left of the specification arrow (the left arrow), the value of the expression to the right of the arrow is displayed. The normal ending new line character is always suppressed.

```
      2 3 4,⍞←'A HA '
A HA 2 3 4 A HA

      ∇ F X
[1]   ⍞←'X '
[2]   ⍞←'IS'
[3]   ⍞←' '
[4]   ⍞←X
[5]   ∇

      F 13
X IS 13
```

```
      ∇ G
[1]   ⍞←2 3⍴⍳6
[2]   ⍞←' IS A MATRIX'
[3]   ∇
      G
  1 2 3
  4 5 6 IS A MATRIX
```

Successive assignments of vectors to ⍞ without any other intervening session input or output can cause attempts to display the arrays on the same output line. Because of this, the sum of the lengths of the vectors should be less than the width of the session display line. Otherwise, unpredictable results can occur.

## Reference

When ⍞ is referenced, session input is requested. The input is returned as a character vector.

```
      RESULT←⍞
WHAT IS 3+4?          Typed by the user.
      ⍴RESULT
12
      RESULT
WHAT IS 3+4?
```

Quotation marks entered on a reference to ⍞ are characters. For example:

```
      X←⍞
'DON''T STOP'
      X
'DON''T STOP'

      ⍴X
13
```

***Prompts and Responses:*** A reference to ⍞ preceded by an assignment without any intervening session input or output creates a *prompt/response interaction*. The last (or only) row of the assignment is called the *prompt*, and the result of the reference is called the *response*. The response is a vector composed of:

- A transformation of the unchanged characters in the prompt, as determined by the prompt replacement system variable ⎕PR (see page 316).
- Session input, including changed characters in the prompt.

For example:

```
      ⎕PR←' '
      ∇ Z←XPRMPT
[1]   ⍞←'SUPPLY X: '
[2]   Z←⍞
[3]   ∇

      RESULT←XPRMPT
SUPPLY X: 19                19 is entered by the user.
      RESULT
          19         Result includes blanks that replace
      ⍴RESULT        the prompt of line 1.
12
```

The sum of the lengths of the prompt and the expected session input should be less than the width of the display area; otherwise, the result may be unpredictable.

On most devices, the prompt can be changed before it is reentered. If ⎕*PR* is assigned the empty vector (`''`), the result of the expression that includes ⍞ is the vector in the display area when it is returned to the system.

```
      ∇Z←FN2
[1]   ⍞←'CHANGE THE PROMPT: '
[2]   Z←⍞
[3]   ∇

      ⎕PR←''
      RESULT←FN2
CHANGE THE ENTRY   45             The word PROMPT is
      ⍴RESULT                     changed to ENTRY and 45 entered.
21
      RESULT
CHANGE THE ENTRY   45
```

In contrast, if ⎕*PR* is not `''`, characters not changed are replaced with ⎕*PR*. However, anything entered in response to ⍞ is not replaced by ⎕*PR*. No replacement occurs even if the response is typed in the prompt display area.

```
      ⎕PR←'*'
      FN2
CHANGE THE PROMPT: _
           ENTRY:  45

CHANGE THE ENTRY:  45
***********ENTRY:  45

      ⎕PR←'*'
      FN2
CHANGE THE ENTRY:  45
***********ENTRY:  45
```

***Interrupting Quote-Quad Input:*** If an interrupt is signaled while the system is waiting for input to a reference of ⍞, the *INTERRUPT* message is displayed and execution is suspended. If execution is resumed (by →⍳0), the result of the expression does not include the prompt.

```
      RESULT←XPRMPT
SUPPLY X:(interrupt signaled)
INTERRUPT
XPRMPT[2]  Z←⍞
             ^
      →⍳0
19                              Cursor or carrier waits at left
      ⍴RESULT                   margin for input.
2
```

## ⎕AF Atomic Function

`Z←⎕AF R` Maps integers to characters and characters to integers.

`R` and `Z`: A simple numeric integer array or a simple character array

```
ρZ ←→ ρR
ρρZ ←→ ρρR
```

Integers in `R` must be nonnegative and less than `2*31`.

`⎕AF R` is like `⎕AVιR` or `⎕AV[R]`, except that it is origin independent (always uses an origin of `0` value) and works on all characters, including those not in `⎕AV`.

`⎕AF` depends on the current internal encoding of data, which can vary among platforms, as well as at different times on the same platform, depending on how the data has been created or manipulated. See "`⎕UCS` Universal Character Set" on page 342 for a platform-independent character mapping.

## ⎕AI Account Information

`⎕AI` Provides user identification and compute, connect, and user response times in milliseconds.

Variable type: Localizing or specifying `⎕AI` has no effect.

`⎕AI` is a four-item numeric vector that provides the following information:

| | |
|---|---|
| `⎕AI[1]` | User identification |
| `⎕AI[2]` | Compute time |
| `⎕AI[3]` | Connect time |
| `⎕AI[4]` | User response time |

All times are in milliseconds and are cumulative during the APL2 session.

```
      ⎕AI
1001 185 53942 30029
```

## ⎕AT Attributes

`Z←L ⎕AT R` Returns an attribute vector selected by the integer specified in *L* for each object named in *R*.

*L*: Integer scalar
*R*: Simple character scalar, vector, or matrix
*Z*: Integer vector or matrix

```
 ρZ ←→ (¯1↓ρR),(3 7 4 2)[L]
ρρZ ←→ ,1⌈ρρR
```

Each row of *R* is interpreted as a constructed name. Each row of *Z* is an attribute vector specified by the integer *L* for the corresponding name in *R*.

The items in an attribute vector are described for the various values of *L* as shown in Figure 31.

*Figure 31 (Page 1 of 2). Description of the Attribute Vector for Various Values of L*

| **Value of L** | **Contents of Attribute Vector** | **Description of Each Item** | **How Reported** |
|---|---|---|---|
| 1 | Valences (length 3) | [1] Explicit result<br>[2] Function valence<br>[3] Operator valence | Z[1] is 1, if the object has an explicit result or is a variable. Z[1] is 0, if otherwise.<br>Z[2] is 0, if the object is a niladic function or not a function. Z[2] is 1, if the object is a monadic function. Z[2] is 2, if the object is an ambivalent function.<br>Z[3] is 0, if the object is not an operator. Z[3] is 1, if the object is a monadic operator. Z[3] is 2, if the object is a dyadic operator. |
| 2 | Fix time, which is the time the definition of the corresponding operation named in *R* was last updated (length 7) | [1] Year<br>[2] Month<br>[3] Day<br>[4] Hour<br>[5] Minute<br>[6] Second<br>[7] Millisecond | Digits are shown for each item.<br>If *R* is not a defined function or operator, the corresponding row of *Z* is all 0's (7ρ0). |

*Figure 31 (Page 2 of 2). Description of the Attribute Vector for Various Values of L*

| Value of L | Contents of Attribute Vector | Description of Each Item | How Reported |
|---|---|---|---|
| 3 | Execution properties (length 4) | [1] Nondisplayable<br>[2] Nonsuspendable<br>[3] Ignores weak interrupts<br>[4] Converts non-resource errors to *DOMAIN ERROR* | A 1 indicates that the corresponding property is set; a 0 indicates that it is not set.<br>The execution properties for a variable are 0's (4⍴0).<br>The execution properties for primitive and system functions are 1 1 1 0. (Parameter substitution can cause a primitive function to have a constructed name.) |
| 4 | Object Size (length 2) | [1] Bytes CDR requires<br>[2] Bytes data portion of CDR requires | CDR is the common data representation of APL2 objects used for shared variables. It consists of structure information and data.<br>Object size for a function or operator is reported as 0 0. |

**Example with** L←1:

```
      ⎕FX 'TOTAL R' '⎕←''TOTAL IS'',+/R'
TOTAL

      1 ⎕AT 'TOTAL'
0 1 0

      ⎕FX 'Z←TOTAL R' 'Z←+/R'
TOTAL

      1 ⎕AT 'TOTAL'
1 1 0

      ANSWER←TOTAL 1 9 3
      1 ⎕AT 2 6⍴'TOTAL ANSWER'
1 1 0
1 0 0
```

**Example with** L←2:

```
      2 ⎕AT 'TOTAL'
1991 12 19 17 38 18 286
```

**Example with** L←3:

```
      3 ⎕AT 'TOTAL'
0 0 0 0

      1 0 0 0 ⎕FX ⎕CR 'TOTAL'
TOTAL

      3 ⎕AT 'TOTAL'
1 0 0 0

      3 ⎕AT '⎕FX'
1 1 1 0
```

**Example with** L←4:

```
      4 ⎕AT 'TOTAL'
0 0

      VARIABLE←10 20 30
      4 ⎕AT 'VARIABLE'
19 3
```

## ⎕AV Atomic Vector

⎕AV Contains 256 characters of the defined character set.

Variable type: localizing or specifying ⎕AV has no effect.

⎕AV is a simple character vector. The results of displaying or printing certain of its items can depend on the type of display device or printer being used.

APL2 supports `2*31` different characters. ⎕AV is a selection of 256 commonly-used characters, including the characters primitive to the APL2 language. The ordering of ⎕AV is selected to match the principle character set of the platform (ASCII or EBCDIC).

See also Appendix A, “The APL2 Character Set” on page 470.

⎕AV determines the order in which objects are displayed as a result of the system commands `)NMS`, `)OPS`, `)FNS`, and `)VARS`.

**Note:** Some characters are terminal control characters and can cause unpredictable results when sent to certain devices.

## ⎕*CR* Character Representation

| | |
|---|---|
| *Z*←⎕*CR R* | Returns the character representation of the displayable defined function or defined operator named in *R*. |

*R*: Simple character scalar or vector
*Z*: Simple character matrix

*R* is the name of one defined operation.

The first row of *Z* is the function or operator header, as described in “Header” on page 347.

Each remaining row of *Z* is a line of the function or operator. The rows contain no unnecessary blanks, except for trailing blanks that pad the row and the blanks in comments (including those immediately preceding the ⍝). Trailing blanks in a comment line may or may not be included, depending on the length of the other rows.

```
      ∇Z←TOTAL R
[1]   Z←+/R
[2]   ∇

      ⎕CR 'TOTAL'
Z←TOTAL R
Z←+/R
```

The character representation of a defined function or operator may contain entirely blank lines. An entirely blank row represents an empty expression in the function. However, the last column of a character representation is not entirely blank.

If *R* is a variable name, the name of a nondisplayable defined function or operator, the name of an external variable, function or operator, or an illegal APL name, the result is an empty matrix.

```
      A←89 34 4
      Z←⎕CR 'A'
      Z
      ⍴Z
0 0
      1 0 0 0 ⎕FX 'Z←TOTAL R' 'Z←+/R'


TOTAL
      Z←⎕CR 'TOTAL'
      Z
      ⍴Z
0 0
```

## ⎕CT Comparison Tolerance

`⎕CT←A` Contains the quantity used by some primitive functions to determine equality.

`A`: Simple, real scalar greater than or equal to 0, but less than 1
Default value: `1E¯13`
Variable Type: Implicit argument

Real numbers `L` and `R` are considered equal if:

`(|L-R)` is less than or equal to `⎕CT×(|L)⌈|R`.

```
      ⎕CT
1E¯13
      L←466.7
      R←466.6999
      ⎕CT×(|L)⌈|R
4.667E¯11
      |L-R
0.00009999999997
      L=R
0
```

```
      ⎕CT
1E¯13
      L←466.7
      R←466.69999999999
      ⎕CT×(|L)⌈|R
4.667E¯11
      |L-R
1.000444172E¯11
      L=R
1
```

Complex numbers `L` and `R` are considered *equal* if both their real and imaginary parts are equal. For comparison purposes, a nonreal number is considered to be *real* if the greater of the absolute values of its imaginary part and the tangent of the angle is much less than `⎕CT`.

Computations of `⎕CT` are approximated for efficiency. For this reason, using values of `⎕CT>1E¯9` is discouraged.

If `⎕CT` is assigned an invalid value or erased and then implicitly used by a primitive function, a `⎕CT ERROR` results.

A reference of `⎕CT` yields its current value.

System tolerance, which cannot be set, is different from `⎕CT`. See "System Tolerance" on page 59.

No number is within `⎕CT` of zero.

***Primitive Functions That Use ⎕CT:*** ⎕CT is an implicit argument of the following primitive functions:

| | | |
|---|---|---|
| Ceiling | ⌈R | page 79 |
| Find | L⍷R | page 129 |
| Floor | ⌊R | page 133 |
| Index of | L⍳R | page 162 |
| Match | L≡R | page 91 |
| Member | L∊R | page 181 |
| Relational functions | L<R | page 219 |
| | L≤R | |
| | L=R | |
| | L≥R | |
| | L>R | |
| | L≠R | |
| Residue | L\|R | page 227 |
| Without | L~R | page 258 |

## ⎕DL Delay

*Z*←⎕*DL R*   Causes a pause of approximately *R* seconds.

*R*: Scalar nonnegative number
*Z*: Scalar real number

*Z* contains the actual number of seconds in the pause.  The actual number of seconds varies from execution to execution.

The pause can be interrupted by signaling an interrupt.

```
      ⎕DL 2
2.010658
      ⎕DL 2
2.010006
```

```
      ⎕DL 4
4.008428
      ⎕DL 4
4.006799
```

## ⎕EA Execute Alternate

| | |
|---|---|
| Z←L ⎕EA R | Executes R. If R fails or is interrupted, executes L. |
| L and R: Simple character vector or scalar | |

The expression represented by R is executed. If an error occurs during its execution or R is interrupted (interrupt signaled), ⎕EM and ⎕ET are set, execution of R is abandoned without an error message, and the expression represented by L is executed. Execution of L is subject to normal error handling.

```
      'ι3' ⎕EA 'ι4.5'
1 2 3

      'ι3.3' ⎕EA 'ι4.5'
DOMAIN ERROR
      ι3.3
      ^
      'ι3.3' ⎕EA 'ι4.5'
      ^      ^
```

```
      'ι3' ⎕EA 'ι4'
1 2 3 4

      '''ERR''' ⎕EA 'ι4.5'
ERR
```

***Effect of Assigning Result:*** If R does not return an explicit result, the attempt to assign the result to Z can generate an immediate VALUE ERROR or may generate an error that causes L to be executed.

```
      Z←'3×2' ⎕EA '→0'
      Z
6

      Z←'3×2' ⎕EA 'DESCRIBE'
THE XYZ WORKSPACE
PROVIDES SEVERAL . . .
VALUE ERROR
      Z←'3×2' ⎕EA 'DESCRIBE'
       ^      ^
```

If *L* is executed and does not return an explicit result, a *SYNTAX ERROR* results.

```
      Z←'DESCRIBE' ⎕EA '!¯3'
THE XYZ WORKSPACE
PROVIDES SEVERAL . . .
VALUE ERROR
      Z←'DESCRIBE' ⎕EA '!¯3'
       ^           ^

      Z←'→0' ⎕EA '!¯3'
SYNTAX ERROR
      →0
      ^
      Z←'→0' ⎕EA '!¯3'
       ^     ^
```

Assigning the results of *R* and *L* separately prevents this problem.

```
      'Z←⍳8' ⎕EA 'Z←⍳3'
      Z
1 2 3

      'Z←⍳8' ⎕EA 'Z←!¯3'
      Z
1 2 3 4 5 6 7 8

      '→0' ⎕EA 'Z←!¯3'

      'DESCRIBE' ⎕EA 'Z←!¯3'
THE XYZ WORKSPACE
PROVIDES SEVERAL . . .
```

***Defined Function Invoked by*** *R*: If *R* calls defined function *F*, the statements executed by *F* are also under the control of the error trap. In particular, *R* can call a long running function, and *L* can be a recovery function.

## ⎕EC Execute Controlled

| | |
|---|---|
| Z←⎕EC R | Executes R. Returns a return code, ⎕ET, and the expression result. |

R: Simple character vector or scalar

The expression represented by R is executed. The first item of the result is a return code as follows:

| | |
|---|---|
| 0 | Error (2+'A') |
| 1 | Expression with a result which would display (2+3) |
| 2 | Expression with a result which would not display (A←2+3) |
| 3 | Expression with no explicit result (F X where F has no result) |
| 4 | Branch to a line (→3) |
| 5 | Branch escape (→) |

The second item of the result is the value ⎕ET would have. This is 0 0 unless an error occurs. The current value of ⎕ET is not affected.

The third item is the result of the expression if the return code is 1 or 2; 0 0⍴0 if the return code is 3 or 5; the argument to branch if return code is 4; and ⎕EM if the return code is 0.

Stops (S∆...) are ignored when executing under ⎕EC. Errors or keyboard interrupts are trapped and produce a zero return code (↑Z), a nonzero ⎕ET (↑1↓Z), and a ⎕EM (↑2↓Z) that details the event. This implies that settings of S∆ are ignored. Quad input is permitted if it returns a value. For example, branch escape (→) and )CLEAR are not permitted.

```
      ⎕EC '2+3'
 1   0 0   5
      (RC ET R)←⎕EC '→'
      RC
5
      (RC ET R)←⎕EC '⍳4.5'
      RC
0
      ET
5 4
      R
DOMAIN ERROR
      ⍳4.5
      ^
```

## ⎕*EM* Event Message

⎕*EM*   Text of the error or event message associated with the first line of the state indicator.

Default value: `3 0⍴' '`
Variable type: Debug variable; specifying or localizing ⎕*EM* has no effect.

When execution of an expression generates an error message, ⎕*EM* contains all lines of the message as displayed, even when the left argument of ⎕*ES* (event simulate) was used to specify the first row of ⎕*EM* as part of event handling. (APL2 error messages are described in Chapter 11, “Interpreter Messages” on page 461.)

The following example shows the message and the value of ⎕*EM* for a *LENGTH ERROR*.

```
      2+3 4 5=6 3
LENGTH ERROR
      2+3 4 5=6 3
       ^     ^

      ⍴⎕EM
3 17
      ⎕EM
LENGTH ERROR
      2+3 4 5=6 3
       ^     ^
```

If there is not enough room in the workspace to form ⎕*EM* at the time of the error, ⎕*EM* is a matrix of shape `3 0`, but the event type code ⎕*ET* is not affected.

If there is not enough room in the workspace to suspend the statement in error, *WS FULL* is reported and ⎕*EM* is set to a matrix of shape `3 0`. ⎕*EM* is automatically local to a function called by a line entered in immediate execution.

⎕*EM* ***and the State Indicator***: ⎕*EM* contains the event message associated with the top line of the state indicator. As the stack is cleared (with → or )*RESET* *n*), ⎕*EM* is reset to the event message associated with the current top line of the state indicator.

If the state indicator is clear, ⎕*EM* is set to `3 0⍴' '`.

## ⎕ES Event Simulate (with either Error Message or Event Type)

| | |
|---|---|
| ⎕ES R | Simulates an event and returns an error report for the event based on the value of R. |

R: Simple character scalar or vector; simple two-item vector of integers between ¯32767 and 32767; or empty vector.

When ⎕ES is executed from within a defined function or operator and R is not empty, the event action is generated as though the function were primitive or locked (by ⍫ or by setting all execution properties using ⎕FX to 1). Suspension occurs at the calling point, not within the defined operation. The message displayed and the setting of ⎕ET and ⎕EM depend on the value of R, as described below.

***When R Is a Character Scalar or Vector***: Normal APL2 error handling is initiated. R is displayed as the error message and set in ⎕EM (error message). ⎕ET (event type) is set to `0 1`.

```
      ∇Z←EXPO A
[1]   ⎕ES(0=A)/'ZERO INVALID'
[2]   Z←*A
[3]   ∇
      EXPO 3
20.08553692

      EXPO 0
ZERO INVALID
      EXPO 0
      ^

      ⎕EM
ZERO INVALID
      EXPO 0
      ^

      ⎕ET
0 1
```

**When *R* Is an Error Code Defined for ⎕ET:** The error message associated with that event type code is reported as the first row of the message matrix in the current national language.

```
      ∇Z←FACTR A
[1]   ⎕ES(0=A)/5 4
[2]   Z←!A
[3]   ∇
      FACTR 3
6

      FACTR 0
DOMAIN ERROR
      FACTR 0
      ∧

      ⎕EM
DOMAIN ERROR
      FACTR 0
      ∧

      ⎕ET
5 4
```

**When *R* Is `0 0`:** In immediate execution, `⎕ES 0 0` has no effect. In a defined operation, `⎕ES 0 0` sets `⎕ET` to `0 0`, `⎕EM` to `3 0⍴' '` but does not simulate an event.

```
       ∇FN
[1]    '2+3' ⎕EA'(A←B)←2)'    ⍝ Causes a syntax error.
[2]    ⎕ET                    ⍝ Reports 2 4 as the event
                              ⍝ type.
[3]    ⎕ES 0 0                ⍝ Resets ⎕EM and ⎕ET.
[4]    ⎕ET                    ⍝ Shows ⎕ET reset to 0 0.
[5]    ∇
       FN
5
2 4
0 0
```

***When R Is a Simple, Two-Item Integer Vector which is not a defined error code***: The value of R is assigned to ⎕ET. An event simulation is generated in the expression that invoked the function, but no message is reported.

```
      ∇Z←RECIP A
[1]   ⎕ES(0=A)/13 17
[2]   Z←÷A
[3]   ∇
      RECIP 3
0.3333333333

      RECIP 0
      RECIP 0
      ^

      ⎕EM

      RECIP 0
      ^

      ⎕ET
13 17
```

***When R Is Empty***: No action is taken. This gives you the ability to signal an event conditionally:

```
⎕ES (cond)/R
```

If the condition is true, the event is simulated. If the condition is not true, no action is taken. The functions EXPO, FACTR, and RECIP used in earlier examples each signal an event conditionally.

## ⎕ES Event Simulate (with both Error Message and Event Type)

*L* ⎕*ES R* Simulates an error, generates an error report, and returns the left argument as the first row of the error message matrix (⎕*EM*).

*L*: Simple character scalar or vector
*R*: Simple character scalar or vector; simple two-item vector of integers between ¯32767 and 32767; or empty vector.

When ⎕*ES* is executed and if *R* is not empty, an error condition is simulated, *R* is assigned to ⎕*ET*, and an APL2 error message matrix is generated with the following contents:

- First row is *L*.
- Second row is the expression or the name of the function within which ⎕*ES* was executed.
- Third row contains the carets marking the error.

```
      'ERROR SIMULATION' ⎕ES 101 9
ERROR SIMULATION
      'ERROR SIMULATION' ⎕ES 101 9
      ^                  ^

      ⎕EM
ERROR SIMULATION
      'ERROR SIMULATION' ⎕ES 101 9
      ^                  ^

      ⎕ET
101 9
```

If ⎕*ES* is executed from within a defined function or operator, the event action is generated as though the function were locked or primitive. Suspension occurs at the calling point, not within the defined operation.

Unlike a monadic event simulate, even though R is an error code defined for ⎕ET, the normally associated error message is not displayed. The character scalar or vector L is always displayed.

```
      ∇Z←FACTR A
[1]   'ZERO INVALID' ⎕ES(0=A)/5 4
[2]   Z←!A
[3]   ∇
      FACTR 4
24

      FACTR 0
ZERO INVALID
      FACTR 0
      ^

      ⎕EM
ZERO INVALID
      FACTR 0
      ^

      ⎕ET
5 4
```

***When*** R ***is*** 0 0: The left argument is ignored and the behavior of monadic ⎕ES is seen.

***When*** R ***Is Empty***: No action is taken. This gives you the ability to signal an event conditionally:

L ⎕ES (*cond*)/R

If the condition is true, the event is simulated. If the condition is not true, no action is taken. The functions FACTR and FMT shown earlier are examples of functions that signal an event conditionally.

# ⎕ET Event Type

| | |
|---|---|
| ⎕ET | Two-integer code indicating the type of the event (error) associated with the first line of the state indicator. |
| Default value: | `0 0` |
| Variable type: | Debug variable; assigning or localizing ⎕ET has no effect. |

The first item of ⎕ET indicates the major classification of the event; the second indicates a more specific category. As a debug variable, ⎕ET can be used to discover the possible source of an error. Figure 32 lists the major classes, the specific event type codes, and their meanings.

*Figure 32. Event Type Codes*

| **Major Class** | **Event Type Code and Description** |
|---|---|
| 0<br>Defaults | 0 0 - No error<br>0 1 - Unclassified event (⎕ES '*message*') |
| 1<br>Resource<br>Errors | 1 1 - *INTERRUPT*<br>1 2 - *SYSTEM ERROR*<br>1 3 - *WS FULL*<br>1 4 - *SYSTEM LIMIT* - symbol table<br>1 5 - *SYSTEM LIMIT* - interface unavailable<br>1 6 - *SYSTEM LIMIT* - interface quota<br>1 7 - *SYSTEM LIMIT* - interface capacity<br>1 8 - *SYSTEM LIMIT* - array rank<br>1 9 - *SYSTEM LIMIT* - array size<br>1 10 - *SYSTEM LIMIT* - array depth<br>1 11 - *SYSTEM LIMIT* - prompt length<br>1 12 - *SYSTEM LIMIT* - interface representation<br>1 13 - *SYSTEM LIMIT* - implementation restriction |
| 2<br>*SYNTAX ERROR* | 2 1 - Required operand or right argument omitted (2×)<br>2 2 - Ill-formed line ([( ])<br>2 3 - Name class (3←2)<br>2 4 - Invalid operation in context ((*A*←*B*)←2)<br>2 5 - Compatibility setting prohibits this syntax |
| 3<br>*VALUE ERROR* | 3 1 - Name with no value<br>3 2 - Function with no result |
| 4<br>Implicit<br>Argument<br>Errors | 4 1 - ⎕*PP ERROR*<br>4 2 - ⎕*IO ERROR*<br>4 3 - ⎕*CT ERROR*<br>4 4 - ⎕*FC ERROR*<br>4 5 - ⎕*RL ERROR*<br>4 7 - ⎕*PR ERROR* |
| 5<br>Explicit<br>Argument<br>Errors | 5 1 - *VALENCE ERROR*<br>5 2 - *RANK ERROR*<br>5 3 - *LENGTH ERROR*<br>5 4 - *DOMAIN ERROR*<br>5 5 - *INDEX ERROR*<br>5 6 - *AXIS ERROR* |

All undefined major event classifications numbered 0 through 99 are reserved. Note that processor 11 external functions, and APL functions (through ⎕ES), can signal events with arbitrary numbers. For more information about particular errors, see Chapter 11, “Interpreter Messages” on page 461.

The following examples show a reference of ⎕ET after an error.

```
      (A←B)←2)
SYNTAX ERROR+
      (A←B)←2)
        ^
      ⎕ET
2 4
```

```
      (ι68)ρ15
SYSTEM LIMIT+
      (ι68)ρ15
      ^    ^
      ⎕ET
1 8
```

⎕ET is automatically local to a function called by a line entered in immediate execution. If there is not enough room in the workspace to suspend the statement in error, WS FULL is reported, ⎕EM is set to a character matrix of shape 3 0, and ⎕L and ⎕R are not set.

⎕ES can set ⎕ET as part of event handling within a defined function.

**⎕ET *and the State Indicator***: ⎕ET contains the event type associated with the top line of the state indicator. As the stack is cleared (with → or )RESET *n*), ⎕ET is reset to the event type associated with the current top line of the state indicator.

If the state indicator is clear, the value of ⎕ET is 0 0.

## ⎕EX Expunge

`Z←⎕EX R` Returns a `1` if the object is disassociated, and returns a `0` if it cannot be disassociated. An object cannot be disassociated for the following reasons:

- The object is a system function.
- The name is not valid.
- The object is an external object and cannot be disassociated at this time.

`R`: Simple character scalar, vector, or matrix
`Z`: Simple Boolean scalar or vector

```
 ρZ  ←→  ¯1↓ρR
ρρZ  ←→  ,0⌈¯1+ρρR
```

Each row of `R` is interpreted as a constructed name. Currently active user names are disassociated from their values, and if they represent shared variables, the shares are retracted. The following system variables can be disassociated from their values: `⎕CT`, `⎕FC`, `⎕IO`, `⎕LX`, `⎕PP`, `⎕PR`, and `⎕RL`. The remaining system variables and system functions cannot be disassociated from their values.

```
      RUNS←3
      ⎕EX 'RUNS'
1
      RUNS
VALUE ERROR+
      RUNS
      ^
      SCORE←43
      ⎕NC 'SCORE'
2
      ⎕EX '⎕NC'
0
      ⎕NC 'SCORE'
2
```

```
      RUNS←1
      ⎕FX 'Z←HITS X' 'Z←+/X'
HITS
      ERRS←2
      ⎕EX 3 4ρ'HITSRUNSERRS'
1 1 1

      ⎕NLT←'SVENSKA'
      ⎕EX '⎕NLT'
1
      ⎕NLT
SVENSKA
```

If an implicit argument system variable is expunged, a primitive function that depends on it as an implicit argument generates an error.

```
      ⎕IO
1
      ⎕EX '⎕IO'
1
      ⍳10
⎕IO ERROR
      ⍳10
      ^
```

Suspended or pendent defined functions can be expunged. However, expunging such functions does not affect their definitions in the state indicator. Until they are cleared from the state indicator, these functions exist only in the state indicator and cannot be edited. See “Clearing the State Indicator” on page 357 for information on clearing the state indicator.

```
      ∇ Z←SQUARE R
[1]   Z←R*2
[2]   ∇
      R←'T'

      SQUARE R
DOMAIN ERROR
SQUARE[1]  Z←R*2
            ^ ^

      ⎕EX 'SQUARE'
1

      )SIS
SQUARE[1] Z←R*2
           ^ ^
*  SQUARE R
   ^

      SQUARE 5
VALUE ERROR+
      SQUARE 5
      ^

      R←5
      →⎕LC
25
      SQUARE 5
VALUE ERROR+
      SQUARE 5
      ^
```

**Relationship to** `)ERASE`**:** `)ERASE` (page 428) removes global variables, defined functions, and defined operators from the active workspace.

## ⎕FC Format Control

`⎕FC←A` Specifies or references characters for decimal point, thousands indicator, fill character, overflow indicator, print-as-blank character, and negative number indicator. It is used by format by example and format by specification (`L⍕R`).

`A`: Simple character vector
Default value: `.,*0_¯`
Variable type: Implicit argument

Although `⎕FC` may be a character vector of any length, only the first six characters are used. If fewer than six characters are specified, the defaults for the missing characters are used. Figure 33 gives the meaning of each of the first six items.

*Figure 33. Format Control Items*

| Item | Default | Meaning |
|---|---|---|
| `⎕FC[1]` | `.` | Character for decimal point |
| `⎕FC[2]` | `,` | Character for thousands indicator |
| `⎕FC[3]` | `*` | Fill for blanks indicated by the digit `8` in format by example |
| `⎕FC[4]` | `0` | Fill for overflows that otherwise cause a `DOMAIN ERROR` |
| `⎕FC[5]` | `_` | Print-as-blank (cannot be `,.0123456789`) |
| `⎕FC[6]` | `¯` | Negative number indicator |

All items of `⎕FC` except `⎕FC[6]` are used as implicit arguments to format by example, page 139. Items `⎕FC[1 4 6]` are used as implicit arguments to format by specification.

If `⎕FC` is assigned an invalid value or erased and then implicitly used by format, a `⎕FC ERROR` results.

A reference of `⎕FC` yields its last specified value.

## ⎕FX Fix (No Execution Properties)

Z←⎕FX R Establishes in the active workspace the defined function or operator represented in character form by R.

R: Simple character matrix or a vector whose items are character vectors or character scalars.
Z: Character vector or integer scalar

Implicit argument: ⎕IO

R represents the definition, in character form, of a function or operator. If the definition is valid, the function or operator is established in the workspace, and the name of the object is returned as the result. Thus, ⎕FX is an alternative to using an editor to define a function or operator. (See Chapter 9, “The APL2 Editors” on page 375.)

```
      ⎕FX 'Z←FMT R' 'Z←⍕R'
FMT

      FMT 'ABCDEF'
ABCDEF

      ⎕FX 'Z←FACTR R' '''⎕ET'' ⎕EA ''Z←!R'''
FACTR

      FACTR 5
120
```

R must be a name unassociated with an object or the name of an existing defined function or operator.

***Invalid Definition:*** If the definition is not valid, Z is a scalar integer indicating the first row of the function or operator line in error. This integer is dependent on ⎕IO.

```
      ⎕FX 'Z←FN R' 'Z←1+R×2' ⎕AV[1]
3

      ⎕IO←0
      ⎕FX 'Z←FN R' 'Z←1+R×2' ⎕AV[1]
2
```

***Acceptable Variations in Format of R:*** ⎕FX accepts a character form with the following variations from that form produced by applying ⎕CR to a definition established in the workspace:

- R may contain unnecessary blanks.
- The header may have blanks instead of semicolons between local names.
- R may be a vector of character scalars and/or vectors instead of a character matrix.
- R may have trailing blanks on comments.

***Changing the Definition of a Suspended or Pendent Operation:*** Suspended or pendent defined functions and operators can be changed by using ⎕FX to establish a new definition. Establishing a new definition for the object in the workspace, does *not*, however, change the definition of the function or operator in the state indicator. The previously invoked definition is retained until it completes execution or is cleared from the state indicator.

After the application of ⎕FX, the previously invoked definition in the state indicator and the current definition can differ.

```
      ⎕FX 'FUNC' '1' '2' '!¯3' '4'
FUNC

      FUNC
1
2
DOMAIN ERROR
FUNC[3]  !¯3
         ^

      )SI
FUNC[3]
*

      ⎕FX 'FUNC' '22' '23' '24' '25'
FUNC
      →4
4

      FUNC
22
23
24
25
```

## ⎕FX Fix (with Execution Properties)

Z←L ⎕FX R Establishes in the active workspace the defined function or operator represented in character form by R with execution properties specified by L.

L: Simple four-item Boolean vector or a Boolean scalar
R: Simple character matrix or a vector whose items are character vectors or character scalars.
Z: Name of the established object or integer scalar

Implicit argument: ⎕IO

As with ⎕FX (with no execution properties), R represents the definition, in character form, of a function or operator.

If L is a four-item Boolean vector, each item of L turns on (1) or off (0) one of four independent *execution properties*:

| | |
|---|---|
| L[1] | Cannot be displayed |
| L[2] | Cannot be suspended |
| L[3] | Ignores attention signal |
| L[4] | Converts any nonresource error to DOMAIN ERROR |

If L is a Boolean scalar, it is used to turn on or off all the above properties.

The function or operator named in R must be either undefined or the name of an existing defined function or operator. If the definition is valid, the function or operator is established in the workspace with the execution properties specified, and the name of the object is returned as the result.

If R is not a valid function or operator definition, Z is a scalar integer that indicates the row of the function or operator line in error. This integer is dependent on ⎕IO.

***Execution Properties:*** Each property can be set independently. If all four execution properties are set, the defined function or operator is locked, as it is with ⍫ when you use an APL2 editor.

If `L[1]` is `1`, the defined function or operator cannot be displayed—not through `⎕CR`, `⎕TF`, or the APL2 editors. Also, the object cannot be traced or edited. An attempt to display it generates a `DEFN ERROR`.

```
      1 0 0 0 ⎕FX 'Z←FACTR R' 'Z←!R'
FACTR

      ∇FACTR[⎕]∇
DEFN ERROR
      ∇FACTR[⎕]∇
               ^
```

If `L[2]` is `1`, the defined function or operator cannot be suspended by an error or an interrupt. The error or interrupt message is displayed, but the operation is not suspended. The state indicator shows the error or interrupt as occurring during the invocation of the operation.

```
      0 1 0 0 ⎕FX 'Z←FACTR R' 'Z←!R'
FACTR

      FACTR ¯3
DOMAIN ERROR
      FACTR ¯3
      ^

      )SIS
*  FACTR ¯3
   ^
```

If `L[3]` is `1`, the defined function or operator ignores the attention signal and stop control settings.

```
      0 0 1 0 ⎕FX 'Z←FACTR R' 'Z←!R'
FACTR

      S∆FACTR←1
      FACTR 4
24
```

If `L[4]` is `1`, an error other than a resource error is converted into a `DOMAIN ERROR`. Resource errors are listed in Figure 32 on page 287.

```
      0 0 0 1 ⎕FX 'Z←L INDEX R' 'Z←R[L]'
INDEX

      3 INDEX 3 4
DOMAIN ERROR
INDEX[1]  Z←R[L]
            ^^
```

**Changing Execution Properties:** If a defined function or operator can be displayed, its execution properties can be changed by executing an expression in the following format:

`L ⎕FX ⎕CR 'name'`

For example:

```
      0 ⎕FX ⎕CR 'INDEX'
INDEX

      3 INDEX 3 4
INDEX ERROR
INDEX[1]  Z←R[L]
           ^^
```

## ⎕IO Index Origin

⎕IO←A    Contains the index of the first item of a nonempty vector.

A: 0 or 1
Default value: 1
Variable Type: Implicit argument

```
      ⎕IO
1
      R←34 18 24
      R[1]
34

      ⎕IO←0
      R[1]
18
      R[0]
34
```

```
      ⎕IO←1
      R←34 18 24
      ⍋R
2 3 1
      ⎕IO←0
      ⍋R
1 2 0
```

If ⎕IO is assigned an invalid value or erased and then implicitly used by another function, a ⎕IO ERROR results.

A reference of ⎕IO yields its current value.

***Primitive Functions That Use ⎕IO:*** ⎕IO is an implicit argument of the following functions**:**

| | | |
|---|---|---|
| Bracket indexing | A[I] | page 70 |
| Deal | L?R | page 89 |
| Grade down | ⍒R | page 147 |
| Grade down (with collating sequence) | L⍒R | page 147 |
| Grade up | ⍋R | page 153 |
| Grade up (with collating sequence) | L⍋R | page 155 |
| Index of | L⍳R | page 162 |
| Interval | ⍳R | page 168 |
| Pick | L⊃R | page 195 |
| Roll | ?R | page 231 |
| Transpose (general) | L⍉R | page 251 |

Index origin also affects axis specification, page 23, and ⎕FX Fix, pages 292 and 294.

## ⎕L Left Argument

| | |
|---|---|
| ⎕L←A | If the first line of the state indicator contains a dyadic function whose execution was suspended by an error or an interrupt, ⎕L is the array value of its left argument. ⎕L can be respecified and execution resumed at the point of the error or interrupt by →⍳0. |

A: New left argument
Default value: None
Variable type: Debug

⎕L is set when an error occurs in a primitive dyadic function. Effectively, it is automatically local to a function called by a line entered in immediate execution and exists only while the statement in error is suspended.

```
      ∇ Z←F A
[1]   Z←(2×A)+3 4 5
[2]   ∇
      F 6 7 10
15 18 25
      F 6 7
LENGTH ERROR
F[1]  Z←(2×A)+3 4 5
       ^     ^
      ⎕L
12 14
      ⎕L←12 14 20
      →⍳0
15 18 25
```

If there is not enough room in the workspace to suspend the statement in error, *WS FULL* is reported. ⎕EM is set to a character matrix of shape `3 0`, and ⎕L and ⎕R are not set.

***With*** *VALENCE ERROR*: If the primitive function fails because of a *VALENCE ERROR*, ⎕L can be respecified only if it is not referenced first. In this situation, if ⎕L is referenced first, a *VALUE ERROR* results.

**Assignment First**

```
      ∇ Z←FDROP R
[1]   Z←↓⍳R
[2]   ∇
      FDROP 8
VALENCE ERROR
FDROP[1]  Z←↓⍳R
           ^^
      ⎕L←¯3
      ⎕L
¯3
      →⍳0
1 2 3 4 5
```

**Reference First**

```
      FDROP 8
VALENCE ERROR
FDROP[1]  Z←↓⍳R
           ^^
      ⎕L
VALUE ERROR+
      ⎕L
      ^
      ⎕L←¯3
     →⍳0
VALUE ERROR+
      ⎕L
      ^
```

***With** VALUE ERROR **or** SYNTAX ERROR*: If the primitive function fails because of a *VALUE ERROR* or a *SYNTAX ERROR*, any respecification of ⎕*L* is ignored, and a reference to ⎕*L* generates a *VALUE ERROR*.

```
      1(2(3)×10+4(5 6)
SYNTAX ERROR+
      1(2(3)×10+4(5 6)
       ^
      ⎕L←5
      →⍳0
SYNTAX ERROR+
      1(2(3)×10+4(5 6)
       ^
      ⎕L
VALUE ERROR+
      ⎕L
      ^
```

***Effect of Resuming Execution:*** Note that the branch expression →⍳0 causes the suspended function to resume at the point of the error with the new value of the left argument. Everything in the statement to the right of the leftmost caret was evaluated prior to the error; only the function indicated by the rightmost caret is reevaluated when execution begins.

```
      ∇ Z←FL A
[1]   Z←(A×1 2 3)÷⍴A
[2]   ∇
      FL 4 5 6
1.333333333 3.333333333 6
      FL 4 5
LENGTH ERROR
FL[1]  Z←(A×1 2 3)÷⍴A
         ^ ^
      ⎕L
4 5
      ⎕L←4 5 6
      →⍳0
2 5 9
```

Because the final result can be misleading, it is important to know where execution resumes after respecification of ⎕*L*. It can be especially important if the statement in error contains shared variables or defined functions or operators.

***⎕L and the State Indicator***: As the state indicator is cleared (with → or )*RESET* *n*), ⎕*L* is reset to the left argument of the primitive function associated with the current first line of the state indicator, if its execution was suspended by an error or an interrupt.

If the state indicator is clear or if the error associated with the first line in the state indicator is not in a primitive function, ⎕*L* has no value.

## ⎕LC Line Counter

| | |
|---|---|
| ⎕LC | Contains the line numbers of defined functions and operators in execution or halted (suspended or pendent), with the most recently activated line number first. |

Default value: Empty vector
Variable type: Debug variable; specifying or localizing ⎕LC has no effect.

If displayed from within a defined function or operator, ⎕LC contains:

- Line number where ⎕LC appears
- Number of the last line executed in each pendent defined function or operator.
- Number of the last line executed in each suspended defined function or operator.

```
      ∇  G
[1]   'G LINE 1'
[2]   'G: ',⎕LC
[3]    H
[4]    ∇
```

```
      ∇  H
[1]   'H LINE 1'
[2]   'H LINE 2'
[3]   'H LINE 3'
[4]   'H: ',⎕LC
[5]    J
[6]   'H LINE 6'
[7]    ∇
```

```
      ∇ J
[1]   'J LINE 1'
[2]   'J LINE 2'
[3]   'J LINE 3'
[4]   'J LINE 4'
[5]   'J LINE 5'
[6]   'J: ',⎕LC
[7]   'J LINE 7'
[8]    ∇
```

During the execution sequence entered by invoking G, notice how the value of the line counter changes:

```
      G
G LINE 1
G:  2
H LINE 1
H LINE 2
H LINE 3
H:  4 3
J LINE 1
J LINE 2
J LINE 3
J LINE 4
J LINE 5
J:  6 5 3
J LINE 7
H LINE 6
```

If referenced while execution is halted, ⎕*LC* contains the number of the last line activated for each suspended and pendent function, with the most recently activated line first. Each item of ⎕*LC* corresponds to a line of the state indicator that contains a name, as reported by )*SI*, )*SIS*, or )*SINL*.

```
      ∇J[6.1] ∘∘∘∘∘∘∇

    ∇ Z←FACTR A
[1]   Z←!A
    ∇
      FACTR ¯3
DOMAIN ERROR
FACTR[1]  Z←!A
           ^^
      ⎕LC
1
      FACTR ¯6
DOMAIN ERROR
FACTR[1]  Z←!A
           ^^
      ⎕LC
1 1
      G
G LINE 1
G:  2 1 1
H LINE 1
H LINE 2
H LINE 3
H:  4 3 1 1
J LINE 1
J LINE 2
J LINE 3
J LINE 4
J LINE 5
J:  6 5 3 1 1
SYNTAX ERROR+
J[7]  ∘∘∘∘∘∘
           ^
```

```
      ⎕LC
7 5 3 1 1
      )SIS
J[7] ∘∘∘∘∘∘
          ^
H[5] J
     ^
G[3] H
     ^
*   G
    ^
FACTR[1] Z←!A
          ^^
*   FACTR ¯6
    ^
FACTR[1] Z←!A
          ^^
*   FACTR ¯3
    ^
```

During debugging, a branch to the line counter (→⎕*LC*) resumes execution with the line number that is the first item of ⎕*LC*.

```
      ∇J[∆7]∇
SI WARNING
      ⎕LC
7 5 3 1 1
      →⎕LC
J LINE 7
H LINE 6
      ⎕LC
1 1
      ⎕R
¯6
```

## ⎕LX Latent Expression

⎕LX←A  Specifies or references the APL2 statement that is automatically executed (by ⍎⎕LX) whenever the workspace is loaded.

A: Simple character scalar or vector
Default value: ''

⎕LX can be used to display a message, invoke an operation, or resume an interrupted operation. For example, to put the copyright notice into the workspaces distributed with APL2, the developer loaded the workspace, defined a variable named COIBM and then set ⎕LX:

```
      ⎕LX←'COIBM'

      )SAVE 1 EXAMPLES
1993-05-21 13.59.50 (GMT-7)
```

When the workspace is loaded, the latent expression is executed automatically.

```
      )LOAD 1 EXAMPLES
Saved 1993-05-21 13.59.50 (GMT-7)
LICENSED MATERIALS - PROPERTY OF IBM
5688-228 (C) COPYRIGHT IBM CORP. 1984, 1994.
```

A reference of ⎕LX yields its current value.

```
      ⎕LX
COIBM
```

If single quotation marks enclose a name of a variable, the value of the variable is displayed. And, if single quotation marks enclose an expression, the expression is evaluated. In other words, when the workspace is loaded, the execute function is applied to ⎕LX:

```
  ⍎⎕LX
```

For a character vector to be printed when the workspace is loaded, the string must be enclosed within three sets of quotation marks. One set encloses the data specified to ⎕LX, and the other two sets indicate quotation mark characters. See also "⍎ Execute" on page 120.

```
      ⎕LX←'''USE THE XYZ GUIDE WITH THIS WS'''

      ⎕LX
'USE THE XYZ GUIDE WITH THIS WS'

      )SAVE COURSE
10.17.24  1993-05-21 (GMT-7)
```

```
      )LOAD COURSE
SAVED 10.17.24  1993-05-21 (GMT-7)
USE THE XYZ GUIDE WITH THIS WS
```

## ⎕NA Name Association (Inquire)

```
Z← ⎕NA R    Queries the associations of the objects named in R.

R: Simple character scalar, vector, or matrix of names
Z: Two item vector or two-column matrix

ρZ   ←→ (¯1↓ρR),2
ρρZ  ←→ ,0⌈¯1+ρρR
```

Each row of Z corresponds to a row of R and provides:

Z[1] The array that was passed to the processor when the name was activated.

Z[2] The processor with which the name is associated.

Names in the APL workspace not otherwise associated are associated with processor 0 (APL itself). For such names, the name class is returned as the first item of Z. Invalid names in R return ¯1 0 in Z.

```
      3 11 ⎕NA 'PFA'
1
      ⎕NA 'PFA'
3 11
     '(AP2VN011)' 11 ⎕NA 'OPTION'
1
      ⎕NA 'OPTION'
 (AP2VN011) 11
      ⎕EX 'DATA'
1
      DATA←'STUFF'
      ⎕NA 'DATA'
2 0
```

A surrogate name can be specified but must match the original surrogate name.

```
      3 11 ⎕NA 'PATTERN PFA'
1
      ⎕NA 'PATTERN NEWNAME'
¯1 0
```

## ⎕NA Name Association (Set)

`Z←L ⎕NA R` Associates names `R` with external objects that are accessed through associated processors. `L` identifies the external processors and contains information passed to them. The result is `1` if the specified association is active, or `0` if it is not.

`L`: Two-item vector or a two-column matrix
`R`: Simple character scalar, vector, or matrix of names
`Z`: Boolean scalar or vector

```
ρZ  ←→ ¯1↓ρR
ρρZ ←→ ,0⌈¯1+ρρR
```

Each row of `R` is interpreted as a name or a name and a surrogate name. Each row of `L` corresponds to a row of `R` and provides:

| | |
|---|---|
| `L[1]` | An array which is passed to the processor when the name is activated. The content and use of this array is determined by the processor to which it is passed. |
| `L[2]` | A nonnegative integer used to identify the processor. The integer zero refers to APL itself. Positive integers refer to other associated processors. |

The result `Z` is a Boolean scalar or vector containing items corresponding to the rows of `L` and `R`. A `1` in the result indicates that the corresponding name was successfully associated with the specified processor and accepted, or activated, by that processor. A `0` in the result indicates that the corresponding name cannot be associated with the specified processor or has not been accepted (activated) by that processor.

Names can be associated through a processor with routines written in languages other than APL, with values that exist outside the workspace, or with APL objects in namespaces. Once a name has been successfully associated with a processor and activated, it behaves like other APL names, except that its value or definition does not exist in the user's active workspace.

```
      0 11 ⎕NA 'DISPLAY'
1
      ⎕NC 'DISPLAY'
3
      1 ⎕AT 'DISPLAY'
1 2 0
      DISPLAY 'NOW' 'IS' 'THE' 'TIME'
.→-------------------------.
| .→--. .→-. .→--. .→---. |
| |NOW| |IS| |THE| |TIME| |
| '---' '--' '---' '----' |
'∊-------------------------'
```

If the processor specified in the left argument of ⎕NA does not exist or if it cannot satisfy the request to activate a name, or if it returns invalid information when contacted, a 0 is returned as the result of ⎕NA, and the name class of the specified object does not change. )MORE can provide additional information about the failure.

If the processor specified in the left argument of ⎕NA does activate the specified name, it must assign name class and attributes to that name if the name did not exist previously in the APL workspace. If the name did exist before ⎕NA was issued, its name class and valence (1 ⎕AT) are not changed as a result of ⎕NA.

```
      ⎕EX 'PFA'
1
       0 11 ⎕NA 'PFA'
1
      ⎕NC 'PFA'
3
      1 ⎕AT 'PFA'
1 2 0
```

An attempt to activate a name that already exists is only successful if the left argument of ⎕NA matches the original left argument of ⎕NA specified when the name was originally activated. This original left argument of ⎕NA can be obtained by issuing monadic ⎕NA for the specified name.

```
      0 11 ⎕NA 'PFA'
1
      ⎕NA 'PFA'
0 11
      3 11 ⎕NA 'PFA'
0
      0 11 ⎕NA 'PFA'
1
      (⎕NA 'PFA') ⎕NA 'PFA'
1
```

Processor 0 is APL itself and allows names in the active workspace to be specified in the left argument of `⎕NA`. Processor 0 expects a valid name class (a digit between `1` and `4`) as the first item of the left argument of `⎕NA` and returns a `1` if:

- the named object exists
- the named object is not associated with another processor
- the named object has a name class which matches that specified.

```
      A←'THIS IS A VARIABLE'
      ⎕NC 'A'
2
      2 0 ⎕NA 'A'
1
      3 0 ⎕NA 'A'
0
```

Processor 0 does not establish names that did not previously exist in the workspace.

***Conformability:*** If `R` is a scalar or vector, `L` must be a two-item vector. If `R` is a matrix, `L` must be a two-column matrix with the same number of rows as `R`, or a two-item vector, in which case it is reshaped to `(¯1↓⍴R),2` before attempting to contact the processor.

The following two expressions are equivalent:

```
      (2 2⍴3 11) ⎕NA 2 3⍴'ATRRTA'
1 1
      3 11 ⎕NA 2 3⍴'ATRRTA'
1 1
```

***Persistence of Associated Names:*** Once a name has been associated with a processor and activated (a result of `1` from `⎕NA`), that name retains its name class, valence and association with the processor until explicitly removed by `⎕EX`, `)ERASE`, `)COPY`, or `)IN`. In particular, the association is retained if the workspace is saved and subsequently reloaded. Associated names can be copied with `)COPY` and `)PCOPY` and retain their name class, valence, and association. An attempt to establish an associated name with `)IN` or `2 ⎕TF`, fails unless the specified processor activates the name. `)IN` returns a `NOT COPIED` message for such failures.

When an associated name is erased, the object and storage with which it is associated is retained until all references to the association are discarded. The other references could arise because of partially executed expressions on the execution stack. The command `)RESET` can be used to discard partially executed expressions.

***Surrogate Names:*** APL2 permits the use of an alias name for associated names (except those associated with Processor 0). This alias is called the surrogate name and can be used to avoid name conflicts. The associated processor recognizes the surrogate name as the name of the object.

When a row of `R` contains a pair of names (separated by spaces), the first name in the pair represents the name of the object to be associated and the second is the name by which the object is known to the associated processor. For example, the following expression associates a function which is referred to as `Bld_Struct` in the workspace, but known as `ATR` by the associated processor:

```
      3 11 ⎕NA 'Bld_Struct ATR'
1
```

If a surrogate is specified for a name already associated with a processor, it must be the same as the one originally used.

```
      3 11 ⎕NA 'Bld_Struct ATR'
1
      3 11 ⎕NA 'Bld_Struct XXX'
0
      3 11 ⎕NA 'Bld_Struct'
1
```

The surrogate name, if one exists, can be determined through the use of `2 ⎕TF`:

```
      2 ⎕TF 'Bld_Struct'
3 11 ⎕NA 'Bld_Struct ATR'
```

# ⎕NC Name Class

`Z←⎕NC R` Returns the name class of objects named in R.

R: Simple character scalar, vector, or matrix
Z: Simple integer scalar or vector

```
 ρZ ←→ ¯1↓ρR
ρρZ ←→ ,0⌈¯1+ρρR
```

R is taken to represent constructed names—either user or distinguished. If more than one name is specified in R, R must be a matrix, with each row representing a constructed name.

Each item of Z is the name class of the corresponding name in R. The items in Z have the following meanings:

| | |
|---|---|
| ¯1 | Invalid name or unused distinguished name |
| 0 | Unused but validly constructed user name |
| 1 | Label |
| 2 | Variable |
| 3 | Function |
| 4 | Operator |

The following examples use the workspace DUMMY whose contents are shown in the figure below.

**CONTENTS OF THE WORKSPACE** DUMMY

**Functions**
C F G HI ROD

**Variables**
DRY FAT IOD ME PRO SALT

**Operators**
HEX MOP TRI

**Labels**
FLAB X

*Figure 34. Sample Workspace for Name List and Name Class System Functions*

```
      ⎕NC 2 3ρ'DRYC  '
2 3

      ⎕NC ⊃'ROD' 'PRO' 'MOP'
3 2 4
```

Symbols representing primitive functions and operators are classified as invalid names. Distinguished names are treated like functions and variables. A distinguished name unassociated with a value is considered invalid.

```
      ⎕NC 4 1ρ'*<5['
¯1 ¯1 ¯1 ¯1

      ⎕NC 2 1ρ'X' 'Z'
0 0
```

```
      ⎕NC 3 3ρ'⎕EA⎕TS⎕KZ'
3 2 ¯1
```

An undefined name is classified as a variable if it has been shared but not yet assigned.

```
      102 ⎕SVO¨ 'CTL102' 'DAT102'
2 2
      ⎕NC ⊃[2] 'CTL102' 'DAT102'
2 2
```

***Name Class of Local Names:*** If ⎕NC is used during the execution of a defined operation or if execution is suspended or stopped, the name class of labels, parameters, and other local names can be queried. The name class is given for local objects, not for similarly named global objects, which are shadowed by the local objects.

```
      ∇Z←L F R
[1]   Z←⎕NC 2 1ρ'LR'
[2]   ∇
      F 2
0 2
      4 F 2
2 2

      ∇Z←L(F OP G)R;V X
[1]   TAG:V←1
[2]   ⎕NC 8 3ρ'L  F  OP G  R  V  X  TAG'
[3]   ∇
      3 +OP× 6
2 3 4 3 2 2 0 1

      (+OP 2)6
0 3 4 2 2 2 0 1
```

## ⎕NL Name List (by Alphabet and Class)

`Z←L ⎕NL R` Lists labels, variables, functions, and defined operators in the active workspace whose name class is `R` and whose first character is in `L`.

`L`: Simple character scalar or vector
`R`: Simple positive integer scalar or vector, `R∊⍳4`
`Z`: Simple character or empty matrix

The values of `R` have the following meanings:

| | |
|---|---|
| 1 | Label |
| 2 | Variable |
| 3 | Function |
| 4 | Defined operator |

The examples in this discussion refer to the workspace `DUMMY` shown in Figure 34 on page 309.

```
      'H' ⎕NL 3
HI
```

```
      'H' ⎕NL 4
HEX
```

If `L` is a vector, `Z` contains all objects in the name class `R` whose names begin with an item of `L`.

```
      'CF' ⎕NL 3
C
F
```

```
      'HMT' ⎕NL 4
HEX
MOP
TRI
```

If `R` is a vector, `Z` is a list of all objects in the name classes `R` whose names begin with an item of `L`.

```
      'F' ⎕NL 2 3
F
FAT
```

```
      'FH' ⎕NL 2 3
F
FAT
HI
```

If executed while a defined function or operator is halted, `L ⎕NL R` returns a list of objects that includes labels and local names. The name class of local objects shadows the name class of global objects of the same name.

```
      'FX' ⎕NL 1 2 3
F
FAT
```

```
      S∆F←1
      F 3
F[1]
      'FX' ⎕NL 1 2 3
F
FAT
FLAB
X
```

## ⎕NL Name List (by Class)

`Z←⎕NL R` Lists labels, variables, functions, and defined operators in the active workspace whose name class is R.

R: Simple positive integer scalar or vector, `R∊⍳4`
Z: Simple character or empty matrix

The values of R have the following meanings:

| | |
|---|---|
| 1 | Label |
| 2 | Variable |
| 3 | Function |
| 4 | Defined operator |

The examples in this discussion refer to the workspace DUMMY shown in Figure 34 on page 309.

```
      ⎕NL 4
HEX
MOP
TRI
```

```
      ⎕NL 3
C
F
G
HI
ROD
```

```
      ⎕NL 3 4
C
F
G
HEX
HI
MOP
ROD
TRI
```

If R is a vector, Z is a list of all objects in the name classes R.

If executed while a defined function or operator is halted, `⎕NL R` returns a list of objects that includes labels and local names. Within a class, local objects are listed. Any similarly named global objects are shadowed by the local objects.

```
      ⎕NL 1 2
DRY
FAT
IOD
ME
PRO
SALT
```

```
      S∆F←1
      F 3
F[1]
      ⎕NL 1 2
DRY
FAT
FLAB
IOD
ME
PRO
SALT
X
```

## ⎕NLT National Language Translation

| | |
|---|---|
| ⎕NLT←A | Specifies or references the name of the national language in which system messages are reported and system commands can be entered. (System commands can also be entered in American English regardless of the current language. Any messages not defined in the current language are displayed in uppercase American English.) |

A: Simple character vector
Default value: Installation-dependent
Variable type: Session

When associated with a value available to the system, ⎕NLT sets the corresponding language as the language for the text of system commands and messages. If ⎕NLT is set to be empty or blank, uppercase American English is used (though commands can still be entered in mixed case). Any other assignment to ⎕NLT is ignored, and its last valid value is retained. Leading and trailing blanks in the value assigned to ⎕NLT are ignored.

In most cases, either the spelled-out language name, or a three-character abbreviation adopted across IBM products can be assigned to ⎕NLT, but the three-character abbreviation is never returned when ⎕NLT is referenced.

The initial value of ⎕NLT at the beginning of each session is an installation attribute. The system is shipped with a default of mixed-case English, but that default can be changed for any installation.

If the installation-specified language is not available in the system, ⎕NLT is initialized to an empty character vector, yielding uppercase English.

If an invalid value is assigned to ⎕NLT during the APL2 session (the language is not available, or the language file contains formatting errors), it remains set to its last valid value.

```
      ⎕NLT←'NOR'
      )AOID
ER AO NULLSTILT
      ⎕NLT
NORSK
      ⎕NLT←'MARTIAN'
      ⎕NLT
NORSK
      ⎕NLT←''
      )WSID
IS CLEAR WS
```

**Note:** If ⎕NLT is set to a value for which a user-defined language table exists, and there is an error in that table, the previous value of ⎕NLT is restored. This is treated like an implicit error, so normally no error message is displayed. One or more messages describing the problem have been queued, though, and can be displayed using )MORE.

## ⎕*PP* Printing Precision

| | |
|---|---|
| ⎕*PP*←*A* | Specifies or references the number of significant digits in the display of numbers. |

*A*: Positive integer scalar
Default value: 10
Variable type: Implicit argument

The minimum value for ⎕*PP* is 1. If ⎕*PP* is specified at or above the maximum precision displayed by the system, all available precision is displayed.

```
      2÷3
0.6666666667
```

```
      ⎕PP←200
      7÷9
0.77777777777777778
      ⎕PP
18
```

However, in some cases, ⎕*PP* does not influence the display of integers.

```
      ⎕PP←1
      333
333
```

If ⎕*PP* is assigned an invalid value or erased and then implicitly used by format, a ⎕*PP ERROR* results.

A reference of ⎕*PP* yields its current setting.

***Implicit Argument:*** ⎕*PP* is an implicit argument of default format (⍕), page 135, and output of numbers.

## ⎕PR Prompt Replacement

`⎕PR←A` Controls the interaction between an assignment (the *prompt*) and a successive reference (the *reference*) of the character input/output system variable (⍞).

*A*: Character scalar or vector of length 1 or empty vector
Default value: `1⍴' '` (character blank)
Variable type: Implicit argument

The character assigned to ⎕PR replaces the unchanged characters in the prompt (last row of character array assigned to ⍞), becoming the first part of the response vector. The remainder of the response vector contains data that was changed or added by the session manager.

All the examples below use the following defined function:

```
      ∇Z←F
[1]   ⍞←'ENTER NAME: '
[2]   Z←⍞
[3]   ∇
```

```
      ⎕PR←' '                  │         ⎕PR←'*'
      RESULT←F                 │         RESULT←F
ENTER NAME: MCMILLAN           │   ENTER NAME: MCMILLAN
      ⍴RESULT                  │         ⍴RESULT
20                             │   20
      RESULT                   │         RESULT
            MCMILLAN           │   ************MCMILLAN
```

Any part of the prompt that is changed by session input is not affected by the value of ⎕PR.

```
      ⎕PR←' '
      RESULT←F
ENTMCMILLAN          User input replaces
                     part of the prompt.
      ⍴RESULT
12
      RESULT
   MCMILLAN
```

If ⎕PR is an empty vector, unchanged characters in the prompt are not replaced, and the response vector contains unchanged prompt characters and the session input.

```
      ⎕PR←''
      RESULT←F
ENTER NAME: MCMILLAN
      ρRESULT
20
      RESULT
ENTER NAME: MCMILLAN
```

```
      RESULT←F
ENTER MCMILLAN
      ρRESULT
14
      RESULT
ENTER MCMILLAN
```

A reference of ⎕PR returns its current value.

If ⎕PR is assigned an invalid value or erased and then implicitly used by format, a ⎕PR ERROR results.

## ⎕PW Printing Width

⎕PW←A    Specifies or references the number of characters displayed per line of output.

A: Positive integer scalar
Default value:  System and device dependent
Variable type: Session

The minimum value that can be assigned to ⎕PW is 30.  If an invalid value is specified, it is ignored.  Display of an array R wider than the value of ⎕PW is folded at or just before the column specified by ⎕PW.  The folded portions are indented six spaces and are separated from the first part by N blank lines, where  N is `0⌈¯1+⍴⍴R`.

```
      W←'SUPERCALIFRAGILISTIC-EXPIALIDOCIOUS'
      ⍴W
35
      ⎕PW←30
      W
SUPERCALIFRAGILISTIC-EXPIALIDO
      TIOUS
```

The rows of a matrix are folded together and the pages of a multidimensional array are folded together.

```
      ⎕PW
30
      2 36⍴'AAaBBbCCcDDdEEeFFfGGgHHhIIiJJjKKkLLl'
AAaBBbCCcDDdEEeFFfGGgHHhIIiJJj
AAaBBbCCcDDdEEeFFfGGgHHhIIiJJj

      KKkLLl
      KKkLLl
```

The display of a simple array containing numbers may be folded at a width less than ⎕PW so that individual numbers are not split.

```
      2 3∘.⍟10 20 30
3.321928095 4.321928095
2.095903274 2.726833028

      4.906890596
      3.095903274
```

If ⎕PW is small and ⎕PP is large, the display of some complex numbers in a simple array may extend beyond ⎕PW.  If ⎕PW is at least `13+2×⎕PP`, individual numbers in a simple array do not exceed ⎕PW.  Numbers in a nested array may be split with any value of ⎕PP.

A reference of ⎕PW yields its current value.

## ⎕R Right Argument

`⎕R←A` If the first line of the state indicator contains a function whose execution was suspended by an error or an interrupt, `⎕R` is the array value of its right argument. `⎕R` can be respecified and execution resumed at the point of the error or the interrupt by `→⍳0`.

`A`: New right argument
Default value: None
Variable Type: Debug

`⎕R` is set when an error or interrupt occurs in a primitive function. Effectively, it is automatically local to a function called by a line entered in immediate execution and exists only while the statement in error is suspended.

```
      ∇Z←F R
[1]   Z←(R×1 2)+3 4 5
[2]   ∇
      F 10
LENGTH ERROR
F[1]  Z←(R×1 2)+3 4 5
        ^        ^
      ⎕R
3 4 5
      ⎕R←3 4
      →⍳0
13 24
```

If there is not enough room in the workspace to suspend the statement in error, `WS FULL` is reported. `⎕EM` is set to a character matrix of shape `3 0`, and `⎕R` and `⎕L` are not set.

***With*** `VALUE ERROR` ***or*** `SYNTAX ERROR`: If the error is a `VALUE ERROR` or a `SYNTAX ERROR`, any respecification of `⎕R` is ignored. If `⎕R` has not been set by the system, a subsequent reference to `⎕R` results in a `VALUE ERROR`.

```
      ⍝ ⎕R ASSIGNMENT IGNORED
      ∇Z←FA R
[1]   Z←(R×1 2)+3(4(5)
[2]   ∇
      FA 1
SYNTAX ERROR+
FA[1]  Z←(R×1 2)+3(4(5)
                  ^
      ∇FA[⎕1]
[1]   Z←(R×1 2)+3(4(5)
[1]   Z←(R×1 2)+3(4 5)
[2]   ∇
SI WARNING+

      ⎕R
4 5
      ⎕R←88 89
      →⍳0
SYNTAX ERROR+
FA[¯1]
      →1
 4  6 7
      ⍝ ⎕R VALUE ERROR
      ∇Z←FB R
[1]   Z←(R×1 2)+
[2]   ∇
      FB 3
SYNTAX ERROR+
FB[1]  Z←(R×1 2)+
         ^      ^
      ⎕R←4 5
      →⍳0
SYNTAX ERROR+
FB[1]  Z←(R×1 2)+
         ^      ^
      ⎕R
VALUE ERROR+
      ⎕R
      ^
```

***Effect of Resuming Execution:*** Note that the branch expression →⍳0 causes the suspended function to restart at the point of the error with the new value of the right argument. Everything in the statement to the right of the leftmost caret was evaluated prior to the error; only the function indicated by the rightmost caret is re-evaluated when execution begins.

```
      FR 4 5 6
1.333333333 3.333333333 6
      FR 4 5
LENGTH ERROR
FR[1]  Z←(1 2 3×A)÷⍴A
          ∧      ∧
      ⎕R
4 5
      ⎕R←4 5 6
      →⍳0
2 5 9
```

Because the final result can be misleading, it is important to know where execution resumes after respecification of ⎕R. It can be especially important if the statement in error contains shared variables or defined functions or operators.

⎕R ***and the State Indicator***: As the state indicator is cleared (with → or )RESET *n*), ⎕R is reset to the right argument of the primitive function associated with the current first line of the state indicator, if its execution was suspended by an error or an interrupt.

If the state indicator is clear or if the error associated with the first line in the state indicator is not in a primitive function, ⎕R has no value.

## ⎕RL Random Link

⎕RL←A Used or set to establish a basis for calculating random numbers.

Data: Simple positive integer scalar less than or equal to `¯2+2*31`
Default value: `16807`
Variable type: Implicit argument

The random number algorithm uses the value of ⎕RL in its calculation of a random number and sets ⎕RL to a new value after the random number is calculated.

```
      ⎕RL                    │        ⎕RL
16807                        │  282475249
      ?5                     │        ?5
1                            │  4
```

Because the random numbers selected by roll and deal are determined by an algorithm, they are not truly random numbers, but rather are pseudo-random numbers. A collection of them, however, satisfies many tests for randomness.

Repeatable results can be obtained from the functions roll (`?R`) and deal (`L?R`) if ⎕RL is first set to a particular value: For example, setting ⎕RL to `16807` and then entering `?5` returns a `1`, as in the previous example.

```
      ⎕RL←16807
      ?5
1
```

If ⎕RL is assigned an invalid value and then implicitly used by roll or deal, a `⎕RL ERROR` results.

A reference of ⎕RL yields its current value.

## ⎕SVC Shared Variable Control (Inquire)

| | |
|---|---|
| `Z←⎕SVC R` | Returns the access control vectors imposed on the variables named in R. |

R: Simple character scalar, vector, or matrix
Z: Simple Boolean vector or matrix

```
 ρZ ←→ (¯1↓ρR),4
ρρZ ←→ ,1⌈ρρR
```

Each row of R is interpreted as a variable name. Z contains a four-item access control vector for each corresponding variable name in R. The meaning of the items is given with ⎕SVC, shared variable control (set), page 324.

```
      ⎕SVC 'CTL102'
0 0 0 0
      102 ⎕SVO 'CTL102'
2
      ⎕SVC 'CTL102'
0 0 0 1
```

The access control vector `0 0 0 0` denotes either that no access control was set by either partner or that the variable has not been offered.

## ⎕SVC Shared Variable Control (Set)

`Z←L ⎕SVC R` Sets the protocol (access control vector `L`) regulating the sequences for the setting and use of the variable(s) `R` by the two partners and returns the resulting access control vector.

`L`: Simple Boolean scalar, vector, or matrix
`R`: Simple character scalar, vector, or matrix
`Z`: Simple Boolean vector or matrix

```
 ρZ ←→ (¯1↓ρR),4
ρρZ ←→ ,1⌈ρρR
```

Each row of `R` is interpreted as a variable name. Each row of `L` is interpreted as the corresponding access control vector for the name or names in `R`. The access control vector indicates whether repeated attempts to set or use a variable by one partner require an intervening use or set by the other.

```
      2001 ⎕SVO 'X'
1
      0 1 0 1 ⎕SVC 'X'
0 1 0 1
```

`Z` contains the resulting access control vectors imposed on each variable name in `R`. The resulting access control for each variable may be more restrictive than specified by `L` because a processor can only increase the degree of control imposed by the other processor.

```
      ⎕SVC 'CTL3'
0 0 0 1

      1 0 1 0 ⎕SVC 'CTL3'
1 0 1 1
```

Zeros in the access control vector are interpreted as no control imposed. Ones in the access control vector are interpreted as follows (the *first processor* refers to the user's processor; the *second processor* refers to the processor with which sharing is taking place).

| | |
|---|---|
| First Item | Two successive sets by the first processor require an intervening set or use by the second processor. |
| Second Item | Two successive sets by the second processor require an intervening set or use by the first processor. |
| Third Item | Two successive uses by the first processor require an intervening set by the second processor. |
| Fourth Item | Two successive uses by the second processor require an intervening set by the first processor. |

If a variable has a degree of coupling of 0, any specified access control vector results in an imposed access control vector of `0 0 0 0`.

***Posting Rules:*** A partner in sharing is not notified of your use of a shared variable unless that use is regulated.

***Conformability:*** `L` must be a matrix of shape `((¯1↓ρR),4)`, a vector of length 4, or a scalar. If `L` is a scalar or a vector, it is reshaped to `((¯1↓ρR),4)` before access control vectors are applied to `R`:

```
      1 ⎕SVC 'CTL'
1 1 1 1

      1 0 1 0 ⎕SVC 2 4ρ'CTL CTL3'
1 0 1 1
1 0 1 1

      L←2 4ρ1 1 0
      L
1 1 0 1
1 0 1 1

      L ⎕SVC 2 4ρ'CTL CTL3'
1 1 0 1
1 0 1 1
```

## ⎕SVE Shared Variable Event

| | |
|---|---|
| ⎕SVE←A | Specifies the amount of time in seconds to be used in a wait for a shared variable event and starts the timer. |
| X←⎕SVE | Suspends execution until the specified number of seconds has elapsed or a shared variable event occurs, as described below. When an event occurs, returns the time remaining in the timer. |

A: Simple nonnegative scalar
Default value: 0
Variable type: Localizing ⎕SVE has no effect.

***Assignment—Start the Timer:*** When ⎕SVE is assigned a positive value *n*, a countdown from *n* seconds begins.

***Use—Check for Events or Wait for One:*** If an event does not exist, execution is suspended until one occurs. The next section lists shared variable events.

If an event exists or if an event occurs during the suspension period:

- The use of ⎕SVE completes and the value of the timer is returned.
- All events are cleared, even those on shadowed variables.

For example, the function SHR waits for up to 5 seconds for an offer to be matched.

```
      ∇Z←AP SHR SHRVAR
[1]    ⎕SVE←5
[2]    ⍝ EXIT IF REJECT OR MATCH
[3]    TRY:→(1≠Z←AP ⎕SVO SHRVAR)/0
[4]    ⍝ WAIT FOR SHARED VARIABLE EVENT
[5]    →(0≠⎕SVE)/TRY
[6]    ∇
```

***Shared Variable Events:*** A shared variable event occurs when one of the following happens:

- An incoming offer to share a variable does not match a pending offer (made by you).
- Your partner matches a pending (or outstanding) offer from you or retracts a variable already shared.
- Your partner sets the access control vector (any dyadic ⎕SVC) on a fully-shared variable.
- The ⎕SVE timer expires.
- Your partner attempts to access a variable under the situations shown in Figure 35 on page 327. ACV used in the figure represents the left argument of ⎕SVC. Note that the access state vector may or may not change.

*Figure 35. Accesses to Variable That Signal a Shared Variable Event*

| **Constraints (Per ACV Settings)** | **Event Occurs If** |
|---|---|
| ACV[1] ←→ 1<br>(Two successive sets by *me* require an intervening set or use by *my partner*.) | *My partner* uses the variable, causing a change to the access state vector. |
| ACV[2] ←→ 1<br>(Two successive sets by *my partner* require an intervening set or use by *me*.) | *My partner* attempts to set the variable, but the specification cannot be completed because of access control constraint. This does not cause a change to the access state vector. |
| ACV[3] ←→ 1<br>(Two successive uses by *me* require an intervening set by *my partner*.) | *My partner* sets the variable, causing a change to the access state vector. |
| ACV[4] ←→ 1<br>(Two successive uses by *my partner* require an intervening set by *me*.) | *My partner* attempts to use the variable, but the use cannot be completed because of access control vector constraint. This does not cause a change to the access state vector. |

A shared variable event does **not** occur if:

- You specify ⎕SVE.
- Your partner uses one of the following inquiry system functions: ⎕SVO R, ⎕SVC R, ⎕SVQ R, ⎕SVS.

***Other Circumstances That Clear Events:*** In addition to all events being cleared when reference to ⎕SVE is completed, events for all variables are cleared when the active workspace is replaced using the commands )CLEAR, )LOAD, or )OFF.

Also an event for a single variable is cleared when:

- You set or use the variable.
- Your attempt to access the variable is unsuccessful because of something other than access constraints (for example, WS FULL).
- The variable is retracted (explicitly or implicitly).

You can explicitly clear all events by setting ⎕SVE to 0 and then using it:

```
      ⎕SVE←0
      ⎕SVE
0
```

## ⎕SVO Shared Variable Offer (Inquire)

`Z←⎕SVO R`   Returns the degree of coupling for the variables named in `R`.

`R`: Simple character scalar, vector, or matrix
`Z`: Integer scalar or vector in the set `0 1 2`

```
 ρZ ←→ ¯1↓ρR
ρρZ ←→ ,0⌈¯1+ρρR
```

Each row of `R` is interpreted as a variable name.

`Z` contains the degree of coupling for each corresponding variable name in `R`, as described in Figure 13 on page 61.

```
      211 ⎕SVO 'X'
1
      ⎕SVO 'X'
2

      R ← 2 3ρ'CTLDAT'
      124 ⎕SVO R
1 1
      ⎕SVO R
2 2
```

## ⎕SVO Shared Variable Offer (Set)

`Z←L ⎕SVO R` Offers variables named in `R` to processors identified in `L`. The result is the degree of coupling, indicating whether the attempt to share was successful:

0 - Unshared
1 - Offered
2 - Shared (coupled)

`L`: Simple integer scalar or vector
`R`: Simple character scalar, vector, or matrix of names
`Z`: Integer scalar or vector in set `0 1 2`

```
 ρZ ←→ ¯1↓ρR
ρρZ ←→ ,0⌈¯1+ρρR
```

Each row of `R` is interpreted as a variable name. Each integer in `L` identifies the corresponding processor for a variable name in `R`.

```
      127 ⎕SVO 'CTL'
1
```

**Note:** The variable is not fully coupled until the processor with whom you have offered to share counters your offer with an offer of a variable of the same name.

If the degree of coupling is 1 or 2, a repeated offer has no further implicit result and either monadic or dyadic `⎕SVO` can be used for inquiry.

***Conformability:*** `L` must have the same number of items as `R` has rows (`ρL ←→ ¯1↓ρR`) or `L` can be a scalar, in which case it is reshaped to `¯1↓ρR` before the shared variable offer is attempted. The following offers two variables to AP 211:

```
      211 ⎕SVO 'VAR1'
1
      211 ⎕SVO 'VAR2'
1
```

The same offers can also be made with the expression:

```
      211 ⎕SVO 2 4⍴'VAR1VAR2'
1 1
```

Or with:

```
      211 ⎕SVO¨ 'VAR1' 'VAR2'
1 1
```

***Surrogate Names:*** To maintain compatibility between two independent processors, APL2 permits the use of an alias name for a shared variable. This alias is called the *surrogate name*. The surrogate name is the name by which the variable is known to the processor to which the offer is being made.

When a row of `R` contains a pair of names (separated by a space), the first name in the pair represents the name of the variable to be shared and the second is the name by which the variable is known by your partner. For example:

```
      124 ⎕SVO 'MYCNTL C124'
```

AP 124 (the APL2 text display auxiliary processor) requires its control variable to start with `C`. The variable name used in the APL2 operation that offers the variable to AP 124, however, is `MYCNTL`.

(The name of a variable may be its own surrogate, which is the default when no surrogate name is specified.)

***General Share Offer:*** A share offer to processor `0` is interpreted as a general share offer to any available processor. A general offer is coupled by the first specific offer to the caller from any processor of a variable with the same name. General offers are not coupled with general offers, and a general offer does not cause a shared variable event to occur.

## ⎕SVQ Shared Variable Query

| | |
|---|---|
| `Z←⎕SVQ R` | Identifies processors making share offers or returns the names of variables being offered by an identified processor but not yet matched by you. |

*R*: Simple integer scalar or one-item vector or empty vector
*Z*: Integer vector or a character matrix

`ρZ` ←→ unpredictable
`ρρZ` ←→ `,1` if *R* is empty; otherwise `,2`

*R* is an empty vector or contains the identification of a processor.

If *R* is an empty vector, *Z* is an integer vector of identifications for processors making share offers to you. For example:

```
      126 ⎕SVO 'CTL126'
2
      ⎕SVQ ⍳0
126
```

If *R* is not empty, *Z* is a character matrix containing the names of variables not yet shared but being offered by the processor identified in *R*. For example:

```
      ⎕SVQ 126
DAT126
```

## ⎕SVR Shared Variable Retraction

`Z←⎕SVR R` Requests retraction of each shared variable named in `R` and returns its prior degree of coupling.

`R`: Simple character scalar, vector, or matrix
`Z`: Simple integer scalar or vector

```
 ρZ  ←→  ¯1↓ρR
ρρZ  ←→  ,0⌈¯1+ρρR
```

Each row of `R` is interpreted as a variable name. `Z` contains the degree of coupling prior to the retraction for each corresponding variable name in `R`. Figure 13 on page 61 defines each degree of coupling. After a variable is retracted, it is no longer shared. (Its degree of coupling is less than `2`.) For example:

```
      CTL←''
      ⎕SVO 'CTL'
2
      ⎕SVR 'CTL'
2
      ⎕SVO 'CTL'
0
```

Multiple variables can be retracted in the same statement using the each (¨) operator or a character matrix. For example, the following two retractions have the same effect:

```
      ⎕SVR¨ 'CTL123' 'DAT123'
2 2

      ⎕SVR 2 6ρ'CTL123DAT123'
2 2
```

All shared variables can be retracted using the system function `⎕NL` (name list), page 313:

```
      ⎕NL 2
A
B
C
CTL124
DAT124

     ⎕SVR ⎕NL 2
0 0 1 2 2
```

`A` and `B` in the example are not shared variables. Variable `C` had been shared, but the share had not been matched by the partner at the time of the retract. `CTL124` and `DAT124` had been fully-coupled shares.

If a shared variable has no value when retracted, it does not persist in the workspace after retraction.

***Implicit Retraction of a Shared Variable:*** A variable may be retracted implicitly by any of the means listed below:

- You use )ERASE or ⎕EX (expunge) to delete a shared variable from your active workspace.
- You exit from a defined function that has the shared variable declared as a local variable.
- You use )COPY to copy a variable with the same name as a currently shared variable.
- You use )IN or ⎕TF (transfer form) to establish a variable or function with the same name as a currently shared variable.
- You use one of the following system commands:
  - )CLEAR
  - )LOAD
  - )OFF
  - )CONTINUE

## ⎕SVS Shared Variable State

`Z←⎕SVS R`  Returns the access states of each variable named in `R`.

`R`: Simple character scalar, vector, or matrix
`Z`: Simple Boolean vector or matrix

```
 ρZ ←→ (¯1↓ρR),4
ρρZ ←→ ,1⌈ρρR
```

The access state vector indicates which partner knows the current value and which partner last set a value unknown to the other partner.

Each row of `R` is interpreted as an APL2 name (variable name). `Z` contains a four-item vector of access states for each corresponding variable name in `R`.

A vector of access states may have one of the following four values. (*First processor* refers to the user's processor. *Second processor* refers to the processor with which sharing is taking place.)

| | |
|---|---|
| `0 0 0 0` | Not a shared variable. |
| `0 0 1 1` | Set by one of the processors and used by the other. Also signifies the initial state before either partner sets a value. |
| `1 0 1 0` | Set by the *first processor*, not yet used by the second. |
| `0 1 0 1` | Set by the *second processor*, not yet used by the first. |

For example:

```
      ⎕SVO 'CTL'
2
      1 0 0 1 ⎕SVC 'CTL'
1 0 0 1
      CTL←'SOMETHING'
      ⎕SVS 'CTL'
0 1 0 1
      RETURN←CTL
      ⎕SVS 'CTL'
0 0 1 1
```

## ⎕TC Terminal Control Characters

⎕TC Contains a three-item character vector of terminal control characters:

⎕TC[1]—backspace
⎕TC[2]—new line (return)
⎕TC[3]—line feed

Variable type: Specifying or localizing ⎕TC has no effect.

References of items of ⎕TC cause the terminal to display the corresponding character. Use ⎕TC rather than ⎕AV to avoid system dependencies (because the order of ⎕AV is different in different APL implementations).

***Backspace:*** On display terminals, the character specified after ⎕TC[1] replaces the character specified before it. On typewriter-like terminals, the second character overstrikes the first character.

**Display**

```
      '∇',⎕TC[1],'|'
|
```

**Typewriter**

```
      '∇',⎕TC[1],'|'
⍒
```

***New Line:*** After ⎕TC[2], the cursor or print element carrier is positioned at the left margin of the next line.

```
      'NEW',⎕TC[2],'LINE',⎕TC[2],'CHARACTER'
NEW
LINE
CHARACTER
```

***Line Feed:*** After ⎕TC[3], the cursor or print element carrier is positioned on the next line at the same column position.

```
      'LINE',⎕TC[3],'FEED',⎕TC[3],'CHARACTER'
LINE
    FEED
        CHARACTER
```

## ⎕TF Transfer Form

`Z←L ⎕TF R` Creates the transfer form as specified in `L` of a variable, displayable defined operation, or external object named in `R`.

Or can establish an object in the active workspace from `R`, which is of transfer form `L`.

`L`: Simple integer scalar or one-item integer vector
`R`: Simple character scalar or vector
`Z`: Simple character vector

`L` contains an integer (`1` or `2`) that specifies either the migration transfer form or the extended transfer form:

The *migration transfer form* (`L` is `1`) represents the name and value of a simple and nonmixed variable or a displayable defined function. It is not permitted for nested or mixed variables or defined operators.

The migration form vector consists of four parts:

1. A data type code header character:

   `'F'` for a function
   `'N'` for a simple numeric array
   `'C'` for a simple character array

2. The object name, followed by a blank.
3. A character representation of the rank and shape of the array, followed by a blank.
4. A character representation of the array items in row-major order (any numeric conversions are carried to 18 digits).

The *extended transfer form* (`L` is `2`) is a simple character vector that represents the name and value of a variable, a displayable defined function or operator, or an external object. It is permitted for any variable and displayable defined operation.

### Creating the Extended or Migration Transfer Form

See Appendix B, "APL2 Transfer Files and Extended Transfer Formats" on page 484 for further details about transfer form.

***Of a Variable or Defined Operation:*** If `R` is the name of a variable or displayable defined operation, `Z` is a character vector that is the transfer form specified in `L` for that object.

### Example 1: Transfer Forms of a Function

```
      ∇Z←ITEMS R
[1]   Z←1
[2]   →(0∊⍴R)/0
[3]   Z←×/⍴R
[4]   ∇

⍝ MIGRATION TRANSFER FORM

      Z←1 ⎕TF 'ITEMS'
      ⍴Z
49
      Z
FITEMS 2 4 9 Z←ITEMS RZ←1      →(0∊⍴R)/0Z←×/⍴R

⍝ EXTENDED TRANSFER FORM

      Z←2 ⎕TF 'ITEMS'
      ⍴Z
42
      Z
⎕FX 'Z←ITEMS R' 'Z←1' '→(0∊⍴R)/0' 'Z←×/⍴R'
```

### Example 2: Transfer Forms of a Simple Variable

```
       A←''                  A←2 3⍴1⌽⍳6
       1 ⎕TF 'A'             1 ⎕TF 'A'
 CA 1 0                NA 2 2 3 2 3 4 5 6 1
       2 ⎕TF 'A'             2 ⎕TF 'A'
 A←''                  A←2 3⍴2 3 4 5 6 1

       A←' Don''t '          A←.000000000001
       1 ⎕TF 'A'             1 ⎕TF 'A'
 CA 1 7  Don't         NA 0 1E¯12
       2 ⎕TF 'A'             2 ⎕TF 'A'
 A←' Don''t '          A←1E¯12
```

### Example 3: Transfer Forms of an APL2 Variable

```
        A←('' (⍳0))('Q' 3.2)(2+3×⍳4) 'Don''t'
        B←⊂(⊂1 0 1) (2 3⍴4 6)
        C←⎕AF 256⊥¨(⊂300 66),¨⎕AF 'AB'  ⍝ Kanji
        DISPLAY A
  .→----------------------------------------.
  | .→--------. .→----. .→--------. .→----. |
  | | .⊖. .⊖. | |Q 3.2| |5 8 11 14| |Don't| |
  | | | | |0| | '+----' '~--------' '-----' |
  | | '-' '~' |                             |
  | '∊--------'                             |
  '∊----------------------------------------'
        2 ⎕TF 'A'
  A←(''(0⍴0))('Q' 3.2)(5-3×⎕IO-⍳4)'Don''t'
```

```
      DISPLAY B
.-------------------------.
| .→--------------------. |
| | .---------. .→----. | |
| | | .→----. | ↓4 6 4| | |
| | | |1 0 1| | |6 4 6| | |
| | | '~----' | '~----' | |
| | '∊--------'         | |
| '∊--------------------' |
'∊------------------------'
      2 ⎕TF 'B'
B←⊂(⊂1 0 1)(2 3⍴4 6 4 6 4 6)
      DISPLAY C
.→-.
|ωω|
'--'
      2 ⎕TF 'C'
C←⎕AF 19677889 19677890
```

**Note:** `1 ⎕TF` is not supported for any of these cases, and returns an empty result.

**Example 4: Transfer Form of an External Object**

```
      000 11.0 ⎕NA'see DISPLAY'
1
      1 ⎕TF 'see'   ⍝ No migration form

      2 ⎕TF 'see'   ⍝ Extended transfer form
0 11 ⎕NA 'see DISPLAY'
```

***Of a System Variable:*** If *R* contains the name of a system variable, *Z* contains the transfer form specified by *L* of the variable at its current value:

```
      2 ⎕TF '⎕TS'
⎕TS←1992 3 27 14 34 4 724
```

***Of a System Function:*** If *R* contains the name of a system function, *Z* is an empty character vector. (System functions are not displayable.)

```
      Z←2 ⎕TF '⎕DL'
      ⍴Z
0
```

***Of a Shared Variable:*** If *R* is the name of a shared variable, creating its transfer form constitutes a reference of the variable. The value of *Z* depends upon the value of *L* and the value of the variable at the time of the reference:

```
      101 ⎕SVO 'CTL'
2
      2 ⎕TF 'CTL'
CTL←0
```

## Creating the Inverse Transfer Form

If R is the transfer form specified by L of a variable or defined operation, that variable or defined operation is established in the active workspace. Z is a simple character vector containing the object's name. Such use of ⎕TF is known as the *inverse transfer form*.

```
      )CLEAR
CLEAR WS
      SCORES←34 18 20
      R←1 ⎕TF 'SCORES'
      )VARS
R       SCORES
      )ERASE SCORES
      )VARS
R

      1 ⎕TF R
SCORES

      )VARS
R       SCORES
```

If the transfer form in R is invalid, Z is an empty character vector (`''`).

## ⎕TS Time Stamp

|   |   |
|---|---|
| ⎕TS | Contains the current system date and time. |

Variable type: Localizing or specifying ⎕TS has no effect.

The time stamp ⎕TS is a simple integer vector composed of the following seven items:

|   |   |
|---|---|
| ⎕TS[1] | Current year |
| ⎕TS[2] | Current month |
| ⎕TS[3] | Current day |
| ⎕TS[4] | Current hour |
| ⎕TS[5] | Current minute |
| ⎕TS[6] | Current second |
| ⎕TS[7] | Current millisecond |

The value of ⎕TS is offset from Greenwich Mean Time (GMT) according to the value of the Time Zone system variable (⎕TZ), page 341.

```
      ⎕TS
1992 3 27 21 6 43 251
      ⎕TZ
¯4
      ⎕TZ←¯10
      ⎕TS
1992 3 27 15 6 48 247
```

Use format by example (⍕), page 139, to display the date and time in different formats.

```
      '0006/06/00 06:06:06:000'⍕⎕TS
1992/03/27 08:12:30:548

      '06/06/00 06:00'⍕100|⎕TS[2 3 1 4 5]
03/27/92 08:12
```

# ⎕TZ Time Zone

| | |
|---|---|
| ⎕TZ←A | Specifies or references the offset in hours between local time and Greenwich Mean Time (GMT). |

A: Simple real scalar
Variable type: Session
Default value: Installation-dependent

The value of ⎕TZ affects the current hour reported by ⎕TS (page 340). ⎕TZ must be in the range ¯12 ⎕TZ and ⎕TZ 12. For example, ¯5 is Eastern Standard Time, and 1 is Central European Standard Time. Although usually an integer, the value associated with ⎕TZ may be a fraction.

```
      ⎕TZ
¯4
      ⎕TS
1992 3 27 8 35 53 829

      ⎕TZ←5
      ⎕TS
1992 3 27 17 35 56 926
```

⎕TZ affects the time stamp reported by the system commands )CONTINUE, )COPY, )DROP, )LOAD, )PCOPY, )SAVE, and )TIME; it also affects the time stamp reported for defined functions and operators in 3 ⎕AT R or when displayed.

An invalid value assigned to ⎕TZ is ignored.

```
      ⎕TZ←12.5
      ⎕TS
1992 6 28 17 36 11 931
```

## ⎕UCS Universal Character Set

`Z←⎕UCS R` Converts integers to characters and characters to integers using the ISO 10646 standard, which includes the Unicode subset.

`R` and `Z`: A simple numeric integer array or a simple character array

Integers in `R` must be nonnegative and less than `2*31`.

`⎕UCS` on characters produces the integer that specifies the character position in the universal character set given in Figure 71 on page 475. These numbers are platform independent.

`⎕UCS` on numbers produces the corresponding character.

```
      ⎕UCS 'ρA B'
9076 65 32 66
```

## ⎕UL User Load

| | |
|---|---|
| ⎕UL | Contains the number of users on a system where that number can be determined. |

Variable type: Localizing or specifying ⎕UL has no effect.

⎕UL is a simple nonnegative integer scalar. Its value is `0` on systems in which the number of users cannot be determined.

## ⎕WA Workspace Available

| | |
|---|---|
| ⎕WA | Contains the number of available bytes in the active workspace. |
| Variable type: Localizing or specifying ⎕WA has no effect. | |

⎕WA is a simple nonzero integer scalar. Depending on the APL2 implementation, the value of ⎕WA can vary between two situations that appear to be the same.

# Chapter 7. Defined Functions and Operators

This chapter discusses functions and operators in terms of:

- Structure
- Definition contents
- Execution
- Debug controls

Many problems can be solved by merely entering APL2 expressions in immediate execution mode. However, when a series of expressions needs to be entered repeatedly in different situations, when a general solution can be applied to several similar problems, or when expressions should be executed based on certain conditions, you may prefer to *define* an operation (a function or operator) to hold the necessary code.

A defined function or operator is *fixed* or established in the active workspace in one of the following ways:

- Defined, using one of the APL2 editors. (The editors are discussed in Chapter 9, “The APL2 Editors” on page 375.)
- Fixed, using the system function `⎕FX` or `⎕TF`, which changes a character representation of the operation to an executable form. (`⎕FX` is discussed in “`⎕FX` Fix (No Execution Properties)” on page 292 and “`⎕FX` Fix (with Execution Properties)” on page 294, and `⎕TF` is discussed in “`⎕TF` Transfer Form” on page 336.)
- Copied, using one of the copy system commands—`)COPY` or `)PCOPY`—brought into the active workspace as a result of the system command `)LOAD`, or retrieved from a transfer file using the system commands `)IN` or `)PIN`. (These system commands are discussed in “Storing and Retrieving Objects and Workspaces” on page 414.)

When a defined function or operator is *invoked*, the statements in it are executed. For example, the defined function `ROUND` shown below rounds a number to a specified number of decimal places. If no number of places is indicated, two places are assumed.

```
      3 ROUND 45.678235     |        ROUND 45.678235
45.678                      |  45.68
```

The syntax and execution of `ROUND` are similar to those of a primitive function. The definition of `ROUND` is shown and commented upon in Figure 36 on page 346.

```
      ∇ROUND[⎕]∇
    ∇
[0]   Z←Y ROUND X
[1]   →(0≠⎕NC 'Y')/RN          ⍝ GO TO RN IF Y HAS VALUE
[2]   Y←2                      ⍝ SET A VALUE FOR Y
[3]  RN:Z←(10*-Y)×⌊.5+X×10*Y   ⍝ ROUND X TO Y PLACES
    ∇    1993-03-27  7.47.10 (GMT-4)
```

When a definition is displayed using )EDITOR 1, the display begins and ends with a *del* (∇) and includes line numbers. Alternatively, a definition can be listed using the system function `⎕CR` (character representation), which does not show line numbers or dels (`⎕CR` is discussed on page 274).

`[0]` The *header* establishes the syntax of the function. It shows the number of arguments the function takes and the parameter names of the arguments and result within the definition.

`[1]` The *branch statement* directs the flow of control to statement `[3]` (label `RN`) if a left argument is entered when `ROUND` is invoked.

`[2]` Statement sets the left argument to `2` if `ROUND` is invoked without a left argument. Statements `[1]` and `[2]` are defined so that `ROUND` is ambi-valent.

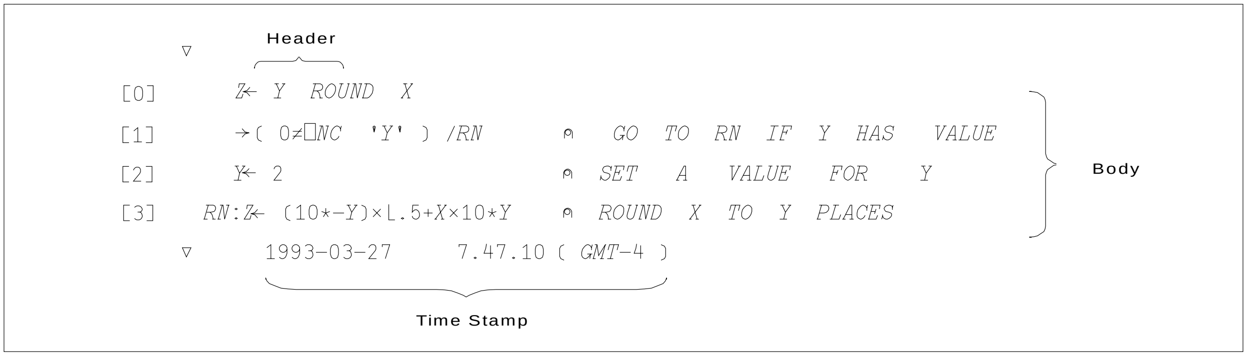
`[3]` Rounds the argument as specified. It begins with a *label* to identify the statement.

**Note:** The defined function `ROUND` shows no validity checking to ensure that conformable numeric arguments are entered. Additional statements to handle these checks can be added.

*Figure 36. An Example of a Defined Function*

## Structure

Defined functions and operators have three parts, as illustrated in Figure 36 for the `ROUND` function.



```
      ∇
[0]       Z← Y ROUND X
[1]       →( 0≠⎕NC 'Y' ) /RN      ⍝  GO TO RN IF Y HAS VALUE
[2]       Y← 2                    ⍝ SET A VALUE FOR Y
[3]    RN:Z← (10*-Y)×⌊.5+X×10*Y   ⍝ ROUND X TO Y PLACES
      ∇     1993-03-27    7.47.10 ( GMT-4 )
```

*Figure 37. The three parts of defined functions and operators*

Each of these parts is explained in this section.

## Header

The operation *header* is the first line of a defined operation. The header establishes the syntax for the defined operation, including:

- Name of the operation
- Valence of the operation, and in the case of defined operators, also the valence of the derived function
- Parameter names
- Nature of the result—explicit or not explicit
- Local names

“Defined Functions and Operators” on page 31 shows the possible types of headers for defined functions and operators and discusses how defined functions and operators are used in expressions.

### Name of Operation

The name of the defined function or operator is a user name and follows the rules for constructing names (see “Rules for Constructed Names” on page 25). If a name is already in use as a variable name, an attempt at operation definition generates a *DEFN ERROR*. An attempt to specify as a variable a name already in use as the name of a defined operation generates a *SYNTAX ERROR*.

### Valence

A defined function can have two, one, or no arguments. A defined operator can have two or one operands, and its derived function can have two or one arguments. If two arguments are shown in a defined function, the function is ambivalent (it can be invoked with either one or two arguments), and the possibility of receiving only a right argument should be accounted for in its definition. Defined operators, like primitive operators, are never ambi-valent with respect to their operands.

### Parameter Names

The result, right argument, left argument, right operand, and left operand named in the header are *parameters*, used in the body of the definition. When the operation is invoked, the values entered as arguments and/or operands are associated with the parameter names. This is how the parameter names are first associated with values. During the course of execution of the operation, the parameter names can be associated with other values.

When execution of the operation is completed, the value of the result parameter is returned as the result of the function.

### Local Names

The argument, operand, and result parameters have value only within the context of the defined operation. When execution of the operation is completed, the parameter names are no longer associated with the values they had during execution. Thus, they are called *local names* because their values are local to—exist only within—the defined operation.

In addition to parameter names, you can declare other constructed names to be local to the operation. These can be names that hold intermediate values, set counters, set system variables especially for the operation, or are otherwise not needed after the operation has been executed. Labels are also local names.

All other names identify global objects in the workspace and are called *global names*. The value associated with a global name is not available during execution of the operation if the same name is local to that operation. Only the local value is available, and the global name is said to be *shadowed*. After the operation has been executed, however, only the global value exists in the workspace. For example:

```
      Z←'TEST'
      X←2000
      ∇Z←FN X
[1]   X
[2]   Z←X×3
[3]   ∇

      FN 5
5
15
      Z
TEST
      X
2000
```

In the operation header, a semicolon separates local names from the operation syntax and from each other. For example, to declare `⎕IO`, `I`, and `CNT` local to the monadic function `FN`:

```
∇Z←FN X;⎕IO;I;CNT
```

Separating local names by semicolons is optional on input. On display, semicolons are always included in the operation header.

Objects identified by global names are available to a called operation if they are not shadowed by the local names in it.

“Use of Local Names” on page 353 further discusses the use of local names in the operation definition.

## Body

The definition body is made up of *statements*, which, as described in “Expressions” on page 27, can include any of the following:

- Label
- Expressions
- Comment

These take the form:

label **:** expressions ⍝ comment

The `ROUND` function in Figure 36 on page 346 shows various forms of statements.

## Time Stamp

Each defined operation is associated with a time stamp that identifies when the operation was last fixed in the workspace. An operation is fixed in the active workspace by using the system function ⎕*FX* or ⎕*TF* or by using one of the APL2 editors to create or modify an operation. When the defined operation is displayed with )EDITOR 1, its time stamp and the offset from Greenwich Mean Time (of the current session) are also displayed.

The system function ⎕*AT* (attributes), page 270, can be used to determine the time stamp of an object without displaying it.

## Definition Contents

Within the body of a defined operation, you can use any APL2 statement. Direct entry of system commands or editor commands to be executed as part of the defined operation is not permitted. Flow of control within a defined operation is sequential, from the first statement to the last, except as altered by *branching*.

A defined operation may invoke another defined operation. It is thus possible to write modular applications and easily reuse defined functions and operators in different applications. A defined operation may also invoke itself. This is called *recursion*.

Defined or primitive operators may be combined with defined or primitive functions to produce derived functions.

## Branching

A branch expression explicitly determines the next line of a defined function or operator to be executed. It consists of a branch arrow (→) and an expression:

```
→expression
```

Figure 38 shows the possible branch actions according to the value of the branch expression.

*Figure 38. Action Based on Value of Branch Expression within Defined Functions*

| **If Value of Branch Expression Is...** | **Then Next Action Is...** |
|---|---|
| Line number *N* within the function or operator | Line *N* of the function or operator is executed. |
| 0 or any other line number not within the function or operator | Flow of execution returns to the invoking expression. |
| Empty vector | Next sequential expression (either the next expression to the right of a diamond in the same line, or else to the next line, if there is one, of the function or operator). |
| Vector of numbers | The first number determines the branch action. |

For example, the `ROUND` function, shown earlier, contains the branch statement:

```
[1]    →(0≠⎕NC 'Y')/RN     ⍝GO TO RN IF Y HAS VALUE
```

The result of executing this statement is to branch to the line labeled `RN` if a parameter `Y` has been associated with a value. Otherwise, function execution continues with the next sequential statement.

Branching is also used in immediate execution to resume execution of a suspended immediate execution statement, defined function, or defined operator. This use of branching is discussed in “Errors and Interrupts in Immediate Execution” on page 59 and “Clearing the State Indicator” on page 357.

**Note:** Executed branches are discussed under the heading “⍎ Execute” on page 120.

## Labels

If you branch to explicit line numbers, you have to review the branch expressions and edit them every time you make a change to a function or operator. Using *labels* avoids these steps. A label is a name that precedes an expression:

```
[3]    RN:Z←(10*-Y)×⌊.5+X×10*Y
```

A label is a local constant; that is, it has meaning only within the context of the function or operator. The value of the label is the line number with which it is currently associated. If the line number changes, so does the value of the label.

Any branch expression whose result is expressed as a label takes the value of the label—the line number currently associated with the label. Thus, when you edit a defined function or operator and add or delete lines, your branch expressions always point to the correct line. If you use line numbers instead of labels in branch expressions, you must check every branching expression to ensure that it still points to the correct line.

**Always use labels when branching to a line in the operation.**

## Conditional Branch

When a branch expression takes different values depending on relationships or conditions, the branch is called a *conditional branch*. It is constructed by using `→` with relational and selection operations.

The statement `→(0≠⎕NC 'Y')/RN` is a conditional branch statement because its value may be `RN` or the empty vector, depending on the value of the relationship in parentheses.

Conditions for branch expressions evaluate to `0` or `1`. The relational functions (`< ≤ = ≥ > ≠` ) are often used to express simple conditions.

Figure 39 shows three frequently used conditional branch expressions. In each case, the *condition* evaluates to `0` or `1`.

*Figure 39. Frequently Used Branch Statements*

| Form | Description |
|---|---|
| →( *condition* ) / 0 | End the function or operator execution if the relationship is true.<br>Execute the next sequential expression if the relationship is false. |
| →( *condition(s)* ) / *label(s)* | Continue execution at the labeled line if the relationship is true.<br>Execute the next sequential expression if the relationship is false.<br>Any number of conditional expressions can be used as long as there are the same number of labels. |
| →*label*×*condition* | Execute the labeled line if the relationship is true.<br>End the function or operator execution if the relationship is false. |

**Note:** Compression is the most commonly used operation in constructing branch expressions. It works equally well for a one- or several-way branch. It is not origin dependent.

## Unconditional Branch

When the branch expression contains a single constant or label name, it is called an *unconditional* branch.

Unconditional branches have two main uses:

- End the function execution by branching to line zero (→0), a line outside the function.
- Create a branch back to the beginning of a loop.

## Branch to Escape

A branch arrow with no expression on the right causes the defined operation to immediately terminate. Any functions pendent on this one are also terminated. See "⎕*EC* Execute Controlled" on page 280 for an exception.

## Branch in a Line with Diamonds

When a branch expression is one of several expressions separated by diamonds the following possibilities exist:

- If the branch is taken, expressions to the right of the branch expression are not evaluated.
- If the branch is not taken, execution continues with the expression to the right of the branch expression.

### Looping Is Rarely Needed

Many programmers who come to APL2 after using other languages structure their function and operator definitions with the equivalent of DO loops, working with data an item at a time. This approach is expensive in performance and introduces the likelihood of programming errors.

Looping requires APL2 to interpret each expression in the loop each time it is evaluated. For efficient use of system resources and programs that are easier to debug and maintain, looping should be avoided whenever possible.

APL2's array processing and operators help you avoid most looping. Array operations are entirely data-driven. They allow computations to be performed where the data itself controls the limits of the operation. Summation (+/), for instance, is controlled only by the data being summed. Loop control statements such as DO and IF THEN ELSE are not needed. You can do arithmetic on entire collections of numbers in a single operation.

In a practical sense, the operator each (¨), pages 109 and 107, is the equivalent of a DO loop, except that the loop limits are not explicitly mentioned, but instead are implicit in the data.

## Structuring Ambi-valent Functions

Primitive, defined, and derived functions may be called with either one or two arguments. (Defined functions may also be called with no arguments.) If a primitive function or a derived function does not have a monadic definition, a *VALENCE ERROR* is generated if it is used without a left argument. If a dyadic defined function or a function derived from a defined operator has not accounted for the possibility of a monadic call in its definition and subsequently references the missing argument, a *VALUE ERROR* is generated.

To define an ambi-valent function or derived function, you can define a conditional branch to the code that executes the appropriate version of the program. Or you can define a default value for an argument when one is not supplied. For example, the function *ROUND*, shown in Figure 36 on page 346, supplies a default value to take effect if no left argument is entered.

If you do not want the function to have a monadic definition, you can give a *VALENCE ERROR* message by using ⎕*ES*. Figure 40 on page 353 shows a way of providing such a message.

## Event Handling

APL2 provides two system functions and two system variables that allow user handling of error conditions:

- System functions

  ⎕*EA*—Execute alternate, page 278
  ⎕*EC*—Execute controlled, page 280
  ⎕*ES*—Event simulate, pages 282 and 285

- System variables

  ⎕*EM*—Event message, page 281
  ⎕*ET*—Event type, page 287

For example, when you simulate an error with ⎕*ES*, the defined function performs as if it were a primitive function. An APL2 error report is generated and the message displays a caret (∧) to mark the error. Suspension then occurs at the calling point, not within the defined function. Figure 40 shows how ⎕*ES* can be used to simulate a *VALENCE ERROR* in the *ROUND* function.

```
      ∇Z←Y ROUND2 X
[1]   ⎕ES(2≠⎕NC 'Y')/5 1  ⍝SIGNAL VALENCE ERROR
[2]   Z←(10*-Y)×⌊.5+X×10*Y ⍝ROUND X TO Y PLACES
[3]   ∇

      3 ROUND2 4.5677887
4.568

      ROUND2 4.5677887
VALENCE ERROR
      ROUND2 4.5677887
      ∧
```

*Figure 40. Example Use of Event Simulation*

## Use of Local Names

Localizing names is a way of controlling the value of those names. They can have no values other than those assigned within the defined operation. For example, if a defined function depends on particular settings of system variables, such as ⎕*FC* (format control) for reports, ⎕*IO* (index origin), or ⎕*CT* (comparison tolerance) for data analysis, then these variables can be declared as local. Execution of the operation is not affected by their global values, and the global values are not affected by execution of the operation.

Names that are needed within a defined operation but have no importance after the operation is executed should be localized. These names appear as global to operations called from this one. If they are not, execution of the operation creates them as global names whose values persist in the workspace and take up space until they are explicitly erased. Specification of values may destroy the values of global objects with the same name.

## Execution

When a defined operation appears in an expression, it is evaluated in the context of that expression following the evaluation rules in “Evaluating Expressions” on page 32. The execution of the operation is controlled by its definition and its execution properties, which affect the operation's behavior in error or interrupt situations (see “Execution Properties” on page 360).

Each statement in the definition is executed in sequence or as directed by branching statements. If the function has been defined with an explicit result, the last specification of the result parameter name is returned as the result of executing the operation. This result is then available for further evaluation of the expression in which the defined operation appears.

## Suspension of Execution

Execution of a defined operation may be suspended in either of two ways:

- By an attention
- By an interrupt or an error

You can suspend execution of a defined operation (or an expression) through the keyboard in one of two ways: attention or interrupt. An *attention* suspends execution at the end of the current statement being executed. An *interrupt* causes the system to behave as though an error were encountered; it suspends execution immediately. All discussions concerning the effects of errors and their handling apply to interrupts as well.

Attention signals and interrupts differ among input devices and host systems. For information on attention and interrupt for your system, see the appropriate workstation user's guide or *APL2/370 Programming: System Services Reference*.

If an error is encountered in a statement during execution of the defined operation or if an interrupt is signaled, execution of the operation is suspended, and a message and the suspended operation are displayed. For example:

```
      ∇Z←F X
[1]   Z←10÷X
[2]   ∇

      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^  ^
```

## Calling Sequence

If a statement in a defined operation contains the name of a defined function or operator, that operation is called and flow of control passes to it. While the called operation is executing, the calling operation is said to be *pendent*, waiting to complete execution. If the called function or operator, in turn, calls another, it is pendent along with the original calling operation. Figure 41 on page 355 illustrates this flow of control from one operation to another for the calling sequence beginning with `FUNCTIONA`.

```
FUNCTIONA                Original function begins executing.
 .
 .
 .
[AN]   FUNCTIONB         FUNCTIONB called.
                         FUNCTIONA is pendent.
 .
 .
 .
[BN]   OPERATORA         OPERATORA called.
                         FUNCTIONB is pendent.
 .
 .                       OPERATORA completes execution.
 .                       Control returns to FUNCTIONB.
 .
[BN]  FUNCTIONB          FUNCTIONB resumes execution at the
                         expression immediately after where it.
 .                       was called. When FUNCTIONB completes,
 .                       control returns to FUNCTIONA.
 .
[AN]  FUNCTIONA          FUNCTIONA resumes and completes execution.
```

*Figure 41. Flow of Control of Calling and Called Operations*

As each operation (or immediate execution expression) is invoked, it is placed in the *execution stack*, and the line currently being executed is placed in the *state indicator*. When the line completes, it is removed from the state indicator. When execution of an operation or expression completes, it is removed from the execution stack.

The number of called functions or operators is not limited except as constrained by the space available within the workspace. Pendent operations take up space; a sequence of called and calling operations may create a *WS FULL* condition if there is a large number of them or if any of them requires a sizable work area for calculation.

A function that calls itself is *recursive*. Local copies of the function behave as separate functions in the execution stack. When a recursive function is called from an operator, its name may be shadowed by local names.

## State Indicator

When an operation is suspended, the suspended statement and its calling sequence are found in the *state indicator*. The state indicator is a list of:

- The calling sequence of defined functions and operators along with their calling line numbers and associated statements (see Figure 41).
- Asterisk(s) and the associated expression for all immediate execution expressions that did not complete, either because of an error in the expression or because the function invoked by the expression is pendent or suspended.
- Defined functions or operators that are in definition mode, with the statement currently being defined.

The system commands `)SIS` (page 453), `)SI` (page “`)SIS`—Display the State Indicator with Statements” on page 457 form=pageonly), and `)SINL` (page “`)SINL`—Display the State Indicator with Name List” on page 456 form=pageonly) display the state indicator and information about its contents:

- `)SIS` displays each statement in the state indicator with one or two carets to indicate how far evaluation of the statement proceeded before it was stopped.
- `)SI` displays statement numbers and asterisks.
- `)SINL` displays local names for each defined operation in the state indicator.

For example:

```
       )SIS                 )SI              )SINL
F[1] Z←10÷X          F[1]             F[1]     Z         X
       ^  ^          *                *
*   F 0
    ^
```

While an operation is suspended, local names are available for inspection. However, any global values associated with those names are shadowed.

You can also use the system variables ⎕L (left argument) and ⎕R (right argument) to help determine the source of the error.

Figure 42 on page 357 shows an example of actions that add to the state indicator and the resulting response to the `)SIS` command.

```
      )SIS          State indicator is empty.
      3⍳            Error adds statement to
SYNTAX ERROR        state indicator.
      3⍳
      ^^

      )SIS
*   3⍳              Asterisk indicates immediate
    ^^              execution expression that did
                    not complete.

      ∇Z←FN         Function definition.  Note call
[1]   'LINE 1'      of function GN at line 2.
[2]   Z←GN×2
[3]   'LINE 3'∇

      ∇Z←GN         Function definition. Note error.
[1]   Z←3÷0∇

      FN            Function invoked.
LINE 1              First line of function executes.
DOMAIN ERROR        Error in the called function.
GN[1]  Z←3÷0
        ^ ^

      )SIS          First entry in the state indicator
GN[1] Z←3÷0         is last expression that did not
       ^ ^          complete.
FN[2] Z←GN×2
        ^
*   FN
    ^
*   3⍳
    ^^
```

*Figure 42. Actions That Add to the State Indicator*

## Clearing the State Indicator

Statements remain in the state indicator until they have been cleared. If a workspace that has items in the state indicator is saved, the state indicator is also saved. There are several ways, discussed below, to clear the state indicator. The one you use depends on what you are trying to accomplish and the situation that caused statements to be put in the state indicator.

***Escape:*** *Escape* (→), a branch arrow with no expression to its right, abandons further attempts to execute the suspended function and the calling sequence that led to its being invoked. Escape clears the state indicator down to and including the next `*`. For example:

```
      )SI
F[1]
*
      →
      )SI
```

You can then correct the error and recall the function.

```
      ∇F[1] 'Z←''CANNOT DIVIDE BY 0''' ⎕EA 'Z←10÷X'∇
      F 0
CANNOT DIVIDE BY 0
```

Because only one calling sequence was in the state indicator, a single → cleared it. This is not the case in the following example:

```
      )SI
D[1]
*
B[2]
*
      →
      )SI
B[2]
*
      →
      )SI
```

To clear the state indicator, one → is needed for each asterisk in the state indicator. As the calling sequence is removed from the state indicator, `⎕EM` and `⎕ET` are set to values appropriate to the statement at the top of the state indicator.

***)RESET:*** The system command `)RESET` clears the state indicator entirely. `)RESET` *n* clears *n* lines from the display of the state indicator. (See `)RESET`, page 449.)

```
      )SI D[1] *
      B[2] *
      )RESET
      )SI
```

As with escape, `⎕EM` and `⎕ET` are set appropriately for the first entry in the state indicator after the reset.

***Resume or Restart Execution*** You may be able to respecify `⎕L` or `⎕R` to a suitable value and resume execution from the point at which it was halted by entering `→⍳0`. Execution can always be resumed by `→⍳0` if the state indicator shows

```
      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^  ^
      ⎕R←1
      →⍳0
10
      )SI
```

Alternatively, you can correct the line in error and redirect execution to begin at that line or some other line by entering `→⎕LC` to restart execution with the current line (see page 300) or →*n*, where *n* is a line number.

```
      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^  ^
      ∇F[1] 'Z←''CANNOT DIVIDE BY 0''' ⎕EA 'Z←10÷X'∇
SI WARNING
      →1
CANNOT DIVIDE BY 0
```

The message `SI WARNING` is displayed when editing affects a line of an operation appearing in the state indicator (if you edit the line or delete or insert lines before it). In these cases, a negative sign precedes the line number in the state indicator, and no statement is shown.

```
      )SIS
F[¯1]
*  F 0
   ^
```

**Note:** If a line has been edited, you cannot use `→⍳0` to resume execution at the point where it halted. You can, however, restart execution by branching to a line number. If no number is shown within brackets, the operation can be neither resumed nor restarted.

***Do Not Resume Execution by Invoking the Operation Again:*** If you correct the error in the operation and then invoke the operation again, the state indicator is *not* cleared. After the operation has executed, the earlier uncorrected version remains in the state indicator.

```
      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^  ^
      ∇F[1] 'Z←''CANNOT DIVIDE BY 0''' ⎕EA 'Z←10÷X'∇
SI WARNING
      F 0
CANNOT DIVIDE BY 0
      )SIS
F[¯1]
*  F 0
   ^
```

Use → or `)RESET` to clear the state indicator before invoking the operation a second time.

### When a Called Operation Is Suspended

Sometimes, a defined operation which has been called by another defined operation is suspended. The state indicator shows the entire calling sequence. The values associated with local names in the operation at the top of the state indicator are the only accessible values for those names. However, you can use the editor to display calling operations. You cannot restart execution after correcting the error unless the corrected defined operation is the first in the state indicator.

## Execution Properties

A defined operation has four execution properties, which can be set *independently* with `⎕FX` (fix with execution properties) in “`⎕FX` Fix (with Execution Properties)” on page 294. The following describes the execution effect of setting each property.

- The defined function or operator may not be displayed or edited through the APL2 editors, through the system function `⎕CR` (character representation), or through `⎕TF` (transfer form); and it may not be traced.
- The defined function or operator is not suspended by an error or an interrupt and it may not be stopped.
- The defined function or operator ignores attentions and stop control settings during its execution. (Interrupts are never ignored.)

  Suspension of defined functions and operators and interrupts are discussed in “Suspension of Execution” on page 354.
- The defined function or operator converts any error other than a resource error into a `DOMAIN ERROR`. (`INTERRUPT`, `SYSTEM ERROR`, `WS FULL`, and `SYSTEM LIMIT` are classified as resource errors.)

The execution properties of a called function or operator during an execution sequence are determined by "or-ing" its properties with those of the calling function or operator. For example, suppose function `F` has the nonsuspendable property (`0 1 0 0`) and function `G` has the error conversion property (`0 0 0 1`). If `F` calls `G`, both the nonsuspendable property and the error conversion property are imposed on `G` (`0 1 0 1`). Because execution properties are inherited by called functions and operators, if a locked function calls an unlocked function, the unlocked function behaves as though it were locked.

Execution properties can be changed only by using `⎕FX` and only if the operation can be displayed. The execution properties of a defined operation can be determined by using `⎕AT` (attributes), page 270.

The default function or operator definition provided by the APL2 editors has *none* of these properties. If an operation is locked during editing (with `⍫`), all the execution properties are set.

## Debug Controls

APL2 includes two facilities for analyzing the behavior of defined functions and operators: trace control and stop control.

## Trace Control

A *trace* is an automatic display of information generated by the execution of each selected line of a defined function or operator. When a statement is traced, the following information is displayed whenever the statement is executed:

- Function or operator name
- Line number in brackets
- Final array value (or branch) produced by that statement

The trace control for a defined operation is designated by prefixing `T∆` to its name. For example, a trace may be set on lines `1`, `3`, and `6` of a defined operation `RS` by executing:

```
T∆RS←1 3 6
```

A trace may be set on all lines by executing:

`T∆RS←⍳`*number of lines in the operation (or more)*

A trace is turned off by setting the trace control to `⍳0`.

```
T∆RS←⍳0
```

Global names beginning with `T∆` may not be used for any purpose other than trace control.

```
      ∇Z←(F XEACH)STACK;X
[1]    Z←''
[2]    ⍝PROCESS FIRST ITEM; EXIT IF ERROR
[3]    L1:'→0' ⎕EA 'X←F↑STACK'
[4]    Z←Z,⊂X                       ⍝APPEND RESULT
[5]    →(0≠⍴STACK←1↓STACK)/L1       ⍝EXIT IF STACK EMPTY
[6]    ∇
```

For example, the function derived by the operator `XEACH` processes each item in its argument until an error occurs.

```
      ⍳XEACH 2 4 6
 1 2  1 2 3 4  1 2 3 4 5 6
```

No error, so each item is processed.

```
      ⍳XEACH 2 4 ¯2 6
 1 2  1 2 3 4
```

Error in the third item, so processing stops after the second item.

Tracing lines `1 3 4 5` shows the behavior of the operator:

```
      T∆XEACH←1 3 4 5
      ÷XEACH 1 0 7
XEACH[1]
XEACH[3] 1
XEACH[4] 1
XEACH[5] →3
XEACH[3] →0
1
      T∆XEACH←⍳0
```

Trace on a line containing multiple expressions separated by diamonds causes trace output for each expression evaluated.

Trace controls can be both set and referenced. A reference to a trace control vector returns only valid line numbers (in increasing order) upon which a trace has been set.

Settings of trace controls are relocated as a result of line insertion or deletion by the APL2 editors.

Trace settings are ignored if the execution property `'nondisplayable'` is set.

## Stop Control

A defined operation can be made to stop before a selected line is executed. When a statement is assigned a stop control, execution stops just before the statement is to be executed, and the following information is displayed:

- Operation name
- Line number in brackets

Execution may be resumed by entering a branch statement.

The stop control for a defined operation is designated by prefixing *S*∆ to its name. For example, a stop may be set on lines 1, 3, and 6 of a defined operation *RS* by executing:

```
S∆RS←1 3 6
```

A stop may be set on all lines by executing:

*S*∆*RS*←⍳*number of lines in the operation (or more)*

A stop is turned off by specifying the stop control to ⍳0.

```
S∆RS←⍳0
```

For example, with the operator *XEACH* shown in the previous section (page 361):

```
      S∆XEACH←4
      ⍋XEACH (9 44 23) (10 11)
XEACH[4]
      X
1 3 2
      →4
XEACH[4]
      X
1 2
      →4
1 3 2  1 2
```

Global names beginning with *S*∆ may not be used for any purpose other than stop control.

Stop controls may be both set and referenced. A reference to a stop control vector returns only valid line numbers (in increasing order) upon which a stop has been set.

Settings of stop controls are relocated as a result of line insertion or deletion by the APL2 editors.

Stop control settings are ignored if the execution property `'ignore weak interrupt'` is set.

# Chapter 8. Shared Variables

Shared variables constitute an interface through which information is passed between two processors—information to be used by each for its own purpose. The two processors can consist of many possible combinations, including two APL2 users, two auxiliary processors, one auxiliary processor and one user, one user and an APL2 interpreter using the shared variable interface, and so on.

The next sections discuss the concepts and usage requirements of shared variables.

## Shared Variable Concepts

A variable becomes shared when one processor has offered to share it and a second processor has accepted the offer (made a counter offer for a variable with the same name). The variable is then *fully-coupled* between the two partners and data communication can take place.

The two processors are called *share partners*.

A given processor can simultaneously share variables with any number of other processors. However, each sharing is *bilateral*; that is, each shared variable has only two partners. For example, a shared data file can be made directly accessible to a single control processor. That processor can share variables bilaterally with each of several other processors, controlling their individual access to the data, as required.

Either partner can set a value for the variable and also use the value. At any one time, a shared variable has only one value—the value most recently set by either partner.

The communication protocol is controlled by the setting of the *access control vector* (ACV), which is defined by either or both partners. The *access state vector* (ASV), which is set by the system, indicates the current state of the shared variable so that you can execute requests appropriate to the state. For a discussion of the access control mechanism provided by the shared variable facility, see “Synchronization of Asynchronous Processors” on page 367.

## APL2 Shared Variable System Functions and System Variable

There are five system functions and one system variable that can be used to establish, query, and maintain proper communication between an APL2 user and a share partner. Chapter 6, “System Functions and Variables” on page 259 describes the syntax and results of the system functions and the setting and use of the system variable. Figure 43 summarizes the results of the monadic and dyadic forms of the functions, and Figure 44 on page 365 summarizes the setting and use of the variable.

*Figure 43. System Functions Used with Shared Variables*

| **System Function** | **Monadic** | **Dyadic** |
|---|---|---|
| ⎕SVO<br>Shared Variable Offer | Obtain the current degree of coupling of the variable(s) entered as the right argument. | Offer the right-argument variable(s) to the processor(s) identified in the left argument. |
| ⎕SVC<br>Shared Variable Control | Query the setting of the access control vector (ACV) for the variable(s) entered as the right argument. | Set your contribution to the ACV (the Boolean vector(s) entered as the left argument) of the variable(s) entered as the right argument. |
| ⎕SVS<br>Shared Variable State | Query the access state vector(s) (ASVs) for the variable(s) entered as the right argument. | Not applicable. |
| ⎕SVR<br>Shared Variable Retraction | Retract the shared variable(s) entered as the right argument. | Not applicable. |
| ⎕SVQ<br>Shared Variable Query | Obtain a list of unmatched *variables* offered by the processor entered as the right argument.<br>If the right argument is an empty vector (⎕SVQ ⍳0), the function returns a list of *processors* that have made an unmatched offer to you. | Not applicable. |

*Figure 44. System Variable Used with Shared Variables*

| **System Variable** | **Set** | **Use** |
|---|---|---|
| ⎕SVE<br>Shared Variable Event | Specifies the amount of time in seconds to be used in a wait for a shared variable event and starts the timer. | Suspends execution until either the specified number of seconds has elapsed or a shared variable event occurs. When an event occurs, returns the time remaining in the timer. |

## Characteristics of Shared Variables

Syntactically, a shared variable is indistinguishable from any other variable. The only reliable way of knowing whether a variable is shared is to know its *degree of coupling*. The degree of coupling is a scalar integer maintained for each variable. It indicates the number of partners with whom it is shared (0, 1, or 2). It is the explicit result of both monadic and dyadic ⎕SVO and ⎕SVR. For detailed information, see “Degree of Coupling” on page 366.

***Number of Shared Variables:*** Some auxiliary processors distributed with APL2 require a single variable to accomplish a user request; some accept one variable under some conditions and two variables under other conditions; others require a pair of variables matched either by name or by initial value.

***Shared Variable Names:*** The maximum length of a shared variable name cannot exceed 255 characters. Auxiliary processors can restrict the length of the name to less than 255, or can require special naming conventions.

For a variable to be shared, two partners must offer the same name for the variable. To maintain the independence between two autonomous processors, APL2 permits the use of an alias or surrogate name to be shared when one partner requires a certain naming convention that is inconvenient for the other partner to comply with. For the general syntax of offering surrogate names, see the discussion of ⎕SVO in Chapter 6, “System Functions and Variables” on page 259.

***Shared Variable Values:*** The value associated with a variable at the time it is offered to a partner is the *initial value*. Some auxiliary processors require an initial value; some ignore an initial value. With others, an initial value is optional. After sharing has been established, the values you subsequently set or use in a shared variable depend on the function and requirements of the processor with which you are communicating. Some processors restrict the type of data that can be shared. For requirements for shared variable values, see the appropriate associated processor in the workstation user's guides or *APL2/370 Programming: System Services Reference*.

## Communication Procedure

The following general procedure is used to communicate using shared variables.

1. Offer to share the variable(s).
2. Ensure the degree of coupling is 2.
3. Set access control for each variable offered.
4. Access the variable(s) by following the protocol established for them and the requirements for their values.
5. Retract the variable(s).

   Retracting a variable withdraws your share offer with your partner. After retraction, the variable can be offered to another processor or reoffered to the same processor. A variable can be explicitly retracted using ⎕SVR, or it can be implicitly retracted when the variable no longer exists in your workspace or the workspace no longer exists. For a description of ⎕SVR and a list of conditions when a variable is implicitly retracted, see “⎕SVR Shared Variable Retraction” on page 332.

## Degree of Coupling

The degree of coupling is the explicit result of offering a shared variable (dyadic ⎕SVO) or inquiring about a variable's share status (monadic or dyadic ⎕SVO). Explicitly retracting a variable (⎕SVR) returns the degree of coupling the variable had immediately before it was retracted.

The degree of coupling is a scalar integer that indicates the number of partners that share or have shared the variable. The possible values are 0, 1, or 2. The meaning of each value is described below.

***Degree of Coupling = 0:*** Either you have made no offer, or the offer failed. Reasons for a failed offer include:

- The name you have specified as a shared variable is in use as the name of a function, an operator, or a label.

- The name contains invalid characters (including names that begin with the quad, ⎕).
- The variable is already shared with, or has been offered to, another processor (a variable can be shared by only two partners).

***Degree of Coupling = 1:*** Your offer is pending. It may or may not be matched in the near future. Reasons for a pending offer include:

- Unacceptable variable name specified in the offer when the specified partner requires a certain naming convention.
- The processor needs a pair of variables to communicate, but only one has been offered.
- A nonexistent processor ID was specified in the offer.
- The processor has already accepted its maximum number of shared variables.
- Your partner has not yet matched your offer.

  This is typical with the asynchronous behavior of the APL2 auxiliary processors. Use the system variable ⎕SVE to explicitly wait a reasonable amount of time for your offer to be matched. See “Signaling of Shared Variable Events” on page 373.

***Degree of Coupling = 2:*** Sharing is complete; the variable is fully coupled. Each partner has offered the variable to the other.

## Synchronization of Asynchronous Processors

In most practical applications it is important to know that a new value has been assigned by your partner between your successive uses of a shared variable, or that use has been made of a value before you set a new one. The shared variable facility embodies an *access control* mechanism to help ensure proper communication.

The access control operates by inhibiting the setting or use of a shared variable by either or both owners, depending on the values of two Boolean vectors maintained for each shared variable. The vectors are the access control vector (ACV) and the access state vector (ASV).

The access control vector, queried by monadic ⎕SVC and set by dyadic ⎕SVC, contains the protocol that regulates the sequences for access of the variable by the two partners. It indicates whether repeated attempts to set or to use a variable by one partner require either a use or a set by the other.

The access state vector, which is set by the system and can only be queried by you (using ⎕SVS), indicates two things:

- Which partner(s) have used (know) the current value of the variable
- Which partner, if any, has set a value in the variable that has not yet been used by (is unknown to) the other partner.

## Symmetry of the Access Control Mechanism

Although each item of the access control and access state vectors has its own meaning, the relative positions of each item in the vectors relate to each other. Your view of the vectors is:

- The first and third items refer to you
- The second and fourth items refer to your partner
- The first and second items refer to sets
- The third and fourth items refer to uses

Figure 45 shows the meaning of each item in the vectors.

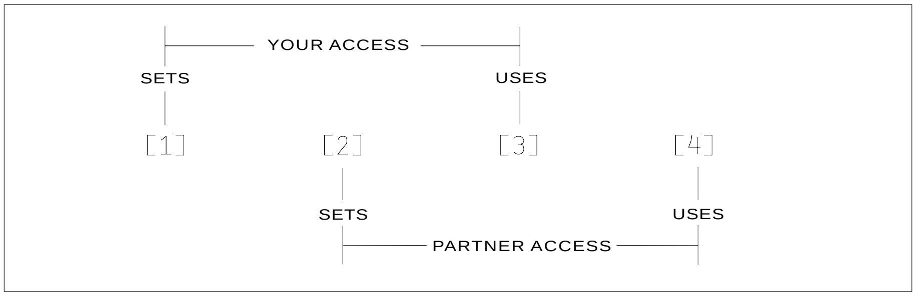


*Figure 45. Items in the Access Control and Access State Vectors*

The view your partner has of the access vectors is the mirror image of yours. To clarify the symmetry of the vectors and to help you remember which items are which, reshape the vector to a 2 by 2 Boolean matrix.

```
      ⎕SVC 'SHARED'
0 1 1 0
      ⎕←ACV←2 2⍴⎕SVC 'SHARED'
0 1
1 0
```

Reverse the matrix to see your partner's point of view:

```
      ⌽ACV
1 0
0 1
```

In matrix form, column one refers to the viewer, and column two refers to the viewer's partner. Row one refers to sets, and row two refers to uses. Figure 46 illustrates the control mechanism in matrix form.

|  | YOUR ACCESS | PARTNER ACCESS |
|---|---|---|
| SETS | [1] | [2] |
| USES | [3] | [4] |

*Figure 46. Access Control or Access State Vectors as a Matrix*

## Access Control Vector

The settings in the access control vector indicate any constraints on the partners for access to a shared variable. A `1` indicates a constraint on the partner and type of access represented for each position of the vector. A `0` places no constraints on access. The constraints placed for each item are:

*ACV*[1] You cannot set the variable two times in a row without an intervening access by your partner. (An access is either a set or a use.)

*ACV*[2] Your partner cannot set the variable two times in a row without an intervening access by you.

*ACV*[3] You cannot use the variable two times in a row without an intervening set by your partner.

*ACV*[4] Your partner cannot use the variable two times in a row without an intervening set by you.

Figure 47 illustrates, in matrix form, the constraints imposed for each position of the access control vector.

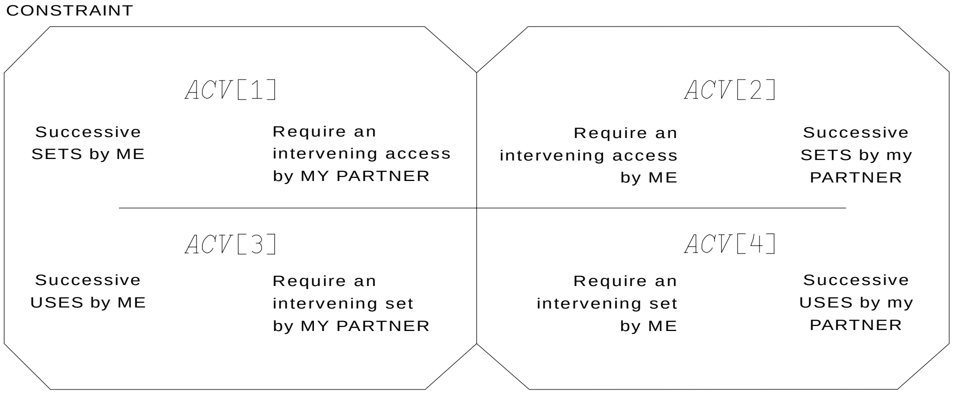


*Figure 47. Access Control Matrix*

### Setting the Access Control Vector

You set access control when you want to synchronize your access of a variable with your partner's access of that same variable. For example, to make sure your partner always has a chance to set new data in the variable each time you use it, you can place a constraint on yourself that inhibits your use until your partner sets the variable. By setting the appropriate constraints on yourself and your partner, you can impose an orderly dialog between you both.

Below are examples of access control vectors and their meanings.

**Note:** Any combination of the four items in the vector is valid.

0 0 0 0 No constraints. Regardless of which partner set the last value and regardless of which partners know or do not know the current value, either partner can both set and use the value. Sharing can be completely asynchronous.

1 1 1 1 Maximum constraint. Neither partner can set the variable two times in a row without an intervening access by the other partner. In addition, neither partner can use the variable two times in a row without an intervening set by the other partner.

| | |
|---|---|
| 1 0 1 0 | Constraint on both your set and your use of the variable, but no constraint on your partner's access. |
| 0 0 1 1 | Constraint on both your use and your partner's use of the variable. Allows either partner to set the variable. This setting ensures that neither partner will see the value more than once. |
| 0 0 1 0 | Constraint on your use of the variable without an intervening set by your partner. |

Access control should be set immediately after you offer a shared variable. If you set it before the offer, it is ignored. After it is established, the ACV remains in effect until you or your partner changes it.

The access control vector (ACV) can be set by either partner. Typically, auxiliary processors set appropriate access control vectors for the services provided. You should generally set the access control vector to prevent accidental loss of data.

Your setting of an access control vector results in the OR (∨) of your setting and the setting established by your partner. You can contribute only to the setting (that is, impose additional constraints), and you can decrease only the constraints you yourself have imposed. You are not allowed to decrease the control established by your partner.

Use monadic `⎕SVC` to query the current ACV setting for a variable. Use dyadic `⎕SVC` to impose additional constraints on the access control set by your partner or to reduce your contribution to the constraints. For example:

```
      ⎕SVC 'CMS100'
0 0 0 1

      1 0 0 0 ⎕SVC 'CMS100'
1 0 0 1

      0 0 0 0 ⎕SVC 'CMS100'
0 0 0 1
```

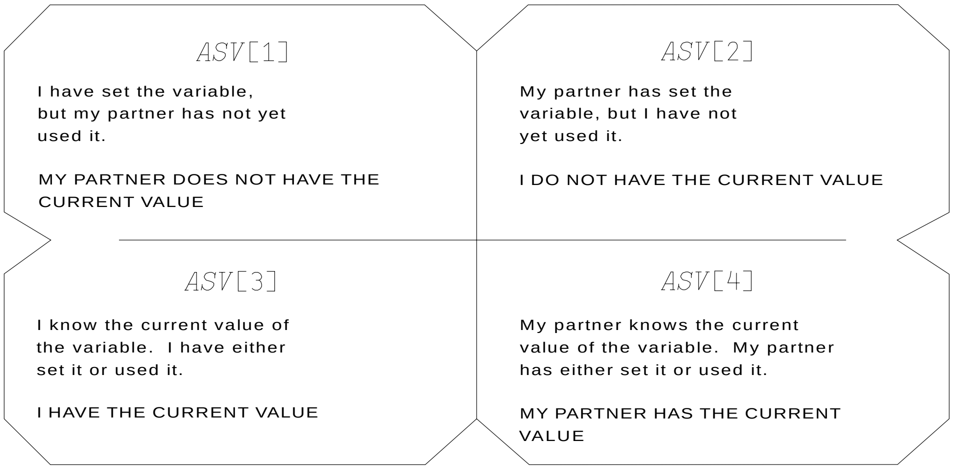
## Access State Vector

The settings in the access state vector indicate which partner(s) knows the current value, and which partner, if any, last set a value unknown to the other partner. The items in the vector have the same relationship to the partners and the access of the variable as the access control vector does.

Use `⎕SVS` to see the access state vector. A `1` in an item of the access state vector (`ASV`) has the following meaning:

| | |
|---|---|
| `ASV[1]` | You have set a value which your partner has not yet used. |
| `ASV[2]` | Your partner has set a value which you have not yet used. |
| `ASV[3]` | You know the current value of the variable. |
| `ASV[4]` | Your partner knows the current value of the variable. |

To illustrate its symmetry, Figure 48 on page 371 shows the meanings of the items in the access state vector in terms of a matrix.



ASV[1]

I have set the variable, but my partner has not yet used it.

MY PARTNER DOES NOT HAVE THE CURRENT VALUE

ASV[2]

My partner has set the variable, but I have not yet used it.

I DO NOT HAVE THE CURRENT VALUE

ASV[3]

I know the current value of the variable. I have either set it or used it.

I HAVE THE CURRENT VALUE

ASV[4]

My partner knows the current value of the variable. My partner has either set it or used it.

MY PARTNER HAS THE CURRENT VALUE

*Figure 48. Access State Matrix*

### Access State Values

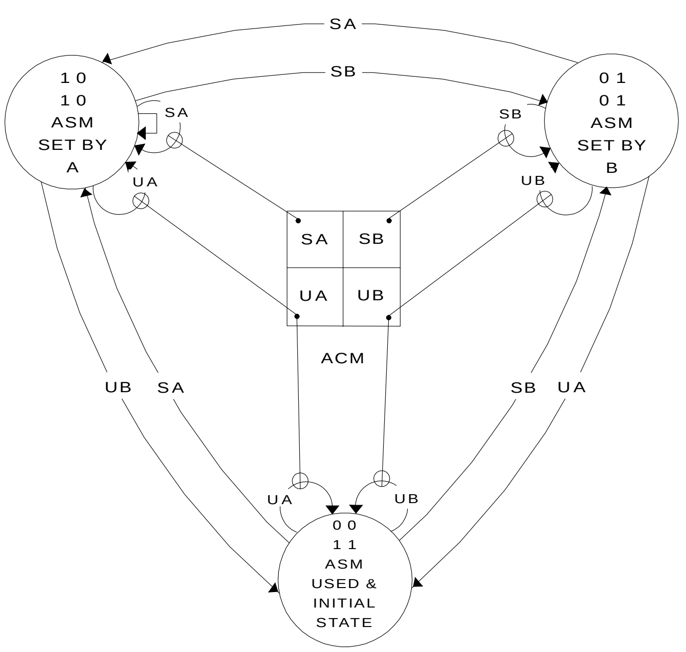
The access state vector can contain only one of four possible values:

| | |
|---|---|
| 0 0 0 0 | Not a shared variable. |
| 0 0 1 1 | The current value is known by both partners. This is also the setting when a variable is first offered. |
| 1 0 1 0 | You have set a value your partner has not yet used. You know what the value is, but your partner does not. |
| 0 1 0 1 | Your partner has set a value you have not yet used. Your partner knows what the value is, but you do not. |

Like the access control vector, the access state vector can be viewed from the perspective of each partner. If your use of the ASV is 1 0 1 0, the ASV is seen by your partner as 0 1 0 1.

## Effect of Access Control and Access State on Communications

Figure 49 on page 372 illustrates the permissible and non-permissible actions that can be taken by two share partners under the possible combinations of settings of the access control and access state vectors. Lines around the perimeter are permissible actions in all cases. Lines around the inside are constrained or inhibited by ACV and ASV values.



**Legend:**

SA SB UA UB: Denote *set* or *use* by A or B.
ACM: Access Control Matrix
ASM: Access State Matrix

A one in an element of ACM inhibits the associated access. Allowable accesses are given by the zeros in `ACM∧ASM` . Access control vectors as seen by A and B, respectively, are `,ACM` and `,⌽ACM`.

The access state matrix represents the last access: ones occur in the last row if it is not a set, and in a column if it is, the first column if set by A and the last if set by B.

*Figure 49. Access Control of a Shared Variable*

## Shared Variable Interlock

Execution is suspended if you attempt to access a shared variable twice in a row when its ACV is set to prevent your successive set or use of the variable. When this occurs, you are *interlocked*. Waiting for an access of the variable by your partner, execution is suspended for an indefinite amount of time. Enter an interrupt to release the suspension and raise an `'INTERRUPT'` signal.

For details on entering an interrupt under a particular system, see the appropriate workstation user's guide or *APL2/370 Programming*: *System Services Reference*. When an interrupt is entered, the INTERRUPT message appears in the session output.

## Over Specification

No access control can be set that can prevent you or your partner from ignoring (not using) the value the other partner has set in a variable. When one partner sets a new value *over* the other's set without first using the value, it is called *overset* or *overspecification*.

Typically, auxiliary processors supplied with APL2 specify a return code indicating success or failure from the most recent operation requested of the processor. Because there is no requirement for you to use a variable your partner has set, you are not *required* to obtain the return code your partner sets. However, you could lose important diagnostic information if a problem should arise while communicating with an auxiliary processor. It is recommended that you always check the return code from every auxiliary processor operation.

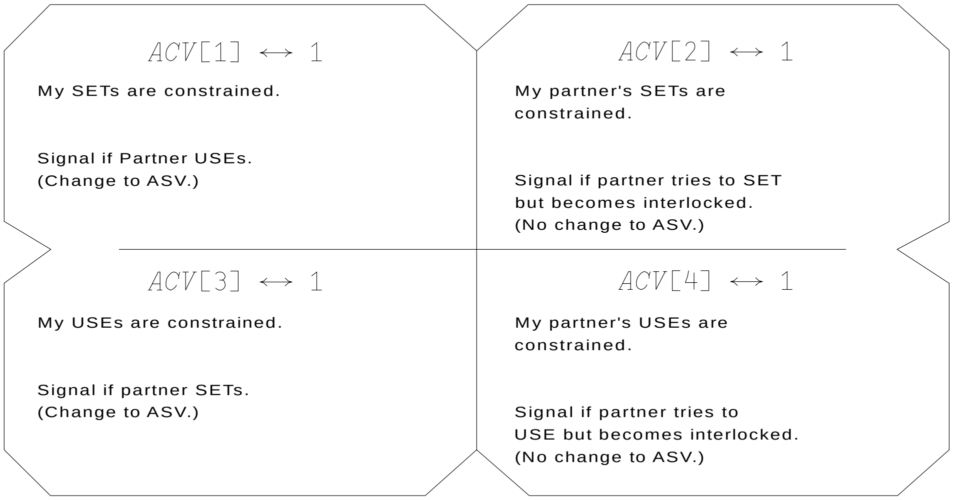
## Signaling of Shared Variable Events

The system variable ⎕*SVE*, shared variable event, gives you the ability to suspend execution until a *shared variable event* occurs.

A shared variable event occurs when one of the following happens:

- An incoming offer to share a variable does not match a pending offer (made by you)
- Your partner matches a pending (or outstanding) offer from you
- Your partner retracts a variable you share
- Your partner sets the access control vector (any dyadic ⎕*SVC*) on a fully-shared variable
- Your partner attempts to access a variable under the situations shown in Figure 50 on page 373.

  **Note:** The access state vector may or may not change.
- The ⎕*SVE* timer expires before some other shared variable event occurs.



*ACV*[1] ↔ 1

My SETs are constrained.

Signal if Partner USEs.
(Change to ASV.)

*ACV*[2] ↔ 1

My partner's SETs are constrained.

Signal if partner tries to SET but becomes interlocked.
(No change to ASV.)

*ACV*[3] ↔ 1

My USEs are constrained.

Signal if partner SETs.
(Change to ASV.)

*ACV*[4] ↔ 1

My partner's USEs are constrained.

Signal if partner tries to USE but becomes interlocked.
(No change to ASV.)

*Figure 50. Shared Variable Accesses that Signal a Shared Variable Event*

The specification of ⎕*SVE* sets and starts a timer for *n* seconds. A use of ⎕*SVE* delays execution until either the specified number of seconds elapses or one of the shared variable events listed above occurs.

Use of ⎕*SVE* gives you a means of determining when to take another action. For example, you can use ⎕*SVE* to:

- Wait for your share offer to be matched
- Wait for offers to be made to you
- Wait for reference or specification of shared variables.

Regardless of the use, the structure of the code to establish a wait is similar to and may be a variation of the following:

```
[1]  ⎕SVE←N                           ⍝ SET THE TIMER
[2]  TRY: →(condition)/0             ⍝ EXIT IF FUNCTION SUCCESSFUL
[3]  →(0≠⎕SVE)/TRY                    ⍝ WAIT FOR A SHARED VARIABLE EVENT
[4]  'FAILURE'                        ⍝ MESSAGE IF TIME EXHAUSTED
```

The condition in line 2 depends on the shared variable function being used. For example, to wait for an appropriate access state:

```
    ∇
[0]   Z←N WAITSET VAR
[1]   ⎕SVE←N                          ⍝ START TIMER
[2]   TRY: →(Z←0 1 0 1≡⎕SVS VAR)/0    ⍝ EXIT IF I CAN USE VARIABLE
[3]   →(0≠⎕SVE)/TRY                   ⍝ WAIT FOR A SHARED VARIABLE EVENT
[4]   'EVENT DIDN''T HAPPEN'          ⍝ TIME EXHAUSTED
    ∇
```

# Chapter 9. The APL2 Editors

APL2 supports a number of editors for creating and modifying defined functions and operators. Most of the editor facilities described in this chapter are available on all APL2 platforms, though there are some differences, especially in the techniques used for invoking the editors. The following types of editors are supported:

- A traditional line editor.

  This editor can be used with all types of session input devices. It can also be used effectively for interactive processing when combined with the line reuse facilities of the APL session manager.

- An APL full-screen (or windowed) editor.

  This editor, often called ***Editor 2***, deals with pages of text and has no command line. It depends on function keys or other special control keys for actions such as scrolling. It supports concurrent editing of multiple objects.

  A simpler full-screen editor is available on some platforms, but does not support concurrent editing or have as many editing features, and can be used only to edit programs, not arrays.

- An interface to system editors.

  This interface passes an APL object from the active workspace to some editor independent of APL, and that is available on the system. Typically, any editor can be used, though with some it is difficult or impossible to handle APL characters. In most cases only a single object can be edited at a time.

- An interface to user-written APL editors.

  Installations or users can provide their own editors written in APL, or written specifically to meet APL2 interfaces.

There are two groups of platforms, with somewhat different characteristics:

***Workstation Platforms:*** The line editor and named system editors can be invoked using ∇ followed by an object name or program header. The editor to be invoked by ∇ is the one that was last specified using the `)EDITOR` system command. The supported choices are:

`)EDITOR 1`
for the line editor (this is the default)
`)EDITOR` *name*
for a system editor

Two other editors are provided in the EDIT workspace:

- The APL full-screen editor can be accessed as follows:

```
)PCOPY 1 EDIT EDITOR_2
EDITOR_2 'object_name'
```

- A smaller and faster, but limited function, APL full-screen editor for defined functions and operators can be accessed in a similar way:

```
)PCOPY 1 EDIT EDIT
EDIT 'program_name'
```

Users can write their own editors in APL2, and can access them as described in the example above.

***APL2/370 Platforms:*** All editors can be invoked using ∇ followed by an object name or program header. The editor to be invoked by ∇ is the one that was last specified using the `)EDITOR` system command. The supported choices are:

`)EDITOR 1`
: for the line editor (this is the default)

`)EDITOR 2`
: for the APL full-screen editor

`)EDITOR` *name*
: for a system editor, where *name* can be any command or command procedure that could be invoked directly from a CMS or TSO command line.

`)EDITOR 2` *name*
: for a user-provided editor. The name provided must be one that is defined as a processor 11 entry point, and can be either a compiled program or an APL program in a namespace. For further information about processor 11, see the *APL2 Programming: System Services Reference*.

User-written APL editors can also be accessed using `)PCOPY` as described for the workstation platforms.

Unless explicitly changed, the selected editor remains available throughout the session. System commands, including `)CLEAR` and `)LOAD`, do not alter the editor setting.

In this chapter, the word *object* refers to an array, a function, or an operator.

## Editor Features

Editors 1 and 2 provide similar functions, although the provision for these functions and their scope differ occasionally. Whenever possible, similar commands are used; you need not learn two different sets of commands.

Figure 51 lists the major features of each editor. Some features are available only when the session manager is used. Editor 2 provides extended capabilities.

*Figure 51. Features of the APL2 Editors*

| **Feature** | **Editor 1 Line Editor** | **Editor 2 Full-Screen Editor** | **Named System Editor**[1] | **Named APL Editor**[2] |
|---|---|---|---|---|
| Define a function or operator | yes | yes | yes | yes |
| Receive line number prompts | yes | no | no | yes |
| Lock a function or operator | yes | yes | yes | yes |
| Abandon editing of an object | yes | yes | yes | yes |
| Edit a function or operator: | | | | |
| Add lines | yes | yes | yes | yes |
| Replace lines | yes | yes | yes | yes |
| Insert lines | yes | yes | yes | yes |
| Insert or delete characters in a line | yes | yes | yes | yes |
| Copy lines from current object | yes[3] | yes | yes | yes |
| Copy lines from another object | yes[3] | yes | no | yes |
| Move a line | yes[3] | yes | yes | yes |
| Globally change text or names in an object | yes | yes | yes | yes |
| Locate occurrences of text or names in an object | no | yes | yes | yes |
| Display object or selected portions | yes | yes | yes | yes |
| Scroll through the display of an object | yes[3] | yes | yes | yes |
| Delete object body or selected portions | yes | yes | yes | yes |
| Edit multiple objects | yes | yes | no | yes |
| Execute expression while in definition mode | yes | yes | no | yes |
| Edit simple character vector or matrix | no | yes | yes | yes |
| Enter system commands while in definition mode | yes | no | no | yes |
| Record display and editing of object in the session log | yes[3] | no | no | yes |
| Renumber lines | no | yes | no | yes |
| Establish object in the workspace without exiting editing | no | yes | no | yes |
| Rename a function | yes | yes | yes | yes |
| Edit arrays | | | | |
| Simple character array | no | yes[4] | yes | yes |
| Numeric array | no | no | yes | yes |
| Nested array | no | no | yes | yes |
| Mixed array | no | no | yes | yes |

**Notes:**

1. The APL2 interface to the named system editor allows all of the features marked “yes.” Particular host editors can restrict some of these features.
2. The named APL editor interface provides full access to APL2 facilities. A given editor can restrict access to APL2 facilities.
3. These features are available only when the session manager is used.
4. This cannot edit simple scalars.

## Characters Permitted within Statements

For statements containing characters that are elements of ⎕*TC*, the following restrictions apply:

- If the statement is displayed by Editor 1 or an error display, or if the ⎕*CR* or ⎕*EM* of the statement is displayed, the ⎕*TC* characters are interpreted as control characters. In the case of backspace, one or more characters preceding the backspace in the line is overstruck or overlaid.
- If the statement is displayed with Editor 2, the ⎕*TC* characters are displayed as blots or blanks.
- Statements containing ⎕*TC* characters cannot be entered with Editor 1 or 2. They can be introduced into objects being edited with Editor 2 with the commands `[∧]` and `[⍎]`.
- Statements containing ⎕*TC* characters can be modified by Editor 1 but those modifications cause loss of the ⎕*TC* characters and possibly characters following the ⎕*TC* characters, since Editor 1 accepts the modified line as displayed.
- Statements containing ⎕*TC* characters can be modified by Editor 2, but those modifications can cause the ⎕*TC* characters to be converted to blanks or deleted.
- When named editors are used, ⎕*TC* characters are handled as defined by the named editor. Many named editors provide for hexadecimal display, modification, and entry, and thus support the editing of these characters.

For statements containing other characters that are contained in ⎕*AV*, but cannot be displayed on the display device in use, the following restrictions apply:

- If the statement is displayed by Editor 1 or 2, or an error display, or if the ⎕*CR* or ⎕*EM* of the statement is displayed, the characters are displayed as blots or blanks.
- Statements containing nondisplayable characters cannot be entered with Editor 1 or 2. They can be introduced into objects being edited with Editor 2 with the commands `[∧]` and `[⍎]`.
- Statements containing nondisplayable characters can be modified with Editor 1 or 2, but modifications can cause those characters to be converted to blanks or lost, or can cause an *ENTRY ERROR* or a *DEFN ERROR* to be generated.
- When named editors are used, all characters are handled as defined by the named editor. Many named editors provide for hexadecimal display, modification, and entry.

Characters that are not contained in ⎕*AV* are referred to as extended characters. Some APL environments can provide a character set identification that defines a range of extended characters that can be correctly handled. (See the description of the DBCS invocation option in *APL2/370 Programming:* *System Services Reference*.) For a given character set identification, N, an extended character, C, is defined to be within the range if

```
N=256⊥2↑(4⍴256)⊤⎕AF C
```

For objects containing extended characters that are outside the range, the following considerations apply if a character set identification is not provided by the APL environment:

- Extended characters are displayed as omegas (⍵) by Editor 1. Modification of statements so displayed causes the extended characters to be converted to omegas.
- Editor 2 and named system editors produce a *DEFN ERROR* when editing is requested.
- If the object to be edited contains extended characters outside the range defined by the character set identification, Editor 2 and named system editors produce a *DEFN ERROR* when editing is requested. Editor 2 also produces a *DEFN ERROR* if such characters are introduced into the object being edited with `[∧]` or `[⍎]` commands. Editor 1 causes such characters to be converted to omega (⍵) prior to display.
- Named APL editors are responsible for prompting their users for input and establishing any changes made to objects in the workspace. Some of these editors can support extended characters on suitably-equipped devices.

For objects containing extended characters within the range, the following considerations apply if a character set identification is provided by the APL environment:

- If the object to be edited contains shift-in or shift-out characters (⎕*AF* 14 or ⎕*AF* 15), Editor 2 and named editors produce a *DEFN ERROR* when editing is requested. Editor 2 also produces a *DEFN ERROR* if shift-in or shift-out characters are introduced into the object being edited with `[∧]` or `[⍎]` commands. Editor 1 treats shift-in and shift-out characters as nondisplayable characters as described above.
- Objects containing extended characters within the range defined by the character set identification are correctly displayed and can be entered and modified with Editors 1 and 2 and certain named editors on suitably-equipped devices.

For additional information concerning display and entry of extended characters, see the appropriate workstation user's guide or *APL2/370 Programming:* *System Services Reference*.

## Named System Editor

APL objects can be passed to a specified named system editor in response to ∇. Unlocked functions and operators and arrays of rank 0, 1, and 2 can be edited. An attempt to edit an array that is not a scalar, vector, or matrix results in a *DEFN ERROR* report.

The named system editor is set with `)EDITOR` *name* and persists for the entire session unless changed. It can refer to an EXEC or a MODULE in CMS, a CLIST in TSO, a shell script in AIX* and Solaris, or an executable file in OS/2 (for example, a .CMD file).

The editor converts the APL object in your active workspace to a character matrix form, writes a temporary file, and then invokes the named system editor, passing it the name of the temporary file.

With array editing, the rank of the array is preserved unless additional items or rows are added. A scalar can be coerced to a vector if it is changed to a nonsingle or a matrix, if other than one row is created. A vector can be coerced to a matrix if additional rows are appended. Array rank never decreases, so a matrix edited to one row results in a 1 by n matrix and not a vector.

When given a name to edit that does not currently exist, it is assumed to be the name of a program. To initialize a variable to be edited as a simple character array, `NAME←0 0⍴''` . (See “Editing Simple Character Arrays”.) To initialize a variable to be edited in evaluated form `NAME←0 0⍴0` . (See “Editing Evaluated Arrays (APL2/370 Only)” on page 381.)

To lock a program, you must begin or end with ⍫. You cannot lock the program during editing.

### Exiting the Editor

To exit the editor, simply exit as indicated by your named system editor. If there is an error in a temporary file you have altered, and you want to exit without saving that file, you must either erase the temporary file or change the file to consist of a single line containing a blank character. Then, you exit.

### Editing a Program

The text of the program being edited is displayed in the same form used by Editor 1 or Editor 2, except that line numbers are omitted. On exit, the program is re-fixed if there were any changes. Any currently suspended version of the same program is replaced by the new version.

### Editing Simple Character Arrays

Simple character arrays of rank 0, 1, and 2 are displayed in their default display form, one record of the file per row of the variable.

The file built for editing a rank 0 or 1 array has variable record length. The file built for editing a rank 2 array has fixed record length equal to which ever is larger, the width of the array or the variable name.

Upon exit from the editor, trailing blank columns are deleted from rank 0 and 1 arrays. Trailing blank columns are not deleted from rank 2 arrays. Some system editors support commands which can be used to change the file's record format and length. The editors may then strip trailing blanks if the data is saved.

To exit the editor without changing rank 0 or 1 arrays, make the file consist of a single line with a blank character, then exit. To exit the editor without changing rank 2 arrays, simply exit.

## Editing Evaluated Arrays (APL2/370 Only)

All arrays that are not simple character arrays are edited in an evaluated form. On exit, each record is evaluated in the user's workspace. Since the records are evaluated, they may contain any APL expression that returns an array result. The resulting array is reconstructed by combining the evaluated records using disclose (⊃).

If an error occurs during evaluation or reconstruction, a *DEFN ERROR* message is displayed briefly, possibly followed by a specific APL error report and the first offending record. The editor is then re-entered after a short delay.

Under CMS, the name of the temporary edit file used is *n* AP2EDEVL, where *n* is the first counting number of a currently unused file name.

For example: Suppose a record was changed from 1 2 (3 4 5) to 1 2+(3 4 5). A new screen would appear showing:

```
DEFN ERROR
LENGTH ERROR
      1 2+(3 4 5)
     ^   ^
```

Then, the editor is re-entered after a short delay. To exit the editor without changing the array, make the file consist of a single line with a blank character, then exit.

**Note:** The display of an array can include APL2 characters that define the array's structure. Most commonly used are ⍴ ( ), ⊂, and '.

For example:

```
A←1ρ¨1 2 3
```

would be displayed as

```
 A
 (,1) (,2) (,3)
```

## Named APL Editor (APL/370 Only)

User requests to edit APL objects can be passed to a named APL editor. In response to a ∇, APL2 uses ⎕NA and processor 11 to create an association to and call the named APL editor to handle the edit request.

The named APL editor is identified with `)EDITOR 2 name` and persists for the entire session unless changed. The named APL editor can reside in an APL2 namespace or be a non-APL program.

The named APL editor is executed as if it had been called directly from the user's current namescope. However, it is not associated in the current namescope so its association does not cause name conflicts.

## Guidelines for Writing a Processor 11 Editor

When the user enters an expression with a leading ∇, APL2 attempts to establish an association with the name specified in the `)EDITOR 2 name` command. APL2 uses `3 11` as the left argument to ⎕NA. If the association fails, APL2 then uses `name 11` as the left argument to ⎕NA. APL2 then calls the function.

The named APL editor function is passed a character vector containing the user's ∇ expression. It is the function's responsibility to parse the vector, interpret the user's request, invoke an editor or provide editing capabilities, and reestablish the object's definition. APL2 does not ensure that the ∇ expression's syntax is valid. It is the responsibility of the processor 11 function to interpret the expression.

**Note:** There is one exception to this. If the expression indicates a valid request for display of all or part of a function's or operator's definition using `)EDITOR 1` rules, the request is answered by APL2; the named APL editor is not called.

If the function resides in a namespace, it can use the *EXP* function to reach back into the user's current namescope to reference or specify object definition(s). If the function is a non-APL program, it may use processor 11 external services (XE, XF, and so on) to access the user's namescope.

## Editor 1 (The Line Editor)

Some editing features of Editor 1 are illustrated in Figure 52, which assumes that a function *PIGLAT* has been defined to convert a word, phrase, or sentence into Pig Latin. Editing of *PIGLAT* is shown.

The current version of *PIGLAT* prompts for an entry:

```
      PIGLAT
ENTER A WORD, PHRASE, OR SENTENCE:
      HAPPY BIRTHDAY
YOUR ENTRY IN PIG LATIN IS:
 APPYHAY IRTHDAYBAY
```

The following sequence illustrates some editing of *PIGLAT*: a line is inserted to enter an expression containing quotation marks; the header is modified; and two lines are deleted to allow a word, phrase, or sentence to serve as an argument to a function instead of a response to a prompt.

```
      ∇PIGLAT[⎕]                Display the function and keep the definition open
    ∇                           for editing (page 388)
[0]   PIGLAT
[1]   'ENTER A WORD, PHRASE, OR SENTENCE:'
[2]   X←⍞
[3]  ⍝ REMOVE DUPLICATE BLANKS
[4]   X←(~'  '⍷X)/X
[5]  ⍝ REPLACE BLANKS WITH QUOTE-BLANK-QUOTE
[6]   ((' '=X)/X)←⊂''' '''
[7]  ⍝ REMOVE NESTING AND EXECUTE
[8]   X←1↓⍎'0 ''',(∊X),''''
[9]   'YOUR ENTRY IN PIG LATIN IS:'
[10]  (1⌽¨X),¨⊂'AY'
    ∇  10.17.51 11/06/83 (GMT-5)
[11]  [2.1] ⍝ DOUBLE ANY QUOTES  Insert lines  (page 390)
[2.2] X←(1+''''=X)/X
[2.3] [∆1 2]                    Delete lines (page 391)
[2.01] [0]   Z←PIGLAT X         Replace a line (page 390)
[0.1][10⎕10]                    Insert characters in line10
[10]  Z←(1⌽¨X),¨⊂'AY'           Insert characters in a line
[11]  ∇                         Close the function definition (page 388)
```

*Figure 52. Editing with Editor 1 (The Line Editor)*

## Line Numbers

Each line of text in the definition of an object begins with a bracketed line number, which is displayed to the left of the line. After a definition is opened for editing, either the bracketed line number is supplied as a prompt by the system or you must type the bracketed line number itself.

The objects header line number is always `[0]`. When the object is first displayed, the statements are assigned consecutive positive integers, beginning with `[1]`. Most editing uses these line numbers as references.

To insert a new line in a definition, number the new line with a fractional number (between brackets) to indicate its position relative to the existing lines. For example, a line inserted between lines `[3]` and `[4]` may be given the fractional number `[3.1]`. A line inserted between `[3]` and `[3.1]` may be given the number `[3.01]`. When a definition is closed, the system renumbers all lines to sequential positive integers.

### Line Number Prompts

Editor 1 displays a bracketed line number as a prompt in the form:

`[n]`

In response, you can do one of the following:

- Enter text for the line
- Change the line number and enter text
- Enter a command

The line number may be an integer or a fraction, depending on which commands for display, deletion, and insertion have been issued previously. For example:

Prompting for an insert following the display of line 1:

```
      ∇ENTER[⎕1]
[1]   I←TABι⊂,NAME
[1.1]
```

Prompting for an additional line following the display of line 2:

```
         ∇ENTER[⎕2]
   [2]   TAB←TAB,(I>ρTAB)/⊂,NAME
   [3]
```

The system attempts to avoid prompting with an existing line number. If unavoidable, the next line is displayed with text. Up to four places after the decimal can be used to specify an added line.

## Editor 1 Commands

Except for the opening and closing ∇ and ⍫ (lock) commands, all Editor 1 commands are entered in brackets.

Editing commands may be entered on any line of a displayed object or may be entered in response to a line number prompt. For example:

| | | |
|---|---|---|
| `[3]` | `[→]` | Abandon editing of the object |
| `[5]` | `[⎕1-3]` | Display lines `1` through `3` |

No text should appear to the left of the left bracket. If several bracketed line numbers or edit commands appear on the same line, only the rightmost is executed. For example:

```
[3]    [→][1.5] 'INSERT AFTER 1'
[1.6]                                Only the insert is made
```

Editor 1 commands are described in the following sections.

## Opening a Definition

To open a definition of an object, use the ∇ (del) command:

∇ *option*

To open a locked definition of an object, use the ⍫ (del tilde) command:

⍫ *option*

The locked function cannot be displayed or edited after the definition is closed. For a description of locked definitions, see “The Effect of Locking” on page 388.

***Options for Opening a Definition:*** The following options for opening a definition allow you to define and name an object, rename the object, and add lines or insert lines to an existing object.

To Define a New Object
∇ *header*

The header defines the syntax of the defined function or operator. At the minimum, the header must contain the name of the object. “Defined Functions and Operators” on page 31 describes the various forms of headers.

After you type a del, the header, and press the Enter key, the line prompt `[1]` is displayed and the cursor or carrier is positioned in column 7, indicated by an underbar (_) in the examples. For example:

```
       ∇Z←MEAN X
[1]    _
```

If the header name is that of a variable or a locked function or operator, a *DEFN ERROR* is generated. If the header consists of only a name and that name already exists in the workspace as a defined operation, a prompt for the next available numbered line is displayed. If the header consists of more than a name (arguments, operands, explicit result, and/or local names) and the name already exists in the workspace as a defined operation, a *DEFN ERROR* is generated.

The header can be edited and the name of the object can be changed. However, if the name is changed, a new object is opened. The definition associated with the former name remains unless the object is explicitly erased with )*ERASE* or ⎕*EX*.

To Add to the End of an Existing Object
∇ *name*

The line prompt `[`*n*`+1]` is displayed, where *n* is the last line of the object. For example:

```
      ∇XEACH
[6]   _
```

If the name has not been established, the definition is opened for a niladic defined function without a result and a prompt for line `1` is displayed. The header may be changed by editing line `0`.

To Edit an Existing Object
`∇` *name* `[`*command*`]`

Editing an existing defined function or operator begins by executing the given command. For example:

```
      ∇XEACH[⎕4]
[4]   Z←Z,⊂X
[4.1] _
```

If only a line number is included within the brackets, a prompt is displayed with that line number.

```
      ∇XEACH[4]
[4]   _
```

If a closing del (`∇`) follows the command, the edit command is executed and the definition is closed. For example:

```
      ∇XEACH[⎕4]∇
[4]   Z←Z,⊂X
```

Suspended operations may be edited. When the definition is closed, the message *SI WARNING* is displayed. Do not reexecute the operation until you have cleared the suspension from the state indicator. See “Clearing the State Indicator” on page 357.

## Opening More Than One Object for Editing

More than one object can be opened for editing at the same time. After a definition is closed, the next previous definition is available for editing. For example, function *B* is opened during the definition of *A*. When the definition of *B* is closed, editing of *A* resumes.

```
      ∇A
[1]   'A1'
[2]   'A2'
[3]   'A3'
∇B                    Second function definition is opened.
[1]   'B1'
[2]   'B2'∇
[4]   [⎕0]            When definition of B is
[0]   A               closed, editing of A resumes.
[0.1] ∇
```

If there are intervening suspensions of execution, previous definitions cannot be resumed until the suspensions are cleared from the state indicator. For example, an intervening suspension of execution prevents the definition of function *F* from being resumed after the definition of *G* is closed.

```
      ∇F
[1]   'F1'
[2]   'F2'
)SI                   State indicator shows that F
F[3] ∇                is being edited.
*
[3]   'F3'            Definition of F continues.
⍳1.2
DOMAIN ERROR          Error in an immediate execution expression.
⍳1.2
^
      ∇G              New object is defined.
[1]   'G1'
[2]   'G2'∇
      )SI             Editing of F will not continue
*                     because editing is not at the top
F[4] ∇.               of the execution stack.
*
      →
[4]   'F4'∇           After the top item in the stack is cleared
                       F can be edited.
```

***Caution:*** An operation already opened for editing should not be opened for editing again. Unexpected results can occur.

## Closing a Definition

To close a definition, enter either the del (∇) or the del tilde (⍫).

The closing ∇ (or ⍫ to lock the object) may appear in the following contexts:

- At the end of any line of text that does not include a comment (⍝). For example:

  ```
  [2]    Z←(+/X)÷⍴X∇
  ```

- Or on a line by itself. For example:

  ```
  [3.1]  ∇
  ```

- At the end of any editing command. For example:

  ```
  ∇F[⎕]∇
  ```

Closing the definition establishes the object in the workspace and ends the editing of it. Use `)SAVE` to store the updated workspace. (See “Abandoning Editing of an Object” on page 392 to learn how to quit definition mode without establishing the object in the workspace.)

If more than one object is opened for editing, the closing del for one object is followed by a line number prompt for the next open definition in LIFO order (unless there are intervening suspensions of execution on the execution stack, as discussed in “Opening More Than One Object for Editing” on page 387.) The closing del on the only remaining open object ends the editing session and returns you to immediate execution mode.

***The Effect of Locking:*** An object whose definition is locked cannot be displayed or edited; it can only be erased and re-created, if it is necessary to change the definition. If you want to lock a function when it is used in a production application, you should keep an unlocked version of the function as an aid to maintenance of the application. (See also `⎕FX`, page 294, for setting execution properties of a defined operation.)

## Changing the Name of an Object

You can change the name of an object by editing the name in line `0`. If the object is subsequently saved, it is saved under the new name. The original definition (under the original name) will not be affected. An attempt to change the name of an object to one that already exists in the workspace is rejected with a `DEFN ERROR`.

## Displaying an Object

To display an object, enter:

[⎕ *option*]

All display commands include a ⎕ (quad) to indicate display. Display commands may be entered when a definition is opened, or at any time during the editing session as a response to a line number prompt.

***Display Options:*** The following options allow you to display either the entire definition or only a range of lines. Line numbers specified as endpoints of display ranges need not exist. Lines falling within the range are displayed.

### To Display a Definition

`[⎕]`

The object is displayed beginning with line `0`. For example:

```
      ∇HEXDEC[⎕]
    ∇
[0]   Z←HEXDEC X
[1]  ⍝ CONVERTS X IN HEXADECIMAL TO DECIMAL
[2]   Z←16⊥¯1+'0123456789ABCDEF'⍳X
    ∇  1993-05-21 14.45.24 (GMT-5)
[3]   _
```

### To Display Specific Lines

`[⎕`*n1 n2  n3 . . .*`]`

The lines specified by the vector of line numbers are displayed. These line numbers may be listed in the command in any order and may contain repetitions and redundant blanks. APL2 expressions to define the line numbers are not permitted. For example:

```
      ∇DUMP[⎕1 2 3 7]
[1]  ⍝  DUMP DEFINITIONS IN THE WORKSPACE
[2]  ⍝  USES FUNCTIONS:  DISP, SHOW
[3]  ⍝  DUMP, DISP, SHOW, NL¯, AND X¯ ARE NOT DUMPED
[7]  ⍝  DISP MAY BE MODIFIED FOR A FILE OR A PRINTER
[7.1] _
```

### To Display a Range of Lines

`[⎕`*n1-n2*`]`

The lines specified from *n1* through *n2* are displayed. For example:

```
      ∇DUMP[⎕4-6]
    ∇
[4]   NL¯←⎕NL 3 4
[5]   X¯←3 4⍴'DISPDUMPSHOW'
[6]   NL¯←(∧/NL¯∨.≠⍉(1↓⍴NL¯)↑[⎕IO+1]X¯)⌿NL¯
[6.1] _
```

To Display from the Beginning Line to Line *n*
`[⎕-n]`

The object is displayed from line `0` through line *n*. For example:

```
       ∇DUMP[⎕-2]
     ∇
[0]    DUMP;NL¯ X¯
[1]  ⍝  DUMP DEFINITIONS IN THE WORKSPACE
[2]  ⍝  USES FUNCTIONS:  DISP, SHOW
[2.1] _
```

To Display from Line *n* through the End Line
`[⎕n-]`

The object is displayed from line *n* through the end. For example:

```
       ∇DUMP[⎕7-]∇
     ∇
[7]  ⍝  DISP MAY BE MODIFIED FOR A FILE OR A PRINTER
[8]    DISP¨⊂[⎕IO+1]NL¯
```

## Replacing or Inserting Lines

To replace or insert lines in a definition, enter:

`[`*n*`]` *text*

where *n* may be a whole or fractional number:

- If *n* is an existing line number, the existing line is replaced.
- If *n* does not exist, the new line is inserted.

For example:

```
[4]    [2.1] X←∊X                    Insert a line
[2.2]  [1] ⍝ X MUST BE NUMERIC       Replace a line
```

## Copying or Moving Lines

Copying lines in Editor 1 is a one-step procedure:

- To copy a line, change its line number.

When you press Enter, the copied line appears as the line number specified, but the original text and line number are unaffected.

Moving lines in Editor 1 is a two-step procedure:

1. Copy a line by changing its line number.
2. Delete the original line.

When you press Enter, the copied line appears as the line number specified, and the original text and line number is deleted.

## Deleting Lines

To delete lines from a definition, use the delta (∆):

`[∆` *option*`]`

The delete command deletes all or part of an object, except the header. The delete command may be entered when a definition is opened or at any time during the editing session.

Some form of the delete command is required to delete lines from the definition of an object that has been established in the active workspace. Merely erasing a line from the screen does not delete it from the object.

No text should follow the delete command. Thus, if the delete command is inserted to precede a number in a bracketed line number, the ERASE EOF key should be used to delete the text from the line.

***Delete Options:*** The following delete options allow you to delete specific lines or a range of lines. These options are similar to the display options. Line numbers specified as endpoints of delete ranges need not be existing line numbers. Line numbers falling within the range are deleted. Renumbering occurs after the definition is closed.

To Delete Specific Lines
`[∆`*n1 n2 n3 . . .*`]`

The lines specified by the vector of line numbers are deleted. These line numbers may be in any order and may contain repetitions and redundant blanks. Zero is ignored. APL2 expressions to define the line numbers are not permitted. For example:

```
[6]    [∆1  4]
[4.1]  _
```

To Delete a Range of Lines
`[∆`*n1-n2*`]`

Lines *n1* through *n2* are deleted. For example:

```
[9]    [∆4-7]
[7.1]  _
```

To Delete from Line *1* through Line *n*
`[∆-`*n*`]`

Lines from line *1* through line *n* are deleted. For example:

```
[4]    [∆-3]
[3.1] _
```

### To Delete from Line *n* through the End Line

`[∆n-]`

Lines from line *n* through the end are deleted. For example:

```
[3]    [∆4-]
[8]
```

### To Delete All the Lines

`[∆1-]`

Except for the header, the entire object is deleted. Use `)ERASE` or `⎕EX` to remove an object from the active workspace.

**Note:** The command `[∆]` results in a `DEFN ERROR`.

***Caution:*** If you delete more lines than you intended, stop editing the object with the `[→]` command. The original definition is retained and can then be reopened for editing. This recovery technique works only for objects previously established in the workspace. Lines deleted in nonestablished objects cannot be recovered.

## Abandoning Editing of an Object

To abandon editing of the function or operator, enter:

```
    [→]
```

This command ends the editing session and retains the most recently established definition of the object. If the object is new (not established), none of its definition is retained. This command may be entered at any time during the editing session.

If more than one object is opened for editing, the `[→]` quits from the most recently opened object. Editing of the next object resumes if there are no intervening suspensions of execution, as discussed in “Opening More Than One Object for Editing” on page 387. When no further open definitions exist, the editing session ends.

### The Display-Edit Command

To display a line for editing when the session manager is not available, enter:

```
[n⎕p]
```

where:

| | |
|---|---|
| *n* | Specifies a particular line of the defined function or operator to be displayed. |
| *p* | Is 0 or a positive integer specifying a position within the line. |

**Note:** You cannot use the [*n*⎕*p*] command to edit lines longer than ⎕*PW* or the screen width.

The edit action:

- Displays line *n* in the input area.
- If *p*=0, places the cursor just after the end of the line.

  If *p*≠0, places the cursor at the position specified.
- Accepts input to change the line.

The display-edit command may be entered as the command when a definition is opened or at any time during the edit session.

## Immediate Execution with Editor 1

Any input line not beginning with a left bracket ([) is executed immediately and is not part of the definition. The definition of the defined function or operator is suspended and the input line is executed in the active workspace. After successful execution, editing is resumed. If an error occurs, the statement in error is normally cleared from the state indicator with → or )*RESET* *n*, so that editing can continue. Clearing the error from the state indicator is especially important if more than one object is opened for editing, as described in “Opening More Than One Object for Editing” on page 387.

### Entering System Commands

System commands can be entered by typing over or erasing the line number prompt or by typing on any blank line. The system command or any response to it is not treated as input to the editor.

The system commands )*SI*, )*SINL*, and )*SIS* use the del (∇) to identify the names of defined functions and operators still in edit mode.

If you enter the )*EDITOR* 2 system command while one or more definitions are open in Editor 1, Editor 2 (the full-screen editor) is started only for the definitions opened after you issue the )*EDITOR* 2 command. The definitions previously opened in Editor 1 remain available within Editor 1 and can be edited or closed when all segments of Editor 2 are closed.

If, during editing, you save the workspace with )*SAVE*, edit mode is resumed when the workspace is loaded.

The following system commands cause editing to end without the object's being established in the active workspace:

```
)CLEAR
)LOAD
)OFF
)CONTINUE
)RESET
```

## System Services and Editor 1

Use of Editor 1 is affected by the APL2 session manager and by the type of display.

### Editor 1 with the APL2 Session Manager

When Editor 1 is invoked, the facilities of the session manager are still available for use with the editor.

Lines in the session log can have bracketed numbers and can be used as input to the definition. Lines selected for reuse that do not have bracketed line numbers are treated as lines for immediate execution, not as lines of text for the definition.

Displayed lines of the definition are lines of output in the session manager and can be edited directly by typing over the displayed text. When Enter is pressed, the lines are redisplayed in the session manager input/output area, following the rules of the session manager. The original lines reappear as they were displayed, and the changed lines appear with bracketed line numbers as input to the definition.

For more information on the APL2 session manager and its use, see the appropriate workstation user's guide or *APL2/370 Programming*: *System Services Reference*.

### Editor 1 without the APL2 Session Manager

If you start an APL2 session without the session manager, Editor 1 operates under the standard input/output protocol of the host system.

---

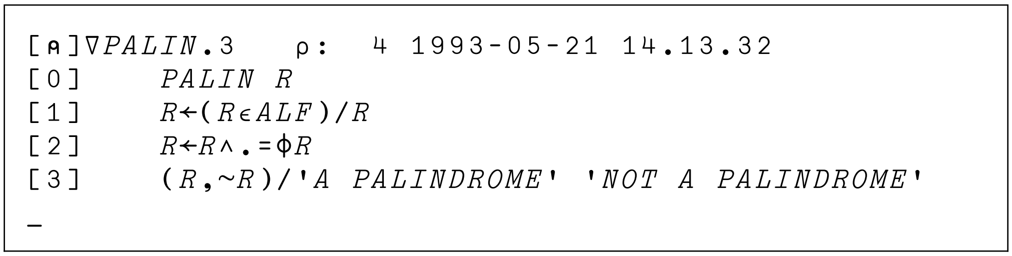
## Editor 2 (Full-Screen Editor)

If used on a display device, Editor 2 operates in full-screen mode and uses function keys to simplify entry of commands.

Changes made through Editor 2 are not recorded in the session log when the editor is invoked from the session manager. The command that starts the editor is recorded, but work within the editor does not become part of the log.

To log the results of changing an object with Editor 2, close the definition to fix it in the active workspace and redisplay the object in the session manager, using the expression `∇NAME[⎕]∇` (where `NAME` is the name of the object).

Figure 53 shows the Editor 2 display of a function opened for editing with `∇PALIN`.



```
[⍝]∇PALIN.3   ⍴:  4 1993-05-21 14.13.32
[0]    PALIN R
[1]    R←(R∊ALF)/R
[2]    R←R∧.=⌽R
[3]    (R,~R)/'A PALINDROME' 'NOT A PALINDROME'
_
```

*Figure 53. Display of Object with Editor 2*

Editor 2 displays one *page* at a time of the definition. The display includes:

| | |
|---|---|
| *information line* | Identified by `[⍝]`. |
| *line number field* | The first six columns of each line. The remaining columns of each line are the *text field*. |

## Information Line

When a definition is opened, an information line precedes the header line. The information line varies for defined operations and variables.

***For Defined Functions and Operators:*** The information line has the following format:

`[⍝]∇`*name.x* `⍴:` *n yyyy-mm-dd hh.mm.ss*

where:

| | |
|---|---|
| *name* | Is the name of the object being edited. |
| *x* | Is a number that indicates the type of object:<br>`3` = Defined function<br>`4` = Defined operator |
| `⍴:` *n* | Indicates the number of rows in the object when last established in the workspace. It is calculated as `1↑⍴⎕CR` *name*. |
| *yyyy-mm-dd* | Is the date the object was last fixed in the workspace. |
| *hh.mm.ss* | Is the time the object was last fixed in the active workspace. |

The above fields display zeros if the object is new.

The information line is updated each time you establish the current definition in the workspace with the `[∇]` command (fix the object in the workspace and remain in edit mode).

***For Simple Character Vectors and Matrixes:*** The information line has the following format:

`[⍝]∇`*name.2* `⍴:` n

where:

| | |
|---|---|
| *name* | Is the name of the variable being edited. |
| *.2* | Indicates a character vector or matrix. |

| | |
|---|---|
| `ρ:` *n* | Is the shape of the variable. This field is updated each time the variable is saved in the workspace with a `[∇]` command. |

## Line Numbers

When they are displayed, all object lines are given bracketed line numbers. The object's header is line `[0]`, and subsequent lines are assigned the positive integers beginning with `[1]`. Editing is done with reference to these line numbers or to the lines themselves.

During editing, inserted lines are displayed with fractional numbers to indicate their positions relative to existing lines. For example, a line inserted between lines `[3]` and `[4]` is displayed with the fractional number `[3.1]`. A line inserted between `[3]` and `[3.1]` is displayed with the number `[3.01]`. All lines are renumbered to positive integers when a definition is closed or in response to the renumber command (`[⍳]`).

### Line Number Prompts

Editor 2 does not display line number prompts. However, after you press Enter, the editor numbers previously unnumbered lines. You can also enter bracketed line numbers as part of a line.

## Editor 2 Commands

Unlike other full-screen editors, Editor 2 does not have a command line. Therefore, editing with this full-screen editor is accomplished by one of the following:

- Issuing commands, as with Editor 1
- Manipulating line numbers within brackets
- Modifying or manipulating displayed lines of the object
- Scrolling through function keys or commands within brackets and not with command line commands

Except for the opening and closing `∇` and `⍫` (lock) commands, editor commands are entered within brackets. They may be entered on any line of a displayed object or on a line by themselves, with no text to the left. For example:

| | |
|---|---|
| `[→]` | Abandon editing of the object |
| `[∆5]   I←1+I` | Delete line 5 |

Two sets of bracketed numbers cannot be adjacent on the same line. For example:

```
[5][2.1]
```

The `[2.1]` is taken to be the text for line `5`; a `SYNTAX ERROR` is generated when the operation is subsequently executed.

Some edit commands are represented by function keys. The function key assignments are shown in Figure 54 on page 397. Each key's use is discussed with the explanation of the associated edit command.

*Figure 54. Function Key Assignments for Editor 2*

| | | |
|---|---|---|
| 1 [?]<br>Display Function Key Settings | 2 [⍳]<br>Renumber | 3 ∇<br>Close Definition |
| 4 | 5 | 6 [∇]<br>Fix Object in Workspace.<br>Resume Editing |
| 7 [↑]<br>Scroll Backward | 8 [↓]<br>Scroll Forward | 9 [⊤]<br>Cursor-Dependent Scroll Forward |
| 10 | 11 | 12 |

If you use a display terminal, the number of lines in the object may exceed the capacity of the definition area for display. When an existing definition is opened, as much of the requested definition as will fit in the definition area is shown. To see the remainder of the display, use the scrolling commands [↑] (PF7), [↓] (PF8), and [⊤] (PF9), as described in “Scrolling through a Definition” on page 401.

If more than one line or command is entered on the screen, they are processed from top to bottom when ENTER is pressed.

## Opening a Definition

To open a definition of an object, use the ∇ (del) command:

∇ *option*

To open a locked definition of an object use the ⍫ (del tilde) command:

⍫ *option*

The locked function cannot be displayed or edited after the definition is closed. For a description of locked definitions, see “Effect of Locking” on page 399.

More than one object can be opened for editing at a time. Editing multiple objects is discussed in “Editing Multiple Objects” on page 411.

***Options for Opening a Definition:*** The following options for opening a definition are available.

To Define a New Object
∇ *header*

The header defines the syntax of the new defined function or operator. At minimum, it must contain the object name. The header can be edited. However, if the name is changed, a new object is implied. The definition associated with the former name remains unless the object is explicitly erased with `)ERASE` or `⎕EX`.

After you press the Enter key, both the information line and line `[0]` display. The remaining lines are blank.

If the header consists of only a name, and that name already exists in the workspace as a defined operation or character vector or matrix, the object is displayed. If the header specifies a monadic or dyadic operation and the name already exists in the workspace as a defined operation, a `DEFN ERROR` is generated.

To Edit an Existing Object
∇ *name* [*command*]

Editing of the existing defined function or operator begins by the user executing the given command. If only a line number is included within the brackets, a `DEFN ERROR` results.

If a closing ∇ follows the command, the edit command is executed and the definition is closed. Full-screen mode is not entered.

If no command is given, the object is displayed. If the name has not been established, the definition is opened as a niladic defined function without an explicit result.

To Edit a Simple Character Vector or Matrix
∇ *name*

Only named simple character vectors and matrixes can be edited. The character vector or matrix is displayed without single quotation marks and may be edited without them. They are implied by the information line that displays .2 after the object name (meaning a variable), and quotation marks should not be explicitly entered unless they are to be displayed as part of the variable. If a vector or matrix is so wide that one row does not fit on the whole screen, then the object cannot be edited.

If the name represents an array that is not a simple character vector or matrix, a `DEFN ERROR` is generated. If the name does not exist in the workspace, it is taken to represent the header of a niladic defined function without an explicit result.

## Closing a Definition

To close a definition, press PF3 or enter either ∇ (del) or ⍫ (del tilde).

The closing ∇ (or ⍫ to create a locked object) may appear on a line by itself or after the text on any line or after a command. For example:

```
[2]  ⍝  L MAY BE NESTED TO DEPTH 2∇
```

```
[∆7 8 12]∇
```

The ∇ establishes the definition in the active workspace and ends editing of the object.

Closing the definition establishes the object in the workspace. Use `)SAVE` to store the updated workspace. For a description of quitting the definition mode without establishing the object in the workspace see “Abandoning Editing of an Object”.

For a description of the effect of the closing del when multiple objects are being edited, see “Editing Multiple Objects” on page 411.

***Effect of Locking:*** An object whose definition is locked cannot be displayed or edited; it can only be erased and re-created if it is necessary to change its definition. Although on rare occasions you may want to lock a function when it is used in a production application, you should keep an unlocked version of the function as an essential aid to maintenance of the application. (See also ⎕FX, page 294, for setting execution properties of a defined operation.)

## Fixing the Object in the Workspace and Staying in Edit Mode

To fix the function, operator, or character vector or matrix in the workspace *and* stay in edit mode, press PF6 or enter:

```
    [∇]
```

This command establishes the current definition of the object in the active workspace but leaves the definition open.

After this command is executed, the information line for defined functions and operators is updated.

## Abandoning Editing of an Object

To abandon editing of the function, operator or character vector, or matrix, enter:

```
    [→]
```

The `[→]` command ends a definition without establishing the object. It must be entered in brackets. It can be entered on a line by itself or it can be typed over any displayed line number in the definition.

## Changing the Name of an Object

You can change the name of an object by editing the name in line 0. If the object is subsequently saved, it is saved under the new name. The original definition (under the original name) is not affected. An attempt to change the name of an object to one that already exists in the workspace is rejected with a *DEFN ERROR*.

## Displaying an Object

To display an object, enter:

[⎕ *option*]

Display commands include a ⎕ (quad) to indicate display. They may be entered as the command when a definition is opened with a ∇ or ⍫ or at any time during the editing session.

If the display command is issued as part of opening a definition and if it is not followed by a closing del, full-screen edit mode is entered. If the display command is followed by a closing del, full-screen edit mode is not entered.

When the display command is issued as part of opening a definition, a full page of the requested display is shown without the information line. When you press Enter or a function key for the first time, the information line and a page of the object are displayed, beginning with the first line requested. You can then scroll through the definition. If, however, you close the definition (with ∇ or PF3), you do not see the information line and the object is not displayed further.

If a display command is entered during an editing session, the requested lines are displayed at the point at which the command was entered and remain displayed until you press Enter or a function key. The object is then redisplayed beginning with the first line shown on the screen. Therefore, in order to move a specific line in the object, issue the display command on the line following the information line.

***Display Options:*** The following display options are available. You can display the entire object, specific lines, or ranges of lines. Line numbers specified as endpoints of display ranges need not be existing line numbers. Lines falling within the range are displayed.

To Display the Definition
[⎕]

The first page of the object is displayed beginning with line 0.

To Display Specific Lines
[⎕ *n1 n2  n3 . . .*]

The lines specified by the vector of line numbers, up to a full page, are displayed.

The line numbers in the command may be in any order and may contain repetitions and redundant blanks. APL2 expressions to define the line numbers are not permitted.

To Display a Range of Lines
[⎕*n1-n2*]

The lines specified from *n1* through *n2* are displayed.

To Display from the Beginning Line to Line *n*
`[⎕-n]`

The object is displayed from line `0` through line *n*.

To Display from Line *n* through the End
`[⎕n-]`

The object is displayed from line n through the end.

## Scrolling through a Definition

To scroll one screen backward or to the top, press PF7 or enter:

```
[↑]
```

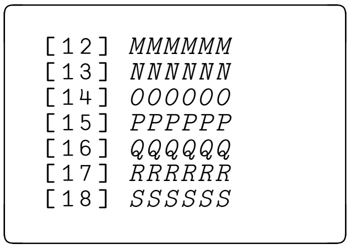
To scroll one screen forward or to the bottom, press PF8 or enter:

```
[↓]
```

To scroll one screen forward from the cursor position, press PF9 or enter:

```
[⊤]
```

***Scroll Backward:*** If this command is entered on the first screen of the definition, no action is taken. For all other screens, the first line on the screen becomes the last line displayed after the command is executed.



```
[12] MMMMMM
[13] NNNNNN
[14] OOOOOO
[15] PPPPPP
[16] QQQQQQ
[17] RRRRRR
[18] SSSSSS
```

**Before** PF7 is pressed
or `[↑]` is entered.

```
[ 6] GGGGGG
[ 7] HHHHHH
[ 8] IIIIII
[ 9] JJJJJJ
[10] KKKKKK
[11] LLLLLL
[12] MMMMMM
```

**After** PF7 is pressed
or `[↑]` is entered.

***Scroll Forward:*** The last line on the screen becomes the first line displayed after the command is executed.

```
[12] MMMMMM
[13] NNNNNN
[14] OOOOOO
[15] PPPPPP
[16] QQQQQQ
[17] RRRRRR
[18] SSSSSS
```

**Before** PF8 is pressed
or `[↓]` is entered.

```
[18] SSSSSS
[19] TTTTTT
[20] UUUUUU
[21] VVVVVV
[22] WWWWWW
[23] XXXXXX
[24] YYYYYY
```

**After** PF8 is pressed
or `[↓]` is entered.

***Cursor-Dependent Scroll Forward:*** The line on which the command is issued becomes the *first* line of the definition displayed. Succeeding lines of the object are displayed in numeric order below the line.

Pressing PF9 makes the line that the cursor is on be the first line of the next displayed screen.

```
[12] MMMMMM
[13] NNNNNN
[14] OOOOOO
[15] PPPPPP
[16] QQQQQQ
[17] RRRRRR
[18] SSSSSS
```

**Before** PF9 is pressed
or `[⊤]` is entered.

```
[16] QQQQQQ
[17] RRRRRR
[18] SSSSSS
[19] TTTTTT
[20] UUUUUU
[21] VVVVVV
[22] WWWWWW
```

**After** PF9 is pressed
or `[⊤]` is entered.

***Cursor-Dependent Scroll Backward:*** To display the line on which the command is issued as the *last* line displayed, press PF9 followed by PF7.

## Adding Lines

Lines can be added to an object or inserted into it by typing the addition or insertion or by copying another line.

***Adding Lines by Typing:*** Input lines can be typed anywhere within the definition area and may be numbered with a bracketed line number `[`*n*`]` or entered without a line number.

When editor processing is requested (by an Enter or a function key), the editor sequentially renumbers the unnumbered lines in the definition area from top to bottom.

A line can be added to the end of a definition by:

- Typing it on a blank line after the last line of the object.
- Typing the new line over line `[0]`. Start your text over the left bracket (`[`). Use the ERASE EOF key to delete any text not needed. When you press Enter, line `[0]` is restored and the new line is added to the end of the object. This line is displayed as the first line of the page; the remaining definition area is blank.

- Typing over any displayed line with the new line number and text. Use the ERASE EOF key to delete any text not needed. When you press Enter, the line typed over is restored and the new line is added to the end of the definition. You can see the added line by using a display command or scrolling to the end of the definition.

***Inserting Text Lines by Typing:*** There are two options for inserting text lines by typing.

*Option 1*

To insert one or more lines between two lines of an existing object, type the new text beginning at the left margin on the line immediately following the line at which the text is to be inserted. The lines typed over retain their original definition, and the inserted lines are numbered by the system.

**Before Insertion**

```
[2]   N←2
[3]  L1:A←∊ARRAY
[4]   Z←N ROUND(+/A)÷⍴A
```

**Insertion Typed**

```
[2] N←2
→(CHARACTER)/NONUM
[4]   Z←N ROUND(+/A)÷⍴A
```

**Result After Enter**

```
[2  ]   N←2
[2.1]  →(CHARACTER)/NONUM
[3  ]  L1:A←∊ARRAY
[4  ]   Z←N ROUND(+/A)÷⍴A
```

If the inserted line is shorter than the line on which it is typed, press ERASE EOF to clear the remainder of text from the line; otherwise, the remaining text appears as part of the inserted line.

*Option 2*

Number any line of the definition with the appropriate fractional line number and type the text.

If the line used is an existing text line, the line typed is restored to its original text, and the line created is inserted as the line number specified.

### Entering Lines Wider Than One Screen Row—Continue Command

To enter a line wider than one screen row, precede the continuation line with:

```
[ ]
```

The continue command is used to create a single logical line from text lines that are wider than the width of the screen. The command can be typed anywhere on the line but may have no text to its left.

If typed as part of a text entry, the continue command indicates that the text line is continued from a previous screen row. When the editor processes a line with continue commands, it creates a single text line that covers more than one screen row. The `[ ]` is never displayed by the editor.

### Creating a Single Line from Two Lines—Continue Command

To append a text line to the line above it, erase, with the terminal's delete key, the text's line number, but not the brackets. The brackets remain:

```
[ ]
```

When you press Enter, the line is appended to the previous line. Indicate that the line is to continue, if necessary, by inserting `[ ]` at the beginning of each successive row.

**Note:** The text of the original line is unchanged.

### Replacing Text Lines

Lines of the definition are replaced by a new line with the same number as the line you want to replace. To replace text lines, do one of the following:

- Display the line to be changed and type over the line text (using the ERASE EOF key as needed); do not change the bracketed line number.
- Number and type the text of the replacement on a blank line.
- Change the bracketed number of a displayed line and type over the displayed text.

  The line typed over retains its original definition; the line created replaces the line with the number specified.

## Inserting and Deleting Characters in a Line

You insert characters into a line by using one of the following:

- Insert mode of the terminal keyboard
- Change command described on page 407

You delete characters using the terminal delete key.

## Deleting Lines

To delete lines from a definition, use the ∆ (delta):

`[∆ option]`

The delete command deletes all or part of an object, except the header. The delete command may be entered when a definition is opened or at any time during the editing session.

Some form of the delete command is required to delete lines from the definition of an object that has been established in the active workspace. Merely erasing a line from the screen does not delete it from the object.

***Delete Options:*** The following delete options allow you to delete specific lines or a range of lines. Line numbers specified as endpoints of delete ranges need not be existing line numbers. Line numbers falling within the range are deleted.

To Delete Specific Lines

[∆*n1 n2 n3 . . .*]

The lines specified by the vector of line numbers are deleted. These line numbers may be in any order and may contain repetitions and redundant blanks. Zero is ignored. APL2 expressions to define the line numbers are not permitted.

To Delete From Line *n1* to Line *n2*

[∆*n1-n2*]

To Delete from Line *1* to Line *n*

[∆-*n*]

To Delete from Line *n* through the End Line

[∆*n*-]

To Delete All the Lines Except the Header

`[∆1-]`

Use `)ERASE` or `⎕EX` to remove an object from the active workspace. Note that the command `[∆]` results in a `DEFN ERROR`.

***Caution:*** To avoid deleting lines that should be retained, always enter one or more line numbers after the ∆. If you delete more lines than intended, stop editing the object with the `[→]` command. The original definition is retained and can then be reopened for editing. This recovery technique works only for objects previously established in the workspace. Deleted lines in objects not previously established cannot be recovered.

## Renumbering Lines

To renumber lines, press PF2 or enter iota (`⍳`) within brackets:

```
[⍳]
```

All lines of the object are renumbered to consecutive integers beginning with `[0]` for the header.

Lines are also renumbered when the definition is closed.

## Locating Strings of Characters—Locate Command

To locate strings of characters and display the lines that contain the string, use the locate command:

[*/string/* `N` *lines*]

where:

*/string/*

Specifies the string of characters to be located.

The delimiter, represented here by `/`, may be any nonalphameric character not occurring in the string **except** the following:

```
] → ↓ ↑ ⍳ ? ⍝ ⎕ ∆ ∇ ∧ ∨ ⊤
```

If the terminating delimiter is not specified neither `N` nor *lines* may be specified.

An entry in the form */string* specifies a search of all object lines.

`N` Specifies that the characters in string represent an APL2 name. Valid matches include strings within quotation marks but exclude any strings of characters that are not also APL2 names.

If `N` is omitted, any occurrence of the string is located.

`N` may be specified in combination with *lines*.

*lines*

Specifies the lines to be searched for the string.

If *lines* is omitted, the search begins with line `0`. If *lines* is not omitted, only one of the following may be specified:

- *n1 n2 n3 . . .*, specifies a vector of line numbers
- *n1*`-`n2, specifies a range of lines
- *n*-, specifies all lines in the object beginning with line *n*
- *-n*, specifies all lines in the object through line *n*.

***Caution:*** No text should follow the locate command. Use the ERASE EOF key, as necessary, to delete text after the command.

***Sample Command:*** In the following example, `α` is the delimiter, and the string to be located is `/∧\`. All lines through line 13 of the object are to be searched for the string.

```
[α/∧\α -13]
```

***Characteristics of the Locate Command Display:*** The located lines replace the current display in the definition, beginning with the line at which the command is entered (including the information line) and continuing for a page. Lines not displayed are unaffected by the command. Located lines may be edited.

When Enter or a function key is subsequently pressed, the object is redisplayed beginning with the first line shown on the screen. If the command was issued on the first line of the screen, the display will begin with the first found line.

If more lines are located than can fit on a screen, scrolling commands cannot be used to view additional screens of output. Instead, a second locate command must be issued specifying line numbers beyond that shown as the last line on the screen.

## Replacing One String of Characters with Another—Change Command

To replace a specified string of characters with another, use the change command:

[*/oldstring/newstring/ form lines*]

*/oldstring/newstring/*

*oldstring* is a string of characters to be replaced by the characters specified as *newstring*.

The delimiter, represented here by `/`, may be any nonalphameric character not occurring in the string **except** the following:

`] → ↓ ↑ ⍳ ? ⍝ ⎕ ∆ ∇ ∨ ∧ ⊤`

The terminating delimiter must be entered.

A change command in the format */oldstring/newstring/* changes the first occurrence of *oldstring* in every line that it appears.

*form*

Specifies the form of the search; it may be omitted. If provided, either or both of the following may be specified, with either operand first.

- `¨` specifies that all occurrences of the old string on any eligible line are to be changed. If the `¨` option is not specified, only the first occurrence of *oldstring* on any eligible line is changed.
- `N` specifies that the characters in a string represent an APL2 name. Valid matches exclude any strings of characters that are not APL2 names.

*lines*

Specifies the lines to be affected by the change command. If no lines are specified, all lines are affected by the command. This operand must follow any specification of form.

Only one of the following may be specified for *lines*:

- *n1 n2 n3 . . .*, specifies a vector of line numbers
- *n1*-*n2*, specifies a range of lines
- *n*-, specifies all lines in the object beginning with line *n*
- -*n*, specifies all lines in the object through line *n*.

Any form operand(s) must precede the lines operand; blanks between operands are not necessary.

***Caution:*** No text should follow the change command. Use the ERASE EOF key, as necessary, to delete the text following the command.

***Sample Command:*** The following forms are equivalent commands:

```
[/TVAR/XV/ ¨ N 6-]
[/TVAR/XV/ N ¨ 6-]
[/TVAR/XV/N¨6-]
```

In the example, `/` is the delimiter, and the APL2 name `TVAR` is to be changed to `XV` wherever it appears in the object. `¨` indicates that all occurrences on a line are to be changed, `N` indicates that the old string is an APL2 name, and 6- indicates that all lines of the object, beginning with line 6, are to be affected by the command.

***Change Command Useful for a Long Line:*** To insert characters in the middle of a long line, especially one that continues to a second line, use the change command for a single line. For example:

```
[/VAR/VARIABLE/ 17]
```

Type the change command on any line except a continuation line, then press Enter.

## Copying Lines Into a Definition—Get Command

To copy lines into a definition, use the get command. The get command gets the lines specified from the character representation of the object named and inserts them at the point of the command in the object being edited.

`[∧ name lines]`

*name*

Any simple character variable of rank 2 or less, an unlocked defined function, or an unlocked defined operator may be specified. Scalars and vectors are treated as one-row matrixes.

If the name is omitted, lines are copied from the current object being edited.

*lines*

Specified as with display and delete commands, lines may be individual line numbers or a range of numbers. (See "Displaying an Object" on page 400.) Lines are selected with index origin 0. Thus, for example, a `3` selects the fourth row of a matrix.

If lines are not specified, the entire object is inserted.

If the object has 0 rows, a `DEFN ERROR` is generated. An object will have 0 rows if:

- It is locked.
- It is a 0 by *n* empty matrix.
- It is undefined.

If an array is an *n* by 0 empty array (where *n* is not equal to 0), *n* lines without text are inserted.

## Copying or Moving Lines within a Definition

Lines of an object may be copied by changing their line numbers, or by using the get command. To move lines, simply copy them, then delete the originals.

- To make a copy of a line, change its line number.
- To copy a line with text changes, change the line number and the text.

When you press Enter, the copied line appears as the line number specified, but the original text and line number are unaffected.

To copy several consecutive lines, change the line number of the first line to be copied and blank out the line number field on subsequent lines to be copied. For example, to copy lines 4 through 6 and place after line 8:

**Original Display**

```
[3]    LINE 3
[4]    LINE 4
[5]    LINE 5
[6]    LINE 6
[7]    LINE 7
[8]    LINE 8
[9]    LINE 9
```

**Lines Marked for Copy**

```
[3]           LINE 3
[8.1]         LINE 4
              LINE 5
              LINE 6
[7]           LINE 7
[8]           LINE 8
[9]           LINE 9
```

**After the Copy**

```
[3  ]    LINE 3
[4  ]    LINE 4
[5  ]    LINE 5
[6  ]    LINE 6
[7  ]    LINE 7
[8  ]    LINE 8
[8.1]    LINE 4
[8.2]    LINE 5
[8.3]    LINE 6
[9  ]    LINE 9
```

To copy a block of lines that occupy more than one screen or are not displayed on the current screen or to copy from another object, use the put and get commands (see “Copying Lines Into a Definition—Get Command” on page 409 and “Copying Lines From a Definition—Put Command”).

## Copying Lines From a Definition—Put Command

To copy lines from a definition, use the put command. The put command takes the character representation of specified lines from the object being edited and creates in the active workspace a character matrix with the specified name. Line numbers are not part of the created matrix.

[∨ *name lines*]

*name*
The name is constructed following the rules for names. If it is the same as a variable in the workspace, the value will be replaced. If it is the same as a defined function or operator in the workspace, a *DEFN ERROR* is generated.

*lines*
Specified as with display and delete commands, lines may be individual line numbers or a range of numbers. (See "Displaying an Object" on page 400.) If lines are not specified, the entire object is used.

## Editing Multiple Objects

Multiple objects can be viewed and edited concurrently by dividing the screen into horizontal segments. Each segment has the same structure as when a single definition is opened and is treated as a separate definition area. All the editing facilities described in "Editor 2 Commands" on page 396 apply to each segment.

### Opening Screen Segments

Segments are opened from within the editor by issuing an appropriate ∇ or ⍫ command (with no text to the left of ∇ on the line). The segment begins on the line at which the ∇ is entered.

If ∇ is entered on a line at which lines of an object are currently displayed, lines of the currently displayed segment are replaced by lines of the segment just opened. The lines of the original segment can still be displayed in their own segment with the display or scroll commands.

A new segment may not be opened on the last line of the screen.

### Working with Multiple Segments

Commands that change or delete lines can affect the entire object. You may be unable to see some of the changes until you display more lines of the object. To avoid changing or deleting lines you cannot see, you can limit such commands to the lines you can see by specifying only the numbers of the lines currently displayed.

When Enter or a defined function key is pressed, the editor scans all lines on the terminal screen for input and processes any modified line. More than one line can be typed before processing is requested. The lines may be typed on any line of the screen segment where they are to take effect. When the screen is split, an edit command affects only the object being edited in the screen segment where the command is issued (although the entire screen is processed). The editor ignores blank lines.

Entering the closing ∇ or pressing PF3 when the cursor is in a segment releases that segment. Any screen segment immediately above the released segment expands to include the screen rows released. If the released segment is the top segment on the screen, the segment immediately below expands upward to include the screen rows released. If only one segment is open, the closing ∇ ends the editor and returns to immediate execution under the system.

Signaling an interrupt ends all segments of Editor 2 without establishing any objects.

## Immediate Execution in Editor 2

An APL2 statement or defined function or operator can be executed within Editor 2 through the execution command (⍎) entered on any line of a segment. The execution command has the form:

`[⍎]`*expression*

The expression is evaluated, and the expression and result remain displayed, as illustrated below:

**Command Is Typed**

```
[⍎]6+⍳4
```

**Enter Is Pressed**

```
[⍎]    6+⍳4
7 8 9 10
```

The result of an executed expression can be made part of the definition. This is done by typing over any character in the result, or inserting or deleting characters in it.

System commands cannot be executed in Editor 2.

To execute an object that you have been editing in another segment, use `[∇]` or PF6 to fix the definition in the workspace. Then execute the object with `[⍎]` expression.

# Chapter 10. System Commands

APL2 provides three types of system commands for:

- Storing and retrieving objects and workspaces
- Using system services and information
- Using the active workspace

This chapter describes all system commands alphabetically. System-specific commands are labeled. The structure of the system command is given at the beginning of each command description, as shown in Figure 55.

)*SAVE* [[*library*] *workspace*]

*Figure 55. Structure of a Command*

Brackets indicate that the enclosed item is an optional parameter--they are not entered as part of the command. Any parameter not shown in brackets must be entered. Parameters must be entered in the order shown.

In this chapter, command names and keywords are shown in uppercase, and fields to be substituted with user data are shown in lowercase. In actual usage, command name and keyword characters may be entered in any case. Some user-supplied fields in system commands refer to APL objects (variables, functions, and operators). These fields are case sensitive, since APL2 treats uppercase object names as distinct from lowercase object names. For other user-supplied fields (such as file, workspace, or editor names), case sensitivity varies by operating system.

Figure 56 displays the APL2 system commands, grouped according to use.

*Figure 56 (Page 1 of 2). APL2 System Commands*

| | | |
|---|---|---|
| **Storing and Retrieving Objects and Workspaces**<br>These system commands move data between the active workspace and other workspaces or external files. They also enable a user to migrate workspaces. | )*LIB* | List workspace names |
| | )*LOAD* | Retrieve workspace from library |
| | )*SAVE* | Save active workspace |
| | )*CLEAR* | Activate a clear workspace |
| | )*WSID* | Query/assign workspace identifier |
| | )*COPY* | Copy objects into active workspace |
| | )*PCOPY* | Copy objects into active workspace |
| | )*MCOPY*[1] | Migrate VS APL objects into active workspace |
| | )*IN* | Read transfer file into active workspace |
| | )*PIN* | Read transfer file into active workspace |
| | )*OUT* | Write objects to transfer file |
| | )*DROP* | Delete a library workspace |

*Figure 56 (Page 2 of 2). APL2 System Commands*

| | | |
|---|---|---|
| **Using the Active Workspace**<br>These system commands provide information about the active workspace, specify the APL2 editor to be used, and remove variables, defined functions, and defined operators from the active workspace. | `)NMS` | List names |
| | `)VARS` | List variables |
| | `)FNS` | List functions |
| | `)OPS` | List operators |
| | `)ERASE` | Delete objects |
| | `)SIS` | Display state indicator |
| | `)SI` | Display state indicator |
| | `)SINL` | Display state indicator |
| | `)RESET` | Clear state indicator |
| | `)PBS`[2] | Printable backspace character |
| | `)EDITOR` | Query/specify editor |
| | `)QUOTA`[2] | Display resource limits |
| | `)SYMBOLS` | Query/modify symbol table size |
| **System Commands for System Services/Information**<br>These system commands provide diagnostic information and access to operating system commands. | `)OFF` | End APL2 session |
| | `)CONTINUE` | Save active workspace and end session |
| | `)HOST` | Execute a host system command |
| | `)MORE` | List additional diagnostics |
| | `)TIME`[3] | Display current time |
| | `)MSG`[3] | Send message to another user |
| | `)OPR`[3] | Send message to the system operator |
| | `)CHECK` | Diagnose errors |

**Note:**

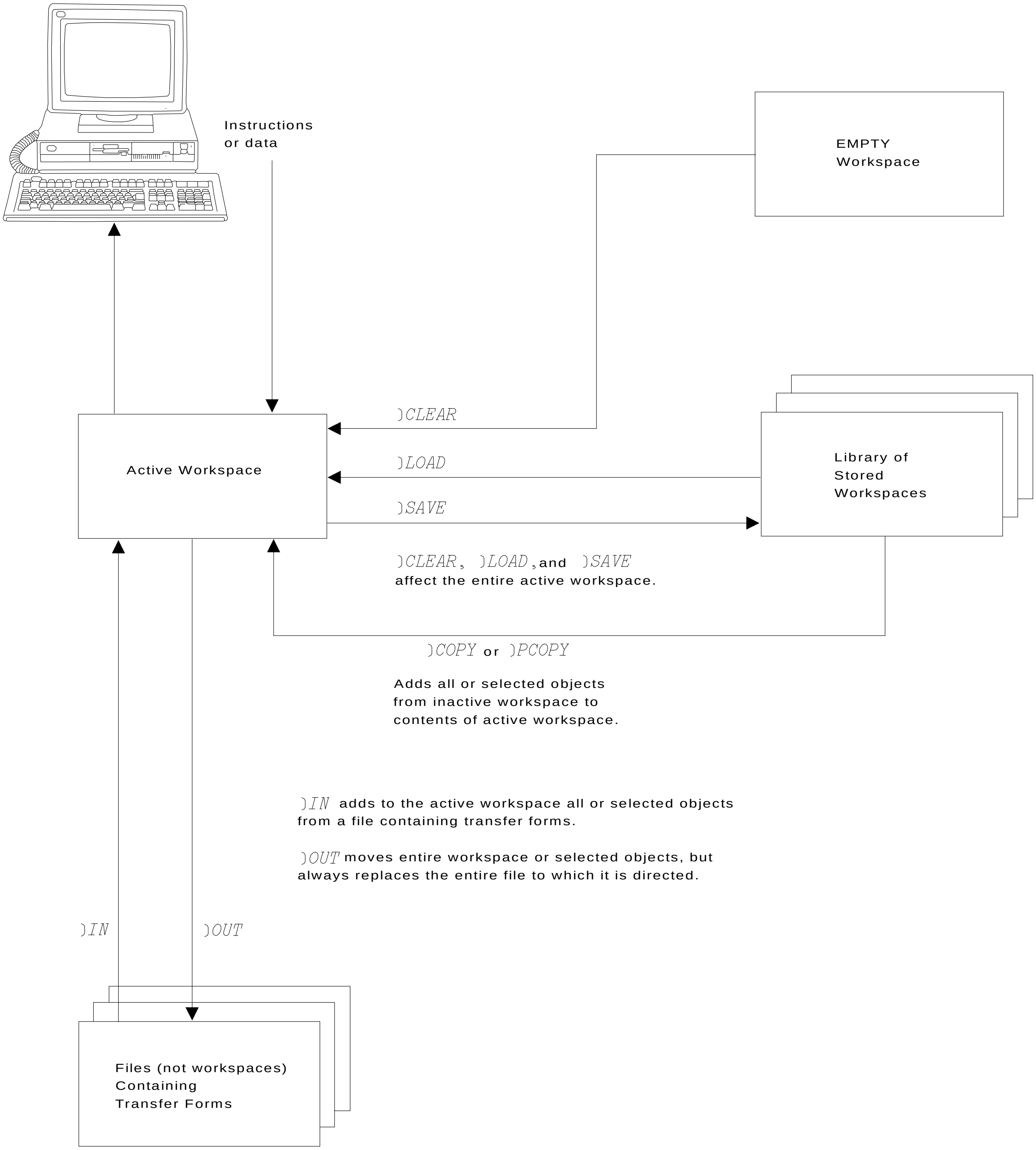
1. APL2/370 only. Refer to the *APL2 Migration Guide* for additional information.
2. APL2/370 only.
3. These system commands are provided on CMS and TSO for compatibility with older APL products.

## Storing and Retrieving Objects and Workspaces

APL2 provides several system commands for storing and retrieving objects and workspaces. These commands:

- Store workspaces, list stored workspace names, and remove stored workspaces.
- Retrieve the contents of stored workspaces.
- Write and read objects to and from transfer files.

Figure 57 on page 415 summarizes the actions of several of these commands.



)CLEAR, )LOAD, and )SAVE affect the entire active workspace.

)COPY or )PCOPY

Adds all or selected objects from inactive workspace to contents of active workspace.

)IN adds to the active workspace all or selected objects from a file containing transfer forms.

)OUT moves entire workspace or selected objects, but always replaces the entire file to which it is directed.

*Figure 57. How Selected System Commands Affect the Active Workspace*

## Common Command Parameters—Library, Workspace

The system commands that move objects to and from libraries may require the library number, as well as the workspace name. These three parameters are explained below without the detailed descriptions of the individual commands. Other parameters are described with the relevant commands in the detailed sections of this chapter.

| Parameter | Meaning |
|---|---|
| *library* | Is the number of the library to be accessed by the command. You must enter a number if the library is not your default library.<br>The library structure is dependent on the host system, but differences in structure do not affect the way the system commands work. |
| *workspace* | Is the name of the workspace to be accessed. Workspace names may contain up to eight characters (A through Z, 0 through 9), and they must begin with a character.<br>The underlying workspace structure is also dependent on the host system, but the differences in structure do not affect the way the system commands work. |

A quoted filename format is supported on some platforms.

## System Services and Information

APL2 provides system commands for ending an APL2 session, gaining access to host system services, and obtaining information from the system. These system commands are used to:

- End the APL2 session
- Issue host system commands
- Obtain additional information on error reports from system commands

## Using the Active Workspace

APL2 provides several system commands for using the active workspace. These commands:

- List or erase global objects (variables, defined functions, and defined operators)
- Display or reset the state indicator
- Set or query the editor or printable backspace

### Common Parameters—First, Last

For the system commands that list objects, you may request an alphabetic range for the list. The parameters for specifying the range are explained below rather than repeated with the detailed descriptions of the individual commands.

| Parameter | Meaning |
|---|---|
| *first* | Begins the list of names with the characters shown for this parameter. *first* may be a single character or a set of characters. |
| *last* | Ends the list of names after names beginning with this parameter have been displayed. *last* may be a single character or a set of characters. |

*First* and *last* are separated by a hyphen. Either name (or both) may be omitted to indicate that the range is unbounded at that end. For example:

```
)FNS QU-  (all names from QU to the end of the list)
)FNS -QA  (all names from the beginning of the list
           through the last one that begins with QA)
)FNS Q-T  (all names beginning with a letter from Q through T
```

The atomic vector (⎕AV) character sequence (see Figure 69 on page 471 for EBCDIC and Figure 68 on page 470 for ASCII) determines the order of the names listed as a result of any of the following commands:

| | |
|---|---|
| )NMS | list global names |
| )VARS | list global variable names |
| )FNS | list defined function names |
| )OPS | list defined operator names |
| )LIB | list workspace names |

All names reported in the list begin at eight-column intervals. A multiple-row list forms columns if the names are short enough to fit every eight columns.

The examples of these system commands assume a workspace with the contents shown in Figure 58.

**CONTENTS OF EXAMPLE WORKSPACE USED WITH )NMS, )FNS, )VARS, AND )OPS**

**Variables**

```
CHANGE_ACTIVITY COIBM   DCS     GPAPL2  GPDESC  TIMER
```

**Functions**

```
ABSTRACT        ASSOC   BIN     COMB    DEC2HEX DESCRIBE
DO      EXAMPLE EXAMPLES        EXPAND  FC      GCD     HELP
HEX2DEC HILB    HOW     IOTAU   LFC     PACK    PALL    PER
PERM    PO      POL     POLY    POLYB   REP     REPLICATE
SORTLIST        TIME    TRUTH   TYPE    UNIQUE  UNPACK
```

**Operators**

```
AND     COMMUTE CR      EL      ELSE    ER      FAROUT  HEX
IF      NOP     PAD     PL      POWER   PR      TRACE   TRAP
TRUNC   ZERO
```

*Figure 58. Sample Workspace for System Command Examples*

## `)CHECK`—Diagnostic Information

`)CHECK` command options common to all platforms are described here. For information about the additional options supported in APL2/370, see *APL2/370 Diagnosis Guide*.

The common options are grouped as follows:

**Workspace validation**

```
)CHECK WS [ON|OFF]
```

**Tracing functions**

```
)CHECK TRACE STMT
)CHECK TRACE OFF
```

**Forcing dumps**

```
)CHECK DUMP
```

### Workspace Validation

```
)CHECK WS
```

Causes an immediate comprehensive check of the workspace. This is independent of the other settings of `)CHECK`.

Diagnostic information is produced as APL2 output if any inconsistency is found and the active workspace is replaced with a clear workspace (`CLEAR WS`).

**Note:** `)CHECK WS` is done automatically when the `)LOAD` and `)SAVE` commands are issued.

```
)CHECK WS ON
```

Causes a comprehensive check of the workspace at the completion of every primitive function and prior to the processing of a new line of input from the keyboard or AP 101 stack.

A minidump of selected areas of APL2 storage is produced as APL2 output if any inconsistency is found and the active workspace is replaced with a clear workspace (`CLEAR WS`).

**Note:** Using this command causes significant performance degradation.

```
)CHECK WS OFF
```

Resets the command `)CHECK WS ON`. The `)LOAD` and `)CLEAR` commands also reset `)CHECK WS ON`.

## Tracing Functions

```
)CHECK TRACE STMT
```

Displays the text of each statement of defined functions or operators as the statement is about to be executed. The text of the statement is preceded in the trace by the current value of the second element of `⎕AI`.

```
)CHECK TRACE OFF
```

Resets the `)CHECK TRACE STMT` request.

## Forcing Dumps

```
)CHECK DUMP
```

Produces a small minidump of selected areas of APL2 storage and replaces the active workspace with a clear workspace (`CLEAR WS`).

## )CLEAR—Activate a Clear Workspace

```
)CLEAR
```

`)CLEAR` replaces the current active workspace with a clear workspace. When `)CLEAR` is executed, these actions take place:

- All shares are retracted.
- The contents of the active workspace are discarded and the state indicator is cleared.
- Most system variables are set to their initial default values. These default values are shown in Figure 59 on page 421.
- The active workspace has no name.

For example:

```
      )CLEAR
CLEAR WS

      )WSID
IS CLEAR WS
```

***System Variables Not Reset:*** The values of the system variables `⎕NLT` (national language translation), `⎕PW` (printing width), and `⎕TZ` (time zone) and the settings established by `)PBS` (printable backspace) and `)EDITOR` (named editor) are not reset. Their values are retained, except that if any of the three system variables had been localized before `)CLEAR`, their value is restored to their last valid global value.

*Figure 59. Environment Reset by CLEAR Command*

| Symbol | Meaning | Default |
|---|---|---|
| ⎕L | Left argument | No value |
| ⎕R | Right argument | No value |
| ⎕CT | Comparison tolerance | 1E¯13 |
| ⎕EM | Event message | 3 0⍴' ' |
| ⎕ET | Event type | 0 0 |
| ⎕FC | Format control | '.,*0_¯' |
| ⎕IO | Index origin | 1 |
| ⎕LC | Line counter | ⍳0 |
| ⎕LX | Latent expression | '' |
| ⎕PP | Printing precision | 10 |
| ⎕PR | Prompt replacement | ' ' |
| ⎕RL | Random link | 7*5 (that is, 16807) |
| ⎕SVE | Shared variable event | 0 |
| ⎕WA | Workspace available | Depends on installation and invocation options |
| )WSID | Workspace name | None (that is, CLEAR WS) |
| )SI | State indicator | Cleared |

## )CONTINUE—Save Active Workspace and End Session

```
   )CONTINUE
```

)*CONTINUE* saves the active workspace in the default private library under the name *CONTINUE* and ends the APL2 session. APL2 displays connect and processor time, and control then returns to the host system.

Each time you start APL2, the *CONTINUE* workspace is loaded automatically, as indicated by the *SAVED* *date/time* message, and any latent expression in the *CONTINUE* workspace is executed. If, however, you have used the *INPUT* parameter in your invocation of APL2, the input specified there replaces the load of the *CONTINUE* workspace.

```
      )WSID
DUMMY
      )VARS
DRY     FAT     IOD     ME      PRO     SALT
      )CONTINUE
 1993-05-21 11.30.56 (GMT-4) CONTINUE
```

Next APL2 session:

```
SAVED 1993-05-21 11.30.56 (GMT-4)
      )WSID
CONTINUE
      )VARS
DRY     FAT     IOD     ME      PRO     SALT
```

After being established by )*CONTINUE* or )*SAVE CONTINUE*, the *CONTINUE* workspace remains in your library unless explicitly dropped.

***Caution:*** Any *CONTINUE* workspace in the library is replaced by the active workspace whenever the )*CONTINUE* command is executed. The name of the active workspace does not need to be *CONTINUE* for this to occur.

## )COPY—Copy Objects into the Active Workspace

```
)COPY [library] workspace [names]
```

The results of the )COPY command depend on whether object names are included in the command:

- If the command lists names, the named global objects either are added to the active workspace or replace global objects of the same name that are currently in the active workspace.
- If the command does not include names, all global objects from the workspace are added to the active workspace or replace global objects of the same name that are currently in the active workspace.

In either case, the definitions of local objects are not affected by the )COPY command. The only system objects that can be copied are ⎕CT, ⎕FC, ⎕IO, ⎕LX, ⎕PP, ⎕PR, and ⎕RL. These system objects are also copied when object names are not included in the command.

The examples below show the )COPY command and the system response to it.

```
      )COPY LANGMAN SHOW TEST TESTING ⎕LX
SAVED 1993-05-21 13.56.08 (GMT-4)

      )COPY LEARN
SAVED 1993-05-21 17.23.58 (GMT-4)
```

The SAVED message indicates the date and time the workspace was last saved.

If an object being replaced is a shared variable, its share is retracted. If an object being replaced is a suspended or pendent function, SI WARNING is reported.

Defined function and operator definitions are copied without any associated trace and stop controls. Because only the definitions are copied, there is no effect on the copy if these defined functions or operators are suspended or pendent in the library workspace.

The )COPY command requires space in both the source workspace and the active workspace in order to copy each of the objects. If there is insufficient room in either workspace to copy an object, )COPY continues to the next object, and so forth, until all of the objects that can be copied are brought into the active workspace. If some objects cannot be brought in, WS FULL and NOT COPIED are displayed along with the names of the uncopied objects.

To circumvent WS FULL problems on )COPY, it may be necessary to invoke APL2 with a larger workspace.

If objects to be copied cannot be found in the source workspace, NOT FOUND is displayed along with the names of those objects.

***Copying Versus Loading a Workspace:*** Copying an entire workspace into a clear workspace is **not** equivalent to using the `)LOAD` command for the same workspace. The `)COPY` command requires the system to do more work than the `)LOAD` command, and it omits some potentially important control information that may be in the stored workspace. The following are not copied:

- Local variables, functions, or operators that are part of suspended or pendent functions or operators in the source workspace
- The state indicator, which lists where evaluation halted in the source workspace
- System variables associated with suspension (⎕EM, ⎕ET, ⎕L, ⎕R, ⎕LC, ⎕SVE)

## Parameters

The introduction to this chapter (page 416) gives the general requirements for the library and workspace parameters. The examples below demonstrate their use.

```
      )COPY 1 EXAMPLES TRACE PL XI MIX
SAVED 1993-05-21 12.21.14 (GMT-4)

      )COPY 1010 TOOLBOX LOCKOUT SUMCOL MODIFY
SAVED 1993-05-21 14.02.54 (GMT-4)
```

***Additional Parameter Information:*** In addition to the general parameter requirements, the following information applies to `)COPY`.

| **Parameter** | **Meaning** |
|---|---|
| *names* | lists valid global object names. One or several names may be included.<br>If the name list includes the name of a simple character scalar, vector, or matrix enclosed within parentheses, its rows are interpreted as APL2 names, and these objects are copied instead of the array itself. The array may also contain its own name and then it is copied as well. This form of copying is called *indirect copy*. Indirect copying offers a convenient way to copy a group of objects simultaneously. Figure 60 shows an example. |

```
      )WSID
TOOLS
      PGRP←⊃'PROMPT' 'EMPTY' 'IF' 'CHARACTER' 'PGRP'
      )SAVE
 1993-05-21 22.02.39 (GMT-4) TOOLS

      )CLEAR
CLEAR WS
      )COPY TOOLS (PGRP)
SAVED 1993-05-21 22.02.39 (GMT-4)
```

All objects named in the matrix *PGRP* are copied into the workspace.

```
      )NMS
CHARACTER.3     EMPTY.3 IF.3    PROMPT.3
PGRP.2

      PGRP
PROMPT
EMPTY
IF
CHARACTER
PGRP
```

*Figure 60. Use and Effect of Indirect Copy*

## )*DROP*—Remove a Workspace from a Library

)*DROP* [*library*] *workspace*

Execution of )*DROP* deletes the named workspace from the indicated library.

```
      )DROP THISWS
 1993-05-21 22.17.56 (GMT-4)
```

The message indicates the current time, date, and time zone.

To drop a workspace, you must have write access. Also, note that only one workspace can be dropped at a time.

### Parameters

The introduction to this chapter (page 416) gives the general requirements for the library and workspace parameters. The examples below demonstrate their use.

```
      )DROP 1008 WSONE
 1993-05-21 22.18.55 (GMT-4)

      )DROP 10 CLASS
 1993-05-21 22.19.34 (GMT-4)
```

## )EDITOR—Query or Select Editor to be Used

```
)EDITOR
)EDITOR 1
)EDITOR 2 [name]
)EDITOR name
```

APL2 provides the following editors for defining and editing functions and operators:

| | |
|---|---|
| 1 | Line editor |
| 2 | APL full-screen editor |
| 2 *name* | Named APL editor |
| *name* | Named system editor |

Use of the editors is described in Chapter 9, “The APL2 Editors” on page 375.

When you start APL2, the initial setting for the editor is 1. To determine the current setting, enter `)EDITOR`:

```
      )EDITOR
IS 1
```

To change the editor setting, enter `1` for the line editor or `2` for the full-screen editor. For example:

```
      )EDITOR 2
      )EDITOR
IS 2
      )EDITOR XEDIT
      )EDITOR
IS XEDIT
```

The editor setting is a session parameter. It is not affected by the `)CLEAR` or `)LOAD` commands.

## )ERASE—Delete Objects from the Active Workspace

```
)ERASE names
```

| Parameter | Meaning |
|---|---|
| *names* | Is a list of valid global object names. One or several names may be included.<br>**Note:** If the name list includes the name of a simple character scalar, vector, or matrix enclosed within parentheses, its rows are interpreted as APL2 names, and these objects are erased instead of the array itself. The array may also contain its own name, which is then erased as well. This form of erasing is called *indirect erase*. Indirect erasing offers a convenient way to erase a group of objects simultaneously. Figure 62 on page 430 shows an example of indirect erasing. |

)ERASE removes the named global objects (variables, defined functions, and defined operators) from the active workspace. For example:

```
      )NMS R
ROUND.3 SHOW.4  STATS.3 SUMCOL.3        TOTS.2
TRACE.4 TRAP.4  TRPLGRP.2       TRUNC.4 UWAY.2

      )ERASE SUMCOL TOTS UWAY STATS

      )NMS R
ROUND.3 SHOW.4  TRACE.4 TRAP.4  TRPLGRP.2
TRUNC.4
```

If an object being erased is a shared variable, its share is retracted. If the name list includes a defined function or defined operator that is pendent or suspended, that object is erased; however:

- SI WARNING is *not* reported and the stack is not affected.
- The defined function or operator in the state indicator retains its original definition until its execution is completed or until the state indicator is cleared (using → or )RESET).
- The name previously associated with the function or operator now has no value, and further execution and editing of the original definition is not possible.

Figure 61 on page 429 demonstrates the effect of erasing a suspended defined function.

```
    ∇
[0]   F
[1]   'LINE 1'
[2]   2÷0
[3]   'LINE 3'
    ∇
      F
LINE 1
DOMAIN ERROR
F[2]  2÷0
      ^^

      )SI
F[2]
*
      )ERASE F

      )SI
F[2]         Definition of F is retained
*            in the state indicator.

      →3     Execution of F resumes
LINE 3       at line 3.

      F      Attempt to invoke F results
VALUE ERROR  in an error because the definition
      F      of F no longer exists.
      ^
```

*Figure 61. Effect of Erasing a Suspended Defined Function*

```
      )CLEAR
CLEAR WS

      MAT←2 3ρ'FORYOU'
      SCA←5
      VEC←5 6 7
      NEST←'HI' 'GUY'
      GROUP←3 4ρ'NESTVEC SCA '
      CHAR←'Y'

      GROUP
NEST
VEC
SCA
      )VARS
CHAR    GROUP   MAT     NEST    SCA     VEC

      )ERASE CHAR (GROUP)

      )VARS
GROUP   MAT
```

*Figure 62. Use and Effect of Indirect Erase*

⎕EX Expunge (page 289) eliminates the currently active objects named in its argument and may be used to eliminate certain system variables.

## )FNS—List Indicated Objects in the Active Workspace

)FNS [*first*] [-] [*last*]

)FNS displays an alphabetic list of the global defined functions in the active workspace.

See the introduction to this chapter (page 416) for explanations of the parameters first and last. The following examples illustrate the commands used to display partial contents of the sample workspace shown in Figure 58 on page 417.

```
      )FNS P-
PACK    PALL    PER     PERM    PO      POL     POLY
POLYB   REP     REPLICATE       SORTLIST        TIME
TRUTH   TYPE    UNIQUE  UNPACK

      )FNS -I
ABSTRACT        ASSOC   BIN     COMB    DEC2HEX DESCRIBE
DO      EXAMPLE EXAMPLES        EXPAND  FC      GCD
HELP    HEX2DEC HILB    HOW     IOTAU

      )FNS I-P
IOTAU   LFC     PACK    PALL    PER     PERM    PO
POL     POLY    POLYB

      )FNS P-I

      )FNS PE-PE
PER     PERM
```

## )*HOST*—Execute a Host System Command

)*HOST* [ *command* ]

)*HOST* allows you to execute host system commands from within APL2.

)*HOST* passes the given command to the system and displays the system return code. If you enter )*HOST* with no parameters, the name of the host system is displayed.

## )*IN*—Read a Transfer File into the Active Workspace

)*IN* *file* [*names*]

A transfer file may be created by using )*OUT* (see page 442), by using auxiliary processors, or by a process external to APL2.

The result of the )*IN* command depends on whether object names are included in the command:

- If the command lists object names, the transfer forms of these objects are read from the named transfer file and are defined in the active workspace.
- If the command does not list object names, the entire transfer file is read and its objects are defined in the active workspace.

The format of the transfer file created by )*OUT* is shown in Appendix B, "APL2 Transfer Files and Extended Transfer Formats" on page 484. )*IN* ignores sequence numbers in the transfer file (columns 73 through 80).

If )*IN* is successful, no messages are displayed, as shown below:

```
      )IN TOOLS

      )IN TRIAL PAL ROUND PIG
```

If a name conflict occurs, the object from the transfer file replaces the one currently in the active workspace.

If the object being replaced is a shared variable, its share is retracted. If the object being replaced is a suspended or pendent function, no warning is reported.

### Parameters

The following information applies to )*IN*.

| Parameter | Meaning |
|---|---|
| *file* | Is the name of a transfer file, following the file naming conventions and defaults of the operating system. |
| *names* | Are names of objects to be read and defined in the active workspace. Names may include system variables if these are present in the transfer file. |

# )*LIB*—List Workspace Names in a Library

`)LIB [library] [first ] [-] [last]`

)*LIB* displays an alphabetic list of workspace names, according to these conditions:

- All names in the library list begin at nine-column intervals so that a multiple-row list forms columns (if ⎕*PW* is appropriate to your display device.)
- The collating sequence gives alphabetic characters higher significance than numeric characters.

For example, to list the workspaces in your private library:

```
      )LIB
BUDGET   BUDGET2  ESTIMATE GENES    INFOEST
LANGMAN  LEARN    OUTTEST  SCHEDULE STATUS
TOOLS    UISAMPLE
```

| **Note:** If the library exists but contains no workspaces, an empty list is displayed.

## Parameters

The introduction to this chapter (page 416) gives the general requirements for the parameters of the )*LIB* command. The examples below demonstrate their use:

```
      )LIB 1 D-E
DISPLAY  EXAMPLES

      )LIB 1010
ADDCUST  DICKNICK EXECMEAN PAULS    SBIC2
TOOLBOX
```

***Additional Parameter Information:*** In addition to the general parameter requirements, the following information applies to )*LIB* parameters:

| **Parameter** | **Meaning** |
|---|---|
| *first* | Provides a partial list of workspaces, starting with any that begin with the indicated letter or set of characters. |
| *last* | Ends the list of workspace names after names beginning with this parameter are displayed. *last* may be a single character or a set of characters. |

For example:

```
       )LIB S
SCHEDULE STATUS   TOOLS    UISAMPLE

       )LIB ST-
STATUS   TOOLS    UISAMPLE

       )LIB G-S
GENES    INFOEST  LANGMAN  LEARN   OUTTEST
SCHEDULE STATUS
```

See the introduction to this chapter, page 416, for more information on the parameters *first* and *last*.

# )LOAD—Bring a Workspace from a Library into the Active Workspace

)LOAD [*library*] *workspace*

See the introduction to this chapter (page 416) for information about the general requirements for the parameters *library* and *workspace*.

When )LOAD is issued:

- A duplicate of the indicated library workspace completely replaces the contents of the active workspace. The original copy of the workspace on the permanent storage device remains intact and in place.
- Any shared variables that were in the active workspace are retracted.

The example below shows a )LOAD command and the system response to it.

```
      )LOAD TOOLS
SAVED 1993-05-21 13.56.08 (GMT-4) 675K(615K)
```

The SAVED message indicates the time, date, and time zone when the workspace was last saved. Also reported may be the size of the active workspace after the )LOAD, and, in parentheses, the size of the workspace when it was last saved. This information is provided only if the load size differs from the saved size.

If the workspace was saved with a latent expression, specified by ⎕LX, the system executes the latent expression (⍎⎕LX) immediately after the )LOAD.

```
      )LOAD 1010 LEARN
SAVED 1993-05-21 17.23.58 (GMT-4)
HI. ARE YOU READY TO LEARN MORE APL2?
```

When a workspace is loaded, the active workspace assumes the name of that workspace, for example:

```
      )WSID
IS CLEAR WS
      )LOAD SCHEDULE
SAVED 1993-05-21 22.54.21 (GMT-4) 675K(783K)
      )WSID
SCHEDULE
```

**Note:** The current values of the session system variables in the active workspace ⎕NLT (national language translation), ⎕PW, (printing width) and ⎕TZ (time zone) and the settings established by )PBS (printable backspace) and )EDITOR (named editor) are not altered by a )LOAD command.

On some APL2 platforms, an additional parameter can be supplied to control the size of the active workspace when loading. Without the size parameter, the maximum workspace size is used.

```
      )LOAD STATUS
SAVED 1993-05-21 12.21.14 (GMT-4) 683K(1043K)
      ⎕WA      ⍝BYTES AVAILABLE IN THE WORKSPACE
605476
```

The size parameter specifies the size of the active workspace, which is reported as part of the *SAVED* message.

```
      )LOAD STATUS 100000
SAVED 1993-05-21 12.21.14 (GMT-4) 97K(1043K)
      ⎕WA
6036
```

The workspace is saved with the current workspace size, which is then reported within parentheses the next time the workspace is loaded.

```
      )SAVE
 1993-05-21 10.18.56 (GMT-4) STATUS
      )LOAD STATUS
SAVED 1993-05-21 10.18.56 (GMT-4) 683K(97K)
```

An error message is displayed if the size parameter is not large enough to accommodate the workspace.

```
      )LOAD STATUS 78000
SYSTEM LIMIT
CLEAR WS
```

*Figure 63. Use and Effect of Size Parameter*

## )MORE—List Additional Diagnostic Information

```
)MORE [number]
```

Error messages display one line of information. The command )MORE is used to request additional information about the error. The following are examples of the use of )MORE:

```
      ⎕AF←⎕AV                       A←'''
SYNTAX ERROR+                 SYNTAX ERROR+
      ⎕AF←⎕AV                       A←'''
      ^  ^                            ^
      )MORE                         )MORE
NAME CLASS                    ILL-FORMED LINE
```

In the case on the left, the additional message indicates that the assignment cannot complete because of the name class of ⎕AF. (Assignment requires a variable name, while ⎕AF is a function.) In the case on the right, the message indicates the character constant beginning at the caret is not formed properly. (Quote characters within a character constant must be doubled.) The plus sign on the error message indicates that more information is available.

In situations where you do not get an error message but do not get the expected response, )MORE may give information to help you diagnose the problem. For example, if a function merely returns an auxiliary processor return code, )MORE may provide more information.

)MORE must be used immediately after a message is displayed. If any input other than )MORE is entered, the information on the message is erased.

If no diagnostic information is available, a message is displayed indicating that no further information is available.

```
      )MORE
NO MORE INFORMATION
```

To display more than one message at a time, use the optional *number* parameter. For example, if you want to see the last three error messages issued, enter:

```
      )MORE 3
```

*APL2/370 Messages and Codes* lists the messages received from )MORE and suggests corrective actions.

## )*NMS*—List Names in the Active Workspace

```
)NMS [first] [-] [last]
```

`)NMS` displays an alphabetic list of the global objects (variables, defined functions, and defined operators) in the workspace.

Each name reported is followed by a dot and an integer indicating its name class:

| **Integer** | **Name Class** |
|---|---|
| 2 | Variable |
| 3 | Defined function |
| 4 | Defined operator |

These numbers are the same as those produced by `⎕NC` for these objects (see page 309).

See the introduction to this chapter, page 416, for explanations of the parameters *first* and *last*. The following example shows the `)NMS` display for the contents *A* through *E* of the sample workspace shown in Figure 58 on page 417.

```
      )NMS A-E
ABSTRACT.3      AND.4   ASSOC.3 BIN.3   CHANGE_ACTIVITY.2
COIBM.2 COMB.3  COMMUTE.4       CR.4    DCS.2   DEC2HEX.3
DESCRIBE.3      DO.3    EL.4    ELSE.4  ER.4    EXAMPLE.3
EXAMPLES.3      EXPAND.3
```

# )*OFF*—End APL2 Session

```
)OFF
```

)*OFF* ends the APL2 session. Any active workspace objects not previously saved are lost. Control returns to the host system.

## )*OPS*—List Indicated Objects in the Active Workspace

```
)OPS [first] [-] [last]
```

)*OPS* displays an alphabetic list of the global defined operators in the active workspace.

See the introduction to this chapter, page 416, for explanations of the parameters first and last. The following examples illustrate the commands used to display partial contents of the sample workspace shown in Figure 58 on page 417.

```
      )OPS -P
AND     COMMUTE CR      EL      ELSE    ER      FAROUT  HEX
IF      NOP     PAD     PL      POWER   PR

      )OPS I-
IF      NOP     PAD     PL      POWER   PR      TRACE   TRAP
TRUNC   ZERO

      )OPS I-P
IF      NOP     PAD     PL      POWER   PR

      )OPS P-I

      )OPS TR-TR
TRACE   TRAP    TRUNC
```

## )OUT—Write Objects to a Transfer File

)OUT *file* [*names*]

| **Parameter** | **Meaning** |
|---|---|
| *file* | Names the transfer file. The conventions governing the name (and name defaults) of the file, its location, control of access to it, and its permanence are all local conventions of the particular operating system under which APL2 runs. (See the appropriate workstation user's guide or *APL2/370 Programming: System Services Reference*.) |
| *names* | Names of objects whose transfer forms are to be written to the named file. |

The result of the )OUT command depends on whether object names are included in the command invocation:

- If the command lists object names, the transfer forms of the named objects are written to the named transfer file.
- If the command does not list object names, the transfer forms of all unshared variables, defined functions, and defined operators, and the system variables ⎕CT, ⎕FC, ⎕IO, ⎕LX, ⎕PP, ⎕PR, and ⎕RL are written to the named transfer file.

If the command is successful, no messages are displayed, as shown below:

```
      )OUT TOOLS ALTER TRACE SHOW

      )OUT TRIAL
```

System variables in addition to those listed above can be transferred with )OUT if specifically requested:

```
      )OUT SV ⎕PW ⎕TZ
```

In this case, only the named objects are transferred.

***Transfer File Format:*** A transfer file has fixed-length 80-character records. Either migration transfer forms or extended transfer forms of APL2 objects may be in the transfer file. See Appendix B, “APL2 Transfer Files and Extended Transfer Formats” on page 484 for format details.

Figure 64 shows a sample workspace written to a transfer file with )OUT. The file contains one function named G.

```
XA⎕LX-''                                    00000100
XA⎕IO-1                                     00000200
XA⎕PP-10                                    00000300
XA⎕CT-1E 13                                 00000400
XA⎕RL-16807                                 00000500
XA⎕FC-'.,≠0_ '                              00000600
XA⎕PR-1 ' '                                 00000700
*(1993 8 17 30 16)                          00000800
XFG ⎕FX 'Z-G X' '2+'                        00000900
```

*Figure 64. Transfer Form of a Workspace (Each record is 80 characters long.)*

**Warning:** )OUT does **not** add to an existing file. If a transfer file by the specified name already exists, its contents are entirely replaced by the transfer forms of objects in the current active workspace.

***Transferring the Most Local Version:*** The most local version of an object is transferred. Figure 65 shows the writing of a local object to a transfer file.

```
      I
GLOBAL
      FN
FN[2]              ⍝FUNCTION IS SUSPENDED

      I            ⍝VALUE OF LOCAL VARIABLE I
LOCAL

      )OUT TEST I  ⍝WRITE I TO A TRANSFER FILE
      →2           ⍝RESUME FUNCTION EXECUTION
END OF FUNCTION
      I            ⍝VALUE OF GLOBAL VARIABLE I
GLOBAL

      )IN TEST I   ⍝READ I FROM TRANSFER FILE
      I
LOCAL
```

*Figure 65. )OUT Writes Local Objects*

## )*PBS*—Query or Set the Printable Backspace Character (APL2/370 Only)

```
)PBS
)PBS ON
)PBS OFF
```

| **Parameter** | **Meaning** |
|---|---|
| *ON* | Turns on the printable backspace character. |
| *OFF* | Turns off the printable backspace character. |

If you are using a terminal that cannot enter all the APL2 characters, you must use the *printable backspace* to enter or edit any line containing one of the characters listed below.

The character _ is the printable backspace character. Within the context of the ten characters shown below, it tells the system to treat the characters entered to its right and left as overstruck, thus forming a single character.

For example, to enter or edit the depth or match symbol ≡ with the printable backspace character, enter either `=__` or `__=`.

The characters that are entered with the printable backspace follow:

| **Character** | **Entered As...** |
|---|---|
| ⌻ | `⎕_∘` |
| ⍸ | `⍳__` |
| ⍷ | `∊__` |
| ⌷ | `[_]` |
| ⍂ | `⎕_\` |
| ≡ | `=__` |
| ∵ | `¨_.` |
| ◊ | `<_>` |
| ⊢ | `[_-` |
| ⊣ | `-_]` |

The overstrike pairs may be entered in either order, with the intervening printable backspace character.

The initial setting is determined by your display. To determine the current setting, enter )*PBS*. To deactivate the printable backspace character, enter )*PBS OFF*.

```
      )PBS
IS _

      )PBS OFF
```

The printable backspace character is effective only in the context of the new APL2 characters. For example:

```
      )PBS ON
      C←'∊__'
      ⍴C
(empty)
      C←'↑_↓'
      ⍴C
3
```

**Note:** APL2 always treats a printable backspace combination as a single character, for example, when determining the width of a display under ⎕PW Printing Width, page 318.

If your terminal has the programmable symbol set (PSS), the new characters are always displayed in their true typographical form.

```
      A←'⎕_∘ ⍳__ ∊__ [_] ⎕_\ ._¨ =__ <_> [_- -_]'
      A
⌻ ⍸ ⍷ ⌷ ⍂ ∵ ≡ ◊ ⊢ ⊣
```

If )PBS is on and you do not have the PSS, the characters will display with the printable backspace character as they were entered.

```
      )PBS
IS _
      A←'⎕_∘ ⍳__ ∊__ [_] ⎕_\ ._¨ =__ <_> [_- -_]'
      A
⎕_∘ ⍳__ ∊__ [_] ⎕_\ ¨_. =__ <_> [_- -_]
```

If )PBS is off and your terminal does not have the PSS, the display depends on the character set built in to your display.

The printable backspace character is a session parameter. It is not affected by a )CLEAR or )LOAD command.

Cases of apparent ambiguity in the use of the printable backspace are resolved by taking as the printable backspace the first underscore that can be a printable backspace. For example:

```
1=__=2 3
```

is 1 match equal 2 3 (which yields a VALENCE ERROR), NOT 1 equal depth 2 3.

## )PCOPY—Copy Objects into the Active Workspace with Protection

`)PCOPY [library] workspace [names]`

)PCOPY is identical to )COPY in all respects except one:

> If the active workspace contains global objects with the same name as any that are requested to be copied, they are not copied and the old ones are not replaced.

The example below shows the )PCOPY command and the system response to it.

```
      )LOAD LEARN
SAVED 1993-05-21 17.23.58 (GMT-4) 675K(783K)

      )PCOPY UISAMPLE AVERAGE ROUND ADDTOTALS
SAVED 1993-06-20 16.23.32 (GMT-4)
NOT COPIED:  ROUND
```

Refer to )COPY, page 423, for details of the command syntax and results.

## )PIN—Read a Transfer File into the Active Workspace with Protection

```
)PIN file [names]
```

`)PIN`, like `)IN`, reads objects into the active workspace from a transfer file. The two commands are identical in all respects except one:

> `)PIN` will not transfer an object if another object of the same name already exists in the active workspace, whereas `)IN` replaces any object in the active workspace that has the same name as the object being transferred in.

If `)PIN` is successful, no messages are displayed, as shown below:

```
      )PIN TOOLS

      )PIN TOPS SIDES SPINS
```

If a name conflict does occur, the object from the transfer file is listed in a `NOT COPIED:` message, as shown below:

```
      )PIN WORK DONE OVER
NOT COPIED:  WORK
```

Refer to `)IN` on page 433 for details of the command syntax and results.

**Note:** System variables are also protected when using `)PIN`. Most or all of them will be included in the `NOT COPIED:` list, unless a specific name list is provided in either the `)PIN` command or the `)OUT` command that created the transfer file.

# *)QUOTA*—List Workspace, Library, and Shared Variable Quotas (APL2/370 Only)

```
)QUOTA
```

*)QUOTA* displays a report on the availability of your private library, workspaces, and shared variables. The report is shown and explained below:

```
      )QUOTA
LIB     3404800  FREE     735200
WS       618496  MAX      618496
SV           88  SIZE      32768
```

Each row of the report provides information on your library, workspace size, and shared variable capabilities, respectively. In some implementations, not all the information is available. For more information, see *APL2/370 Programming: System Services Reference*.

| Item | Meaning |
|---|---|
| *LIB* | Total amount of space (in bytes) in your library |
| *FREE* | The amount of space (in bytes) still available in your library for saving |
| *WS* | The default size (in bytes) in the active workspace |
| *MAX* | The maximum size workspace (in bytes) that may be requested (as with *)CLEAR* or *)LOAD*) |
| *SV* | The maximum number of variables that may be simultaneously shared |
| *SIZE* | The size (in bytes) of your shared storage |

## )RESET—Clear the State Indicator

```
)RESET [number]
```

)RESET is a synonym for )SIC.

***Clearing n Lines from the State Indicator:*** )RESET with *number* clears that number of lines from the state indicator and resets ⎕EM, ⎕ET, ⎕L, and ⎕R to values appropriate to the statement at the top of the state indicator after the reset. If this is a line stopped by an error, , ⎕L and ⎕R indicate the values of the function's arguments at which the error occurred, and ⎕EM and ⎕ET reflect the error. If the line did not stop because of an error, ⎕L and ⎕R have no value, ⎕EM is an empty matrix, and ⎕ET is 1 1 (interrupt). For example:

```
      )SI
GN[1]
FN[2]
*
*
      ⎕EM
DOMAIN ERROR
GN[1] Z←3÷0
       ^ ^
      )RESET 3
      )SI
*
      ⎕EM
DOMAIN ERROR
      1×⍳1.2
       ^^
      ⎕ET
5 4
      ⎕L
3
      ⎕R
0
      ⎕R
1.2
      ⎕L
VALUE ERROR
      ⎕L
      ^
```

⎕L has no value because the function interval has no left argument.

***Clearing the Entire State Indicator:*** If a number is not specified with the command, )RESET clears all suspended and pendent statements and editing sessions from the state indicator. For example:

```
       )SI
GN[1]
FN[2]
*
*
       )RESET
       )SI
```

This is equivalent to entering → (escape) until the state indicator is clear.

Because they are effectively local to functions in lines of immediate execution, )RESET without a number returns the system variables ⎕EM and ⎕ET to their initial values in a clear workspace and removes the values of ⎕L and ⎕R. )RESET also purges and contracts the internal symbol table.

See also “)SIC—Clear the State Indicator” on page 454.

## )SAVE—Save the Active Workspace in a Library

)SAVE [[*library*] *workspace*]

)SAVE stores a copy of the active workspace in the indicated library. )SAVE has one of the following effects on the library:

- If the named workspace does not exist in the library, )SAVE establishes it in the library.
- If the named workspace exists in the library, )SAVE replaces the current contents of the library workspace with the active workspace.
- If the named workspace exists in the library but is not the same as the name of the active workspace, the following error message is displayed:

  NOT SAVED: THIS WS IS *name*

The example below shows a )SAVE and the system response to it.

```
      )SAVE THISWS
1992-03-27 21.48.04 (GMT-4)
```

The message indicates the time, date, and time zone in effect when the workspace was saved.

Current values of any shared variables are saved in the stored copy even though they have not yet been referenced. The state indicator, current values of system variables, and stop and trace controls are also saved.

)SAVE does not affect the contents of the active workspace. However, the active workspace assumes the name given in the )SAVE command.

```
      )WSID
IS CLEAR WS
      )SAVE NEWWS
1992-03-27 21.50.45 (GMT-4)
      )WSID
IS NEWWS
```

### Parameters

The introduction to this chapter (page 416) gives the general requirements for the library and workspace parameters. The examples below demonstrate their use.

```
      )SAVE 10 CLASS
1992-03-27 21.40.23 (GMT-4)
```

```
      )SAVE 1008 WSONE
1992-03-27 21.49.23 (GMT-4)
```

***Additional Parameter Information:*** If you omit the workspace name and associated library number, they are supplied from the current workspace identification (see `)WSID`, page 460).

For example:

```
      )WSID
THISWS
      )SAVE
1992-03-27 21.51.09 (GMT-4) THISWS
```

**Note:** The system response includes the workspace name when it is omitted from the `)SAVE` command.

## )SI—Display the State Indicator

```
)SI [number]
```

The state indicator, discussed in “State Indicator” on page 355, is a list of:

- The calling sequence of defined functions and defined operators (and their pertinent line numbers).
- Asterisk(s) for all immediate execution expressions that did not complete, either because of an error in the expression or because the function invoked by the expression is pendent or suspended.

The command `)SI` without a number specified displays data from each line of the state indicator. If a number is provided, the command does not display more than that number of lines of the state indicator.

The `)SI` command is similar to `)SIS` but it does not list the statement that was being executed at the time the line was added to the state indicator. `)SI` lists the defined functions and defined operators (and their pertinent line numbers) in the state indicator, and an asterisk for all immediate execution expressions that did not complete. For example:

```
      )SI
GN[1]
FN[2]
*
*

      )SI 2
GN[1]
FN[2]
```

If a definition line appears in the state indicator, the value within brackets indicates the status of the object:

| | |
|---|---|
| [Positive integer I] | Execution is suspended at line I. Execution can be resumed by →⍳0. |
| [Negative integer I] | Execution is suspended at line \|I. Execution can be restarted by →⎕LC or →*n*, where *n* is a line number. |
| [*blank*] | Execution is suspended by a line, but which one cannot be determined. Execution can be neither restarted nor resumed. |
| []∇ | The object is being edited. |

***Clearing the State Indicator:*** `)SIC`, page 454, and `)RESET`, page 449, both clear the state indicator as does → (escape), described under “Clearing the State Indicator” on page 357.

## )SIC—Clear the State Indicator

```
)SIC [number]
```

***Clearing n Lines from the State Indicator:*** )SIC with *number* clears that number of lines from the state indicator and resets ⎕EM, ⎕ET, ⎕L, and ⎕R to values appropriate to the statement at the top of the state indicator after the reset. If this is a line stopped by an error, , ⎕L and ⎕R indicate the values of the function's arguments at which the error occurred, and ⎕EM and ⎕ET reflect the error. If the line did not stop because of an error, ⎕L and ⎕R have no value, ⎕EM is an empty matrix, and ⎕ET is 1 1 (interrupt). For example:

```
      )SI
GN[1]
FN[2]
*
*
      ⎕EM
DOMAIN ERROR
GN[1] Z←3÷0
       ^ ^
      )SIC 3
      )SI
*
      ⎕EM
DOMAIN ERROR
      1×⍳1.2
       ^^
      ⎕ET
5 4
      ⎕L
3
      ⎕R
0
      ⎕R
1.2
      ⎕L
VALUE ERROR
      ⎕L
      ^
```

⎕L has no value because the function interval has no left argument.

***Clearing the Entire State Indicator:*** If a number is not specified with the command, )SIC clears all suspended and pendent statements and editing sessions from the state indicator. For example:

```
      )SI
GN[1]
FN[2]
*
*
      )SIC
      )SI
```

This is equivalent to entering → (escape) until the state indicator is clear.

Because they are effectively local to functions in lines of immediate execution, )SIC without a number returns the system variables ⎕EM and ⎕ET to their intitial values in a clear workspace and removes the values of ⎕L and ⎕R. )SIC also purges and contracts the internal symbol table.

)SIC is identical to )RESET (see “)RESET—Clear the State Indicator” on page 449) and was added to meet international APL standards.

## )*SINL*—Display the State Indicator with Name List

)*SINL* [*number*]

The state indicator, discussed in “State Indicator” on page 355, is a list of:

- The calling sequence of defined functions and defined operators (and their pertinent line numbers).
- Asterisk(s) for all immediate execution expressions that did not complete, either because of an error in the expression or because the function invoked by the expression is pendent or suspended.

The command )*SINL* without a number specified displays data from each line of the state indicator. If a number is provided, the command does not display more than that number of lines of the state indicator.

Like )*SI*, )*SINL* lists the defined functions and defined operators (and their pertinent line numbers) in the state indicator, and an asterisk for all immediate execution expressions that did not complete. In addition, it lists the names local to the function or operator. For example:

```
      )SINL
GN[1]    Z
FN[2]    Z
*
*
```

If a definition line appears in the state indicator, the value within brackets indicates the status of the object:

| | |
|---|---|
| [Positive integer *I*] | Execution is suspended at line *I*.<br>Execution can be resumed by →⍳0. |
| [Negative integer *I*] | Execution is suspended at line \|*I*.<br>Execution can be restarted by →⎕*LC* or →*n*, where *n* is a line number. |
| [*blank*] | Execution is suspended by a line, but which one cannot be determined. Execution can be neither restarted nor resumed. |
| [ ]∇ | The object is being edited. |

***Clearing the State Indicator:*** )*SIC*, page 454, and )*RESET*, page 449, both clear the state indicator as does → (escape), described under “Clearing the State Indicator” on page 357.

## )SIS—Display the State Indicator with Statements

```
)SIS [number]
```

The state indicator, discussed on page 355, is a list of:

- The calling sequence of defined functions and defined operators (and their pertinent line numbers).
- Asterisk(s) for all immediate execution expressions that did not complete, either because of an error in the expression or because the function invoked by the expression is pendent or suspended.

The command )SIS without a number specified, will display data from each line of the state indicator. If a number is provided, the command will not display more than that number of lines of the state indicator.

The )SIS command displays each line in the state indicator and the statement that was being executed at the time the line was added to the state indicator. Carets shown on the line below the statement indicate how much of the statement had been executed.

```
      )SIS
GN[1] Z←3÷0
       ^ ^
FN[2] Z←GN×2
        ^
*  FN
   ^
*  3⍳
   ^^
```

First entry in the state indicator is last expression that did not complete.

If a definition line appears in the state indicator, the value within brackets indicates the status of the object:

| | |
|---|---|
| [Positive integer *I*] | Execution is suspended at line *I*. Execution can be resumed by →⍳0. |
| [Negative integer *I*] | Execution is suspended at line \|*I*. Execution can be restarted by →⎕LC or →*n*, where *n* is a line number. |
| [*blank*] | Execution is suspended by a line, but which one cannot be determined. Execution can be neither restarted nor resumed. |
| []∇ | The object is being edited. |

***Clearing the State Indicator:*** )SIC, page 454, and )RESET, page 449, both clear the state indicator as does → (escape), described under "Clearing the State Indicator" on page 357.

## )*SYMBOLS*—Query or Modify the Symbol Table Size

```
)SYMBOLS [number]
```

)*SYMBOLS* refers to the number of symbols in the APL2 *symbol table*.

The symbol table contains the names used in a workspace. When a name is first specified or defined, an entry is made for it in the symbol table.

If a number is not specified with the command, then )*SYMBOLS* purges unassigned names, compresses the internal symbol table, and reports the number of symbols currently in use. This is larger than the number of names of variables, functions, and operators in use. For example:

```
      )CLEAR
CLEAR WS

      )SYMBOLS
IS 47

      A←B←C←D←E←1
      )SYMBOLS
IS 52
```

If *number* is specified with the command, then )*SYMBOLS* expands or compresses the internal symbol table to at least the given number of slots. For example:

```
      ⎕WA
412708
      )SYMBOLS 100
      ⎕WA
412084
```

The symbol table is automatically expandable; system efficiency may be improved by enlarging the symbol table. A larger symbol table consumes more workspace but may save computation time. Some workspace may be reclaimed by compressing the symbol table.

**Note:** System functions and system variables exist in a clear workspace.

## )*VARS*—List Indicated Objects in the Active Workspace

```
)VARS [first] [-] [last]
```

)*VARS* displays an alphabetic list of the global variables in the active workspace.

See the introduction to this chapter, page 416, for explanations of the parameters first and last. The following examples illustrate the commands used to display partial contents of the sample workspace shown in Figure 58 on page 417.

```
      )VARS D-
DCS     GPAPL2  GPDESC  TIMER

      )VARS -G
CHANGE_ACTIVITY COIBM   DCS     GPAPL2  GPDESC

      )VARS D-G
DCS     GPAPL2  GPDESC

      )VARS G-D

      )VARS GP-GP
GPAPL2  GPDESC
```

## )WSID—Query or Assign the Active Workspace Identifier

)WSID [[*library*] *workspace*]

To learn the current identifier of the active workspace (called *wsid*), enter `)WSID`.

```
      )WSID
IS CLEAR WS                          Indicates that no identifier is
                                     associated with the workspace
⋮
      )WSID
IS LANGMAN
⋮
      )WSID
IS 1 DISPLAY
```

To change the current identification of the active workspace, enter the workspace name and, optionally, a library number:

```
      )WSID NEWNAME
WAS LANGMAN

      )WSID 1008 ANOTHER
WAS NEWNAME

      )WSID
IS 1008 ANOTHER
```

### Parameters

The introduction to this chapter (page 416) gives the general requirements for the library and workspace parameters.

# Chapter 11. Interpreter Messages

This chapter lists and explains interpreter messages in alphabetical order. If a message is associated with a specific ⎕*ET* setting, that setting is shown to the right of the message.

APL2 displays interpreter messages as the next line of output, beginning at the left margin. Such messages indicate:

- An interrupt signaled or an error within an expression. This could be an incorrect number of arguments for a function, invalid arguments, or incorrect syntax.
- Successful or unsuccessful completion of actions initiated by system commands.

In some cases messages are displayed with “+” as their final character. This means that additional, more detailed, information is available. That can be obtained by entering `)MORE` at the first opportunity (see “`)MORE`—List Additional Diagnostic Information” on page 438).

Messages for the workstations, including those for the APL2 session manager and auxiliary processors, are explained in the appropriate user's guide. All messages for APL2/370, including those concerning the APL2 session manager and auxiliary processors, are detailed in *APL2/370 Messages and Codes*.

## Interrupts and Errors in APL2 Expressions

Interrupts and errors in expressions generate the following types of messages:

- Classification
- The expression was interrupted or is in error
- Two carets pointing to the expression

Figure 66 shows the message displayed when an error occurs in the expression `2÷'X'`.

```
      2÷'X'         Expression as entered
DOMAIN ERROR        Classification
      2÷'X'         Expression in error
      ^^            Carets pointing to the expression
```

*Figure 66. Display When an Error Occurs within an APL2 Expression*

The left caret indicates how far APL2 interpreted the statement—from right to left. The right caret indicates where the error or interrupt occurred. In Figure 66, APL2 interpreted the entire expression. The error occurred with the divide function because the right argument was not numeric. Sometimes one caret appears because the point where the error occurred and the point at which APL2 interpreted the expression are the same.

The error message can be retrieved using ⎕*EM* (event message), page 281. Further information on the category of error can be obtained using either ⎕*ET* (event type), see page 287, or `)MORE`, see page 438.“Errors and Interrupts in

Immediate Execution” on page 59 discusses clearing the error from the state indicator.

## Interrupts and Errors in Defined Functions or Operators

When execution of a defined operation results in an error, APL2 displays an error message similar to that generated by an error in immediate execution. The name of the operation and the line number precede the display of the statement in error. Figure 67 shows a message displayed when an error occurs within the defined function named *AVERAGE*.

```
      AVERAGE 4 9 'B'    Function invoked
DOMAIN ERROR             Error classification
AVERAGE[1]  Z←(+/X)÷ρ∊X  Statement causing the error
              ^ ^        Pointers showing how far APL2
                         interpreted the statement and
                         where the error occurred
```

*Figure 67. Display When an Error Occurs within a Defined Function or Operator*

“Clearing the State Indicator” on page 357 describes clearing the state indicator when an error suspends execution of a defined operation.

## Errors in System Commands

Messages generated as a result of system commands may indicate successful completion of an operation or an error. For instance, issuing the `)CLEAR` system command to clear the active workspace results in the display of the information message `CLEAR WS`:

```
      )CLEAR
CLEAR WS
```

The message indicates successful clearing of the active workspace. After receiving an information message, you can proceed as normal.

If the message is caused by an error in the execution of a system command, the command is not executed. If the message ends with “+”, additional information is available. The additional information can be obtained by using the system command `)MORE`.

## Messages

**Note:** For descriptions of workstation messages, see the appropriate user's guide. For complete descriptions of APL2/370 messages, standard IBM message numbers, and corrective actions, see *APL2/370 Messages and Codes*.

```
AXIS ERROR                           ⎕ET ←→ 5 6
```

The indicated axis is incompatible with the function or operator and the given arguments; or the operator is not defined with an axis; or the axis specification includes semicolons.

### *CLEAR WS*

The current active workspace was replaced with a clear workspace. See “*)CLEAR*,” page 420, for a description of the initial contents of a clear workspace.

### *DEFN ERROR*

The ∇ or an editing command was misused:

- A syntactically incorrect ∇ or ⍫ command was entered to begin edit mode.
- An invalid character was used outside of a quote string or comment.
- The object cannot be edited. For example, a variable under the line editor or a locked function.
- An invalid edit command was entered.
- The closing ∇ or ⍫ was entered to establish an invalid object.
- Under Editor 1 (the line editor), a ∇ or ⍫ was entered on an unnumbered line to close a definition.
- Under Editor 2 (the full-screen editor), an attempt to pass lines from one segment to another failed because two lines numbered `[0]` appear in the same segment.
- An attempt was made to name an object with a name already in use in the active workspace.

Chapter 9, “The APL2 Editors” on page 375 discusses the use of the editors and explains all the edit commands.

### *DOMAIN ERROR* ⎕*ET* ←→ 5 4

The data type, degree of nesting, or number of arguments or operands specified for a primitive operation is invalid.

A *DOMAIN ERROR* is also generated if:

- A calculation requires or produced data that is beyond the range of the system implementation but does not fit any of the categories of *SYSTEM LIMIT* (this can occur with some mathematical functions).
- A nonresource error occurred in a defined function or operator whose fourth execution property is set to convert nonresource errors to a *DOMAIN ERROR*. (See ⎕*FX*, page 294.)
- A derived function from the slash operator or inner product was presented with an empty argument but no identity function existed for the function operand. Or a derived function from the Each operator or inner product was presented with an empty argument but no fill function existed for the function operand.

### *ENTRY ERROR*

The APL2 system received invalid characters. (Valid characters are listed in Appendix A, “The APL2 Character Set” on page 470.)

### *IMPROPER LIBRARY REFERENCE*

The library number specified for a )*COPY*, )*LOAD*, )*LIB*, or )*SAVE* command is incorrect or does not exist; or you are not authorized to access the library.

### *INCORRECT COMMAND*

The APL2 system command entered is invalid or has invalid parameters.

### *INDEX ERROR* ⎕*ET* ←→ 5 5

The index specified for bracket indexing (*R*[*I*]) or pick (*L*⊃*R*) is invalid with respect to the array given as the argument.

*INTERRUPT* ⎕*ET* ←→ 1 1

An interrupt was signaled from the terminal during processing and execution is halted. Execution can be resumed with →⍳0 or restarted by branching to a line number in the defined operation. If execution is not resumed or restarted, the state indicator should be cleared (with → or )*RESET*), as described in “Clearing the State Indicator” on page 357.

*LENGTH ERROR* ⎕*ET* ←→ 5 3

An argument of a primitive function or operand of a primitive operator has an axis whose length is incompatible with respect to that of the other argument or operand.

*LIBRARY I/O ERROR*

An internal error is preventing successful completion of a )*CONTINUE*, )*COPY*, )*DROP*, )*LOAD*, or )*SAVE* command.

*LIBRARY NOT AVAILABLE*

The )*CONTINUE*, )*COPY*, )*DROP*, )*LOAD*, )*SAVE* operation cannot be successfully completed because other user(s) have temporary control of a shared library; or you do not have write access to the library.

*NOT COPIED*: *object-names*

The listed objects were not copied by the )*PCOPY* system command because the objects already exist in the active workspace. Or the listed objects were not copied by the )*PCOPY* or )*IN* system command because the objects do not fit in the active workspace.

Also, the listed objects specified with the )*IN* system command do not have valid transfer forms in the file specified. Or the listed objects specified with the )*OUT* system command were not written to a transfer file because they do not exist in the active workspace or cannot be transferred.

*NOT ERASED*: *object-names*

The listed objects were not erased by the )*ERASE* command because the objects do not exist in the active workspace.

### *NOT FOUND*: [*object-names*]

The objects listed were either:

)*PCOPY* system command but cannot be found in the specified library workspace.

- Specified with an )*IN* system command but are not in the transfer file.

If no objects are listed, the file specified by name with the )*IN* system command cannot be found or is not a transfer file.

### *NOT SAVED, THIS WS IS* *wsid*

The )*SAVE* system command was issued in a *CLEAR WS* with no specified workspace name; or the workspace named in the )*SAVE* command exists in the library but is not the same as the name of the active workspace.

### *NOT SAVED, LIBRARY FULL*

The space allotted for saving workspaces is full; or the remaining space is not large enough to save the workspace.

### *RANK ERROR* ⎕*ET* ←→ 5 2

An array specified as the argument of a function or operand of an operator has a rank that is incompatible with another argument or operand. If the array is nested, the incompatibility may exist below the top level of structure.

### *SI WARNING*

A suspended or pendent defined function or operator was altered by editing or was replaced by the )*COPY* or )*PCOPY* command; or an attempt was made to use →⍳0 to resume execution of an operation that cannot be resumed (see “Suspension of Execution” on page 354).

### *SYNTAX ERROR* ⎕*ET* ←→ 2 *n*

The displayed APL2 expression is constructed improperly (for example, a function has a missing right argument); or an expression has mismatched parentheses or brackets.

If the error type is 2 5 (compatibility setting error), enter )*CS* 0 to return the compatibility setting to full APL2.

`SYSTEM ERROR` ⎕ET ←→ 1 2

A fault occurred in the internal operation of the APL2 system; or the active workspace was damaged.

On APL2/370, the damaged workspace is copied into a `DUMP`*nnnn* workspace. You may be able to copy objects from the `DUMP`*nnnn* workspace; however, examine and test them to ensure that they have not been damaged.

The active workspace is replaced by a CLEAR workspace.

See *APL2/370 Diagnosis Guide* for other information on recovering data.

`SYSTEM LIMIT` ⎕ET ←→ 1 *n*

The requested operation or action exceeds the system limits for symbol table size, number of shared variables, size of shared variable storage, rank of an array, number of dimensions of an array, number of items in an array, depth of an array, or size of a prompt in a prompt/response interaction.

`VALENCE ERROR` ⎕ET ←→ 5 1

An attempt has been made to specify a left argument for a monadic function, or to specify a single argument for a dyadic function, or to execute a function declared through the use of dyadic `⎕NA` whose definition cannot be activated.

`VALUE ERROR` ⎕ET ←→ 3 *n*

The constructed name being referenced was not specified; or an attempt was made to reference a value from a function that does not return a result; or a defined function that references two arguments was called with only a right argument, and the definition of the function does not check for this.

`WS CANNOT BE CONVERTED`

This message occurs after a `WS FULL` message if, because of system maintenance, the internal format of workspaces was changed and a larger workspace size is now needed.

`WS CONVERTED, RESAVE`

If system maintenance has changed the internal format of workspaces, the workspaces are automatically converted when you issue a `)LOAD` command. Saving the workspace ensures that the library copy of the workspace has the changed internal format. If you do not save the workspace, as directed, this message appears every time you load the workspace.

```
WS FULL                                        ⎕ET ←→ 1 3
```

An attempt was made to execute an operation that requires more storage than is currently available.

```
WS INVALID
```

The `)LOAD` system command was issued to load a file that is not an APL2 workspace.

```
WS LOCKED
```

The password specified with a `)COPY`, `)PCOPY`, or `)LOAD` command differs from that for the library workspace.

```
WS NOT FOUND
```

The workspace specified with a `)DROP` or `)LOAD` command does not exist.

```
⎕CT ERROR                                      ⎕ET ←→ 4 3
```

An attempt was made to execute a primitive function that uses `⎕CT` as an implicit argument when `⎕CT` has an inappropriate value or no value.

```
⎕FC ERROR                                      ⎕ET ←→ 4 4
```

An attempt was made to:

- Execute a primitive function that uses `⎕FC` as an implicit argument but `⎕FC` has an inappropriate value or no value.
- Display a negative number (with `L⍕R`) when `⎕FC[6]` has an inappropriate value or no value.

```
⎕IO ERROR                                      ⎕ET ←→ 4 2
```

An attempt was made to execute a primitive function that uses `⎕IO` as an implicit argument when `⎕IO` has an inappropriate value or no value.

```
⎕PP ERROR                                         ⎕ET ←→ 4 1
```

An attempt was made to display an array when ⎕*PP* has an inappropriate value or no value; or to execute a primitive function that uses ⎕*PP* as an implicit argument when ⎕*PP* has an inappropriate value or no value.

```
⎕PR ERROR                                         ⎕ET ←→ 4 7
```

An attempt was made to use the ⍞ system variable to create a character prompt immediately followed by a request for character input. However, ⎕*PR* has no value or an inappropriate one.

```
⎕RL ERROR                                         ⎕ET ←→ 4 5
```

An attempt was made to execute roll or deal, each of which requires ⎕*RL* as an implicit argument, but ⎕*RL* has an inappropriate value or no value.

# Appendix A. The APL2 Character Set

The APL2 character set is composed of 143 characters (plus blank) for which specified graphics must be used. The ⎕*AV* system variable defines these 144 code points plus an additional 112 deprecated or non-APL code points that may have no defined graphic, or whose graphics may vary.

Different APL implementations may choose differing ⎕*AV* orderings, and several have been used in the past. There are currently two orders being used by APL2 products:

- The ASCII order used on workstations and shown in Figure 68
- The EBCDIC order used on System/370 and shown in Figure 69

Both figures show a matrix corresponding to `16 16⍴⎕AV` and are labeled with hexadecimal indexes into the matrix. The hexadecimal representation *XX* of a character gives its row and column in the table. A corresponding index to ⎕*AV* can be obtained by the expression:

```
1+16⊥¯1+'0123456789ABCDEF'⍳XX
```

The following table shows the ASCII encoding of APL2 characters used on the workstation implementations.

*Figure 68. ASCII Character Set (Workstations)*

| | 0<br>0 | 0<br>1 | 0<br>2 | 0<br>3 | 0<br>4 | 0<br>5 | 0<br>6 | 0<br>7 | 0<br>8 | 0<br>9 | 0<br>*A* | 0<br>*B* | 0<br>*C* | 0<br>*D* | 0<br>*E* | 0<br>*F* | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | | | | | | | | | | | | | | | | | 00 |
| 10 | | | | | | | | | | | | | | | | | 10 |
| 20 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / | 20 |
| 30 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? | 30 |
| 40 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | 40 |
| 50 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ | 50 |
| 60 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | 60 |
| 70 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | | 70 |
| 80 | Ç | ü | é | â | ä | à | å | ç | ê | ë | è | ï | î | ì | Ä | Å | 80 |
| 90 | ⎕ | ⍞ | ⌹ | ô | ö | ò | û | ù | ⊤ | Ö | Ü | ø | £ | ⊥ | ₧ | ⌶ | 90 |
| *A*0 | á | í | ó | ú | ñ | Ñ | ª | º | ¿ | ⌈ | ¬ | | ∪ | ¡ | ⍕ | ⍎ | *A*0 |
| *B*0 | ░ | ▒ | ▓ | │ | ┤ | ⍟ | ∆ | ∇ | → | ╣ | ║ | ╗ | ╝ | ← | ⌊ | ┐ | *B*0 |
| *C*0 | └ | ┴ | ┬ | ├ | ─ | ┼ | ↑ | ↓ | ╚ | ╔ | ╩ | ╦ | ╠ | ═ | ╬ | ≡ | *C*0 |
| *D*0 | ⍸ | ⍷ | ∵ | ⌷ | ⍂ | ⌻ | ⊢ | ⊣ | ◊ | ┘ | ┌ | █ | ▄ | ▌ | ▐ | ▀ | *D*0 |
| *E*0 | α | β | ⊂ | ⊃ | ⍝ | ⍲ | ⍴ | ⍱ | ⌽ | ⊖ | ○ | ∨ | ⍳ | ⍉ | ∊ | ∩ | *E*0 |
| *F*0 | ⌿ | ⍀ | ≥ | ≤ | ≠ | × | ÷ | ⍙ | ∘ | ⍵ | ⍫ | ⍋ | ⍒ | ¯ | ¨ | | *F*0 |
| | 0<br>0 | 0<br>1 | 0<br>2 | 0<br>3 | 0<br>4 | 0<br>5 | 0<br>6 | 0<br>7 | 0<br>8 | 0<br>9 | 0<br>*A* | 0<br>*B* | 0<br>*C* | 0<br>*D* | 0<br>*E* | 0<br>*F* | |

The following table shows the EBCDIC encoding of APL2 characters used on the System/370 implementation.

*Figure 69. EBCDIC Character Set (APL2/370)*

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | | | | | | | | | | | | | | | | | 00 |
| 10 | | | | | | | | | | | | | | | | | 10 |
| 20 | | | | | | | | | | | | | | | | | 20 |
| 30 | | | | | | | | | | | | | | | | | 30 |
| 40 | | A̲ | B̲ | C̲ | D̲ | E̲ | F̲ | G̲ | H̲ | I̲ | ¢ | . | < | ( | + | \| | 40 |
| 50 | & | J̲ | K̲ | L̲ | M̲ | N̲ | O̲ | P̲ | Q̲ | R̲ | ! | $ | * | ) | ; | ¬ | 50 |
| 60 | - | / | S̲ | T̲ | U̲ | V̲ | W̲ | X̲ | Y̲ | Z̲ | ¦ | , | % | _ | > | ? | 60 |
| 70 | ◊ | ^ | ¨ | ⌻ | ⍸ | ⍷ | ⊢ | ⊣ | ∨ | ` | : | # | @ | ' | = | " | 70 |
| 80 | ~ | a | b | c | d | e | f | g | h | i | ↑ | ↓ | ≤ | ⌈ | ⌊ | → | 80 |
| 90 | ⎕ | j | k | l | m | n | o | p | q | r | ⊃ | ⊂ | | ○ | | ← | 90 |
| A0 | ¯ | ~ | s | t | u | v | w | x | y | z | ∩ | ∪ | ⊥ | [ | ≥ | ∘ | A0 |
| B0 | α | ∊ | ⍳ | ⍴ | ⍵ | | × | \ | ÷ | | ∇ | ∆ | ⊤ | ] | ≠ | \| | B0 |
| C0 | { | A | B | C | D | E | F | G | H | I | ⍲ | ⍱ | ⌷ | ⌽ | ⍂ | ⍉ | C0 |
| D0 | } | J | K | L | M | N | O | P | Q | R | ⌶ | ! | ⍒ | ⍋ | ⍞ | ⍝ | D0 |
| E0 | \ | ≡ | S | T | U | V | W | X | Y | Z | ⌿ | ⍀ | ∵ | ⊖ | ⌹ | ⍕ | E0 |
| F0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | ⍫ | ⍙ | ⍟ | ⍎ | | F0 |
| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | |

**Note:** The characters that are not shown in the above figures may or may not have graphic representations on specific APL2 input and output devices. All characters in ⎕AV and those obtained using ⎕AF can be used in comments and character constants even though they may not have graphic representations. Except for comments and character constants, only APL2 graphic characters whose use is defined in this manual can be meaningfully used in APL2 expressions.

## APL2 Special Characters

Figure 70 shows the APL2 special characters and their names. The names of the characters do not necessarily indicate the operations they represent. The table includes the pages containing descriptions for the APL2 use of the symbols.

*Figure 70 (Page 1 of 3). Names of APL2 Characters*

| Symbol | Symbol Name | Monadic Use (Operation Name and Page) | Dyadic Use (Operation Name and Page) | Other Reference (Page) |
|---|---|---|---|---|
| ¨ | dieresis | Each (109, 107) | — | — |

*Figure 70 (Page 1 of 3). Names of APL2 Characters*

| Symbol | Symbol Name | Monadic Use (Operation Name and Page) | Dyadic Use (Operation Name and Page) | Other Reference (Page) |
|---|---|---|---|---|
| ¯ | overbar | — | — | 11 |
| < | less | — | Less Than (219) | — |
| ≤ | not greater | — | Less Than or Equal (219) | — |
| = | equal | — | Equal (219) | — |
| ≥ | not less | — | Greater Than or Equal (219) | — |
| > | greater | — | Greater Than (219) | — |
| ≠ | not equal | — | Not Equal (219) | — |
| ∨ | down caret | — | Or (68) | — |
| ∧ | up caret | — | And (68) | — |
| - | bar | Negative (185) | Subtract (243) | — |
| ÷ | divide | Reciprocal (208) | Divide (100) | — |
| + | plus | Conjugate (88) | Add (65) | — |
| × | times | Direction (93) | Multiply (183) | — |
| ? | query | Roll (231) | Deal (89) | — |
| ⍵ | omega | — | — | — |
| ∊ | epsilon | Enlist (118) | Member (181) | — |
| ⍴ | rho | Shape (241) | Reshape (225) | — |
| ∼ | tilde | Not (68) | Without (258) | — |
| ↑ | up arrow | First (131) | Take (244), Take with Axis (247) | — |
| ↓ | down arrow | — | Drop (101), Drop with Axis (105) | — |
| ⍳ | iota | Interval (168) | Index Of (162) | — |
| ○ | circle | Pi Times (194) | Circle (80) | — |
| ⋆ | star | Exponential (127) | Power (201) | — |
| → | right arrow | — | — | 349 |
| ← | left arrow | — | — | 27, 39 |
| ⍺ | alpha | — | — | — |
| ⌈ | up stile | Ceiling (79) | Maximum (180) | — |
| ⌊ | down stile | Floor (133) | Minimum (182) | — |
| _ | underbar | — | — | 25, 27, 444 |
| ∇ | del | — | — | 391 |
| ∆ | delta | — | — | 25 |
| ∘ | jot | — | Outer Product (186) | — |
| ' | quote | — | — | 13 |
| ⎕ | quad | — | — | 262 |
| ( | left paren | — | — | 27, 36 |
| ) | right paren | — | — | 27, 36 |
| [ | left bracket | — | Bracket Index (70) | 27, 34 |
| ] | right bracket | — | Bracket Index (70) | 27, 34 |
| ⊂ | left shoe | Enclose (111), Enclose with Axis (113) | Partition (188), Partition with Axis (192) | — |
| ⊃ | right shoe | Disclose (94), Disclose with Axis (96) | Pick (195) | — |
| ∩ | up shoe | — | — | — |
| ∪ | down shoe | — | — | — |
| ⊥ | down tack | — | Decode (90) | — |
| ⊤ | up tack | — | Encode (116) | — |
| \| | stile | Magnitude (172) | Residue (227) | — |
| ; | semicolon | — | — | 27, 347 |

*Figure 70 (Page 2 of 3). Names of APL2 Characters*

| Symbol | Symbol Name | Monadic Use (Operation Name and Page) | Dyadic Use (Operation Name and Page) | Other Reference (Page) |
|---|---|---|---|---|
| : | colon | — | — | 27 |
| , | comma | Ravel (202), Ravel with Axis (204) | Catenate (74), Catenate with Axis (77), Laminate (169) | 291 |
| . | dot | — | Outer product (186), Inner product (165) | 10, 291 |
| \ ⍀ | slope<br>slope bar | Backslash (Expand (122, 124), Scan (239, 240)) | — | — |
| / ⌿ | slash<br>slash bar | Slash (Reduce (209, 217), N-wise Reduce (213, 215), Replicate (220, 222)) | — | — |
| ⍱ | down caret tilde | — | Nor (68) | — |
| ⍲ | up caret tilde | — | Nand (68) | — |
| ⍒ | del stile | Grade Down (147) | Grade Down (with Collating Sequence) (149) | — |
| ⍋ | delta stile | Grade Up (153) | Grade Up (with Collating Sequence) (155) | — |
| ⌽ ⊖ | circle stile<br>circle bar | Reverse (228), Reverse with Axis (229) | Rotate (232), Rotate with Axis (235) | — |
| ⍉ | circle slope | Transpose (with Reversed Axis) (256) | Transpose (General) (251) | — |
| ⊖ | circle bar | See circle stile | See circle stile | — |
| ⍟ | circle star | Natural Log (184) | Logarithm (171) | — |
| ⌶ | I-beam | — | — | — |
| ⍫ | del tilde | — | — | 391 |
| ⍎ | down tack jot | Execute (120) | — | — |
| ⍕ | up tack jot | Format (Default) (135) | Format By Specification (143), Format By Example (139) | — |
| ⍝ | up shoe jot | — | — | 27 |
| ⍞ | quad quote | — | — | 265 |
| ! | quote dot | Factorial (128) | Binomial (66) | — |
| ⌹ | quad divide | Matrix Inverse (177) | Matrix Divide (174) | — |
| ⍙ | delta underbar | — | — | 25 |
| ⍂ | quad slope | — | — | — |
| ⌻ | quad jot | — | — | — |
| ⌷ | squad | — | Index (160), Index with Axis (163) | — |
| ∵ | dieresis dot | — | — | — |
| ≡ | equal underbar | Depth (91) | Match (173) | — |
| ⍷ | epsilon underbar | — | Find (129) | — |
| ⍸ | iota underbar | — | — | — |

*Figure 70 (Page 3 of 3). Names of APL2 Characters*

| Symbol | Symbol Name | Monadic Use (Operation Name and Page) | Dyadic Use (Operation Name and Page) | Other Reference (Page) |
|---|---|---|---|---|
| ◊ | diamond | — | — | 36 |
| ⊢ | left tack | — | — | — |
| ⊣ | right tack | — | — | — |

Figure 71 maps ⎕AV into characters defined by the ISO 10646 standard 32-bit code and the Unicode subset of these characters.

*Figure 71 (Page 1 of 5). ASCII, EBCDIC, Unicode, and Symbol Equivalents*

| ASCII | | EBCDIC | | Unicode | | APL Char | Description |
|---|---|---|---|---|---|---|---|
| Decimal | Hex | Decimal | Hex | Decimal | Hex | | |
| 0 | 00 | 0 | 00 | 0 | 0000 | | Null |
| 1 | 01 | 1 | 01 | 1 | 0001 | | Start of heading |
| 2 | 02 | 2 | 02 | 2 | 0002 | | Start of text |
| 3 | 03 | 3 | 03 | 3 | 0003 | | End of text |
| 4 | 04 | 55 | 37 | 4 | 0004 | | End of transmission |
| 5 | 05 | 4 | 04 | 5 | 0005 | | Enquiry |
| 6 | 06 | 46 | 2E | 6 | 0006 | | Acknowledge |
| 7 | 07 | 47 | 2F | 7 | 0007 | | Bell |
| 8 | 08 | 22 | 16 | 8 | 0008 | | Backspace |
| 9 | 09 | 5 | 05 | 9 | 0009 | | Horizontal tabulation |
| 10 | 0A | 37 | 25 | 10 | 000A | | Linefeed |
| 11 | 0B | 11 | 0B | 11 | 000B | | Vertical tabulation |
| 12 | 0C | 12 | 0C | 12 | 000C | | Formfeed |
| 13 | 0D | 21 | 15 | 13 | 000D | | Carriage return |
| 14 | 0E | 14 | 0E | 14 | 000E | | Shift out |
| 15 | 0F | 15 | 0F | 15 | 000F | | Shift in |
| 16 | 10 | 16 | 10 | 16 | 0010 | | Data link escape |
| 17 | 11 | 17 | 11 | 17 | 0011 | | Device control one |
| 18 | 12 | 18 | 12 | 18 | 0012 | | Device control two |
| 19 | 13 | 19 | 13 | 19 | 0013 | | Device control three |
| 20 | 14 | 60 | 3C | 20 | 0014 | | Device control four |
| 21 | 15 | 6 | 06 | 21 | 0015 | | Negative acknowledgement |
| 22 | 16 | 50 | 32 | 22 | 0016 | | Synchronous idle |
| 23 | 17 | 38 | 26 | 23 | 0017 | | End of transmission block |
| 24 | 18 | 7 | 07 | 24 | 0018 | | Cancel |
| 25 | 19 | 9 | 09 | 25 | 0019 | | End of medium |
| 26 | 1A | 20 | 14 | 26 | 001A | | Substitute |
| 27 | 1B | 39 | 27 | 27 | 001B | | Escape |
| 28 | 1C | 34 | 22 | 28 | 001C | | File separator |
| 29 | 1D | 29 | 1D | 29 | 001D | | Group separator |
| 30 | 1E | 53 | 35 | 30 | 001E | | Record separator |
| 31 | 1F | 13 | 0D | 31 | 001F | | Unit separator |
| 32 | 20 | 64 | 40 | 32 | 0020 | | Space |
| 33 | 21 | 219 | DB | 33 | 0021 | ! | Exclamation mark |
| 34 | 22 | 127 | 7F | 34 | 0022 | | Quotation mark |
| 35 | 23 | 123 | 7B | 35 | 0023 | | Number sign |
| 36 | 24 | 91 | 5B | 36 | 0024 | | Dollar sign |
| 37 | 25 | 108 | 6C | 37 | 0025 | | Percent sign |
| 38 | 26 | 80 | 50 | 38 | 0026 | | Ampersand |
| 39 | 27 | 125 | 7D | 39 | 0027 | ' | Apostrophe, quote |
| 40 | 28 | 77 | 4D | 40 | 0028 | ( | Opening parenthesis |
| 41 | 29 | 93 | 5D | 41 | 0029 | ) | Closing parenthesis |
| 42 | 2A | 92 | 5C | 42 | 002A | * | Star |
| 43 | 2B | 78 | 4E | 43 | 002B | + | Plus sign |
| 44 | 2C | 107 | 6B | 44 | 002C | , | Comma |
| 45 | 2D | 96 | 60 | 45 | 002D | - | Bar |
| 46 | 2E | 75 | 4B | 46 | 002E | . | Dot |
| 47 | 2F | 97 | 61 | 47 | 002F | / | Slash |
| 48 | 30 | 240 | F0 | 48 | 0030 | 0 | Digit zero |
| 49 | 31 | 241 | F1 | 49 | 0031 | 1 | Digit one |
| 50 | 32 | 242 | F2 | 50 | 0032 | 2 | Digit two |

*Figure 71 (Page 2 of 5). ASCII, EBCDIC, Unicode, and Symbol Equivalents*

| ASCII | | EBCDIC | | Unicode | | APL Char | Description |
|---|---|---|---|---|---|---|---|
| **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | | |
| 51 | 33 | 243 | F3 | 51 | 0033 | 3 | Digit three |
| 52 | 34 | 244 | F4 | 52 | 0034 | 4 | Digit four |
| 53 | 35 | 245 | F5 | 53 | 0035 | 5 | Digit five |
| 54 | 36 | 246 | F6 | 54 | 0036 | 6 | Digit six |
| 55 | 37 | 247 | F7 | 55 | 0037 | 7 | Digit seven |
| 56 | 38 | 248 | F8 | 56 | 0038 | 8 | Digit eight |
| 57 | 39 | 249 | F9 | 57 | 0039 | 9 | Digit nine |
| 58 | 3A | 122 | 7A | 58 | 003A | : | Colon |
| 59 | 3B | 94 | 5E | 59 | 003B | ; | Semicolon |
| 60 | 3C | 76 | 4C | 60 | 003C | < | Less-than sign |
| 61 | 3D | 126 | 7E | 61 | 003D | = | Equals sign |
| 62 | 3E | 110 | 6E | 62 | 003E | > | Greater-than sign |
| 63 | 3F | 111 | 6F | 63 | 003F | ? | Query |
| 64 | 40 | 124 | 7C | 64 | 0040 | | Commercial at |
| 65 | 41 | 193 | C1 | 65 | 0041 | A | Capital A |
| 66 | 42 | 194 | C2 | 66 | 0042 | B | Capital B |
| 67 | 43 | 195 | C3 | 67 | 0043 | C | Capital C |
| 68 | 44 | 196 | C4 | 68 | 0044 | D | Capital D |
| 69 | 45 | 197 | C5 | 69 | 0045 | E | Capital E |
| 70 | 46 | 198 | C6 | 70 | 0046 | F | Capital F |
| 71 | 47 | 199 | C7 | 71 | 0047 | G | Capital G |
| 72 | 48 | 200 | C8 | 72 | 0048 | H | Capital H |
| 73 | 49 | 201 | C9 | 73 | 0049 | I | Capital I |
| 74 | 4A | 209 | D1 | 74 | 004A | J | Capital J |
| 75 | 4B | 210 | D2 | 75 | 004B | K | Capital K |
| 76 | 4C | 211 | D3 | 76 | 004C | L | Capital L |
| 77 | 4D | 212 | D4 | 77 | 004D | M | Capital M |
| 78 | 4E | 213 | D5 | 78 | 004E | N | Capital N |
| 79 | 4F | 214 | D6 | 79 | 004F | O | Capital O |
| 80 | 50 | 215 | D7 | 80 | 0050 | P | Capital P |
| 81 | 51 | 216 | D8 | 81 | 0051 | Q | Capital Q |
| 82 | 52 | 217 | D9 | 82 | 0052 | R | Capital R |
| 83 | 53 | 226 | E2 | 83 | 0053 | S | Capital S |
| 84 | 54 | 227 | E3 | 84 | 0054 | T | Capital T |
| 85 | 55 | 228 | E4 | 85 | 0055 | U | Capital U |
| 86 | 56 | 229 | E5 | 86 | 0056 | V | Capital V |
| 87 | 57 | 230 | E6 | 87 | 0057 | W | Capital W |
| 88 | 58 | 231 | E7 | 88 | 0058 | X | Capital X |
| 89 | 59 | 232 | E8 | 89 | 0059 | Y | Capital Y |
| 90 | 5A | 233 | E9 | 90 | 005A | Z | Capital Z |
| 91 | 5B | 173 | AD | 91 | 005B | [ | Left bracket |
| 92 | 5C | 183 | B7 | 92 | 005C | \ | Slope |
| 93 | 5D | 189 | BD | 93 | 005D | ] | Right bracket |
| 94 | 5E | 113 | 71 | 94 | 005E | ^ | Up caret |
| 95 | 5F | 109 | 6D | 95 | 005F | _ | Underbar |
| 96 | 60 | 121 | 79 | 96 | 0060 | | Grave accent |
| 97 | 61 | 129 | 81 | 97 | 0061 | a | Small a |
| 98 | 62 | 130 | 82 | 98 | 0062 | b | Small b |
| 99 | 63 | 131 | 99 | 83 | 0063 | c | Small c |
| 100 | 64 | 132 | 84 | 100 | 0064 | d | Small d |
| 101 | 65 | 133 | 85 | 101 | 0065 | e | Small e |
| 102 | 66 | 134 | 86 | 102 | 0066 | f | Small f |
| 103 | 67 | 135 | 87 | 103 | 0067 | g | Small g |
| 104 | 68 | 136 | 88 | 104 | 0068 | h | Small h |

*Figure 71 (Page 3 of 5). ASCII, EBCDIC, Unicode, and Symbol Equivalents*

| ASCII | | EBCDIC | | Unicode | | APL Char | Description |
|---|---|---|---|---|---|---|---|
| **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | | |
| 105 | 69 | 137 | 89 | 105 | 0069 | i | Small i |
| 106 | 6A | 145 | 91 | 106 | 006A | j | Small j |
| 107 | 6B | 146 | 92 | 107 | 006B | k | Small k |
| 108 | 6C | 147 | 93 | 108 | 006C | l | Small l |
| 109 | 6D | 148 | 94 | 109 | 006D | m | Small m |
| 110 | 6E | 149 | 95 | 110 | 006E | n | Small n |
| 111 | 6F | 150 | 96 | 111 | 006F | o | Small o |
| 112 | 70 | 151 | 97 | 112 | 0070 | p | Small p |
| 113 | 71 | 152 | 98 | 113 | 0071 | q | Small q |
| 114 | 72 | 153 | 99 | 114 | 0072 | r | Small r |
| 115 | 73 | 162 | A2 | 115 | 0073 | s | Small s |
| 116 | 74 | 163 | A3 | 116 | 0074 | t | Small t |
| 117 | 75 | 164 | A4 | 117 | 0075 | u | Small u |
| 118 | 76 | 165 | A5 | 118 | 0076 | v | Small v |
| 119 | 77 | 166 | A6 | 119 | 0077 | w | Small w |
| 120 | 78 | 167 | A7 | 120 | 0078 | x | Small x |
| 121 | 79 | 168 | A8 | 121 | 0079 | y | Small y |
| 122 | 7A | 169 | A9 | 122 | 007A | z | Small z |
| 123 | 7B | 192 | C0 | 123 | 007B | | Left curly brace |
| 124 | 7C | 191 | BF | 124 | 007C | \| | Stile |
| 125 | 7D | 208 | D0 | 125 | 007D | | Right curly brace |
| 126 | 7E | 128 | 80 | 126 | 007E | ~ | Tilde |
| 127 | 7F | 65 | 41 | 127 | 007F | | Delete |
| 128 | 80 | 66 | 42 | 199 | 00C7 | | Capital C cedilla |
| 129 | 81 | 67 | 43 | 252 | 00FC | | Capital U dieresis |
| 130 | 82 | 68 | 44 | 233 | 00E9 | | Small e acute |
| 131 | 83 | 69 | 45 | 226 | 00E2 | | Small a circumflex |
| 132 | 84 | 70 | 46 | 228 | 00E4 | | Small a dieresis |
| 133 | 85 | 71 | 47 | 224 | 00E0 | | Small a grave |
| 134 | 86 | 72 | 48 | 229 | 00E5 | | Small a ring |
| 135 | 87 | 73 | 49 | 231 | 00E7 | | Small c cedilla |
| 136 | 88 | 81 | 51 | 234 | 00EA | | Small e circumflex |
| 137 | 89 | 82 | 52 | 235 | 00EB | | Small e dieresis |
| 138 | 8A | 83 | 53 | 232 | 00E8 | | Small e grave |
| 139 | 8B | 84 | 54 | 239 | 00EF | | Small i dieresis |
| 140 | 8C | 85 | 55 | 238 | 00EE | | Small i circumflex |
| 141 | 8D | 86 | 56 | 236 | 00EC | | Small i grave |
| 142 | 8E | 87 | 57 | 196 | 00C4 | | Capital A dieresis |
| 143 | 8F | 88 | 58 | 197 | 00C5 | | Capital A ring |
| 144 | 90 | 144 | 90 | 9647 | 25AF | ▯ | Quad |
| 145 | 91 | 222 | DE | 9054 | 235E | ⍞ | Quote quad |
| 146 | 92 | 238 | EE | 9017 | 2339 | ⌹ | Quad divide |
| 147 | 93 | 89 | 59 | 244 | 00F4 | | Small o circumflex |
| 148 | 94 | 98 | 62 | 246 | 00F6 | | Small o dieresis |
| 149 | 95 | 99 | 63 | 242 | 00F2 | | Small o grave |
| 150 | 96 | 100 | 64 | 251 | 00FB | | Small u circumflex |
| 151 | 97 | 101 | 65 | 249 | 00F9 | | Small u grave |
| 152 | 98 | 188 | BC | 8868 | 22A4 | ⊤ | Up tack |
| 153 | 99 | 102 | 66 | 214 | 00D6 | | Capital O dieresis |
| 154 | 9A | 103 | 67 | 220 | 00DC | | Capital U dieresis |
| 155 | 9B | 74 | 4A | 248 | 00F8 | | Small o slash |
| 156 | 9C | 104 | 68 | 163 | 00A3 | | Pound sign |
| 157 | 9D | 172 | AC | 8869 | 22A5 | ⊥ | Down tack |
| 158 | 9E | 105 | 69 | 9078 | 2376 | | Alpha underbar |

*Figure 71 (Page 4 of 5). ASCII, EBCDIC, Unicode, and Symbol Equivalents*

| ASCII | | EBCDIC | | Unicode | | APL Char | Description |
|---|---|---|---|---|---|---|---|
| Decimal | Hex | Decimal | Hex | Decimal | Hex | | |
| 159 | 9F | 218 | DA | 9014 | 2336 | ⌶ | I-beam |
| 160 | A0 | 33 | 21 | 225 | 00E1 | | Small a acute |
| 161 | A1 | 161 | A1 | 237 | 00ED | | Small i acute |
| 162 | A2 | 35 | 23 | 243 | 00F3 | | Small o acute |
| 163 | A3 | 36 | 24 | 250 | 00FA | | Small u acute |
| 164 | A4 | 106 | 6A | 241 | 00F1 | | Small n tilde |
| 165 | A5 | 158 | 9E | 209 | 00D1 | | Capital N tilde |
| 166 | A6 | 224 | E0 | 170 | 00AA | | Feminine ordinal indicator |
| 167 | A7 | 181 | B5 | 186 | 00BA | | Masculine ordinal indicator |
| 168 | A8 | 41 | 29 | 191 | 00BF | | Inverted question mark |
| 169 | A9 | 141 | 8D | 8968 | 2308 | ⌈ | Up stile |
| 170 | AA | 95 | 5F | 172 | 00AC | | Not sign |
| 171 | AB | 54 | 36 | 189 | 00BD | | Fraction one half |
| 172 | AC | 171 | AB | 8746 | 222A | ∪ | Down shoe |
| 173 | AD | 43 | 2B | 161 | 00A1 | | Inverted exclamation mark |
| 174 | AE | 239 | EF | 9045 | 2355 | ⍕ | Up tack jot |
| 175 | AF | 254 | FE | 9038 | 234E | ⍎ | Down tack jot |
| 176 | B0 | 10 | 0A | 9617 | 2591 | | Light shade |
| 177 | B1 | 32 | 20 | 9618 | 2592 | | Medium shade |
| 178 | B2 | 42 | 2A | 9619 | 2593 | | Dark shade |
| 179 | B3 | 26 | 1A | 9474 | 2502 | | Forms light vertical |
| 180 | B4 | 63 | 3F | 9408 | 2524 | | Forms light vertical and left |
| 181 | B5 | 253 | FD | 9055 | 235F | ⍟ | Circle star |
| 182 | B6 | 187 | BB | 8710 | 2206 | ∆ | Delta |
| 183 | B7 | 186 | BA | 8711 | 2207 | ∇ | Del |
| 184 | B8 | 143 | 8F | 8594 | 2192 | → | Right arrow |
| 185 | B9 | 49 | 31 | 9571 | 2563 | | Forms double vertical and left |
| 186 | BA | 48 | 30 | 9553 | 2551 | | Forms double vertical |
| 187 | BB | 51 | 33 | 9559 | 2557 | | Forms double down and left |
| 188 | BC | 52 | 34 | 9565 | 255D | | Forms double up and left |
| 189 | BD | 159 | 9F | 8592 | 2190 | ← | Left arrow |
| 190 | BE | 142 | 8E | 8970 | 230A | ⌊ | Down stile |
| 191 | BF | 27 | 1B | 9488 | 2510 | | Forms light down and left |
| 192 | C0 | 30 | 1E | 9492 | 2514 | | Forms light up and right |
| 193 | C1 | 62 | 3E | 9524 | 2534 | | Forms light up and horizontal |
| 194 | C2 | 59 | 3B | 9516 | 252C | | Forms light down and horizontal |
| 195 | C3 | 61 | 3D | 9500 | 251C | | Forms light vertical and right |
| 196 | C4 | 45 | 2D | 9472 | 2500 | | Forms light horizontal |
| 197 | C5 | 44 | 2C | 9532 | 253C | | Forms light vertical and horizontal |
| 198 | C6 | 138 | 8A | 8593 | 2191 | ↑ | Up arrow |
| 199 | C7 | 139 | 8B | 8595 | 2193 | ↓ | Down arrow |
| 200 | C8 | 185 | B9 | 9562 | 255A | | Forms double up and right |
| 201 | C9 | 56 | 38 | 9556 | 2554 | | Forms double down and right |
| 202 | CA | 57 | 39 | 9577 | 2569 | | Forms double up and horizontal |
| 203 | CB | 79 | 4F | 9574 | 2566 | | Forms double down and horizontal |
| 204 | CC | 90 | 5A | 9568 | 2560 | | Forms double vertical and right |
| 205 | CD | 156 | 9C | 9552 | 2550 | | Forms double horizontal |
| 206 | CE | 58 | 3A | 9580 | 256C | | Forms double vertical and horizontal |
| 207 | CF | 225 | E1 | 8801 | 2261 | ≡ | Equal underbar |

*Figure 71 (Page 5 of 5). ASCII, EBCDIC, Unicode, and Symbol Equivalents*

| ASCII | | EBCDIC | | Unicode | | APL Char | Description |
|---|---|---|---|---|---|---|---|
| **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | | |
| 208 | D0 | 116 | 74 | 9080 | 2378 | ⍸ | Iota underbar |
| 209 | D1 | 117 | 75 | 9079 | 2377 | ⍷ | Epsilon underbar |
| 210 | D2 | 236 | EC | 8757 | 2235 | ∵ | Dotted del (dieresis dot) |
| 211 | D3 | 204 | CC | 9015 | 2337 | ⌷ | Squash quad |
| 212 | D4 | 206 | CE | 9026 | 2342 | ⍂ | Quad slope |
| 213 | D5 | 115 | 73 | 9019 | 233B | ⌻ | Quad jot |
| 214 | D6 | 118 | 76 | 8866 | 22A2 | ⊣ | Right tack |
| 215 | D7 | 119 | 77 | 8867 | 22A3 | ⊢ | Left tack |
| 216 | D8 | 112 | 70 | 9674 | 25CA | ◊ | Diamond |
| 217 | D9 | 31 | 1F | 9496 | 2518 | | Forms light up and left |
| 218 | DA | 28 | 1C | 9484 | 250C | | Forms light down and right |
| 219 | DB | 24 | 18 | 9608 | 2588 | | Full block |
| 220 | DC | 40 | 28 | 9604 | 2584 | | Lower half block |
| 221 | DD | 23 | 17 | 166 | 00A6 | | Broken vertical bar |
| 222 | DE | 25 | 19 | 204 | 00CC | | Capital I grave |
| 223 | DF | 8 | 08 | 9600 | 2580 | | Upper half block |
| 224 | E0 | 176 | B0 | 9082 | 237A | ⍺ | Alpha |
| 225 | E1 | 250 | FA | 9081 | 2379 | | Omega underbar |
| 226 | E2 | 155 | 9B | 8834 | 2282 | ⊂ | Left shoe |
| 227 | E3 | 154 | 9A | 8835 | 2283 | ⊃ | Right shoe |
| 228 | E4 | 223 | DF | 9053 | 235D | ⍝ | Up shoe jot |
| 229 | E5 | 202 | CA | 9074 | 2372 | ⍲ | Up caret tilde |
| 230 | E6 | 179 | B3 | 9076 | 2374 | ⍴ | Rho |
| 231 | E7 | 203 | CB | 9073 | 2371 | ⍱ | Down caret tilde |
| 232 | E8 | 205 | CD | 9021 | 233D | ⌽ | Circle stile |
| 233 | E9 | 237 | ED | 8854 | 2296 | ⊖ | Circle bar |
| 234 | EA | 157 | 9D | 9675 | 25CB | ○ | Circle |
| 235 | EB | 120 | 78 | 8744 | 2228 | ∨ | Down caret |
| 236 | EC | 178 | B2 | 9075 | 2373 | ⍳ | Iota |
| 237 | ED | 207 | CF | 9033 | 2349 | ⍉ | Circle slope |
| 238 | EE | 177 | B1 | 8714 | 220A | ∊ | Epsilon |
| 239 | EF | 170 | AA | 8745 | 2229 | ∩ | Up shoe |
| 240 | F0 | 234 | EA | 9023 | 233F | ⌿ | Slash bar |
| 241 | F1 | 235 | EB | 9024 | 2340 | ⍀ | Slope bar |
| 242 | F2 | 174 | AE | 8805 | 2265 | ≥ | Not-less-than sign |
| 243 | F3 | 140 | 8C | 8804 | 2264 | ≤ | Not-greater-than sign |
| 244 | F4 | 190 | BE | 8800 | 2260 | ≠ | Not-equal sign |
| 245 | F5 | 182 | B6 | 215 | 00D7 | × | Times |
| 246 | F6 | 184 | B8 | 247 | 00F7 | ÷ | Divide |
| 247 | F7 | 252 | FC | 9049 | 2359 | ⍙ | Delta underbar |
| 248 | F8 | 175 | AF | 8728 | 2218 | ∘ | Jot |
| 249 | F9 | 180 | B4 | 9077 | 2375 | ⍵ | Omega |
| 250 | FA | 251 | FB | 9067 | 236B | ⍫ | Del tilde |
| 251 | FB | 221 | DD | 9035 | 234B | ⍋ | Delta stile |
| 252 | FC | 220 | DC | 9042 | 2352 | ⍒ | Del stile |
| 253 | FD | 160 | A0 | 175 | 00AF | ¯ | Overbar |
| 254 | FE | 114 | 72 | 168 | 00A8 | ¨ | Dieresis |
| 255 | FF | 255 | FF | 160 | 00A0 | | Nonbreaking space |

## Explanation of Characters

The alphabetic characters are:

```
ABCDEFGHIJKLMNOPQUSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
∆⍙
```

The alphameric characters include the alphabetic characters, and also:

```
0123456789_¯
```

The blank is not visible in the EBCDIC and ASCII character set figures (Figure 69 on page 471 and Figure 68 on page 470), but is encoded in EBCDIC as `⎕AF 64` (`X'40'`) and in ASCII as `⎕AF 32` (`X'20'`).

The underbarred APL alphabet is now deprecated, having been replaced by lowercase letters, and is not defined at all in ASCII. When APL objects containing underbarred letters are transferred to an ASCII based system, the characters are mapped as follows:

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| hex | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| EBCDIC | A̲ | B̲ | C̲ | D̲ | E̲ | F̲ | G̲ | H̲ | I̲ | J̲ | K̲ | L̲ | M̲ | N̲ | O̲ | P̲ | Q̲ | R̲ | S̲ | T̲ | U̲ | V̲ | W̲ | X̲ | Y̲ | Z̲ |
| ASCII | ⌂ | Ç | ü | é | â | ä | à | å | ç | ê | ë | è | ï | î | ì | Ä | Å | ô | ö | ò | û | ù | Ö | Ü | £ | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
| hex | 7F | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 8A | 8B | 8C | 8D | 8E | 8F | 93 | 94 | 95 | 96 | 97 | 99 | 9A | 9C | 9E |

The ASCII character set figure (Figure 68 on page 470) shows a number of characters that are not displayed in the EBCDIC figure (Figure 69 on page 471). There is a one-for-one code point mapping of these characters, but either the EBCDIC code points are below `X'40'` so that they may not display or print on some devices, or the graphics for the EBCDIC code points are national language dependent. The following table shows these code points and the graphics defined by a few commonly used code pages:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ASCII | ¢ | $ | ¬ | ` | # | @ | { | } | ┘ | ┼ | ─ | ┬ | ├ | ┴ | ┤ | │ | ┐ | ┌ | └ |
| ASCII hex | 9B | 24 | AA | 60 | 23 | 40 | 7B | 7D | B3 | BF | DA | C0 | D9 | C5 | C4 | C2 | C3 | C1 | B4 |
| EBCDIC hex | 4A | 5B | 5F | 79 | 7B | 7C | C0 | D0 | 1A | 1B | 1C | 1E | 1F | 2C | 2D | 3B | 3D | 3E | 3F |
| | | | | | | | | | | | | | | | | | | | |
| Codepage 037 | ¢ | $ | ¬ | ` | # | @ | { | } | | | (Canada, US) | | | | | | | | |
| Codepage 273 | Ä | $ | ^ | ` | # | § | ä | ü | | | (Austria, Germany) | | | | | | | | |
| Codepage 275 | É | $ | ^ | ã | Õ | Ã | õ | é | | | (Brazil) | | | | | | | | |
| Codepage 277 | # | Å | ^ | ` | Æ | Ø | æ | å | | | (Denmark, Norway) | | | | | | | | |
| Codepage 278 | § | Å | ^ | é | Ä | Ö | ä | å | | | (Finland, Sweden) | | | | | | | | |
| Codepage 280 | ° | $ | ^ | ù | £ | § | à | è | | | (Italy) | | | | | | | | |
| Codepage 281 | £ | ¥ | ¬ | ` | # | @ | { | } | | | (Japan-Latin) | | | | | | | | |
| Codepage 282 | [ | $ | ^ | ` | Ã | Õ | ã | ´ | | | (Portugal) | | | | | | | | |
| Codepage 284 | [ | $ | ¬ | ` | Ñ | @ | { | } | | | (Latin America, Spain) | | | | | | | | |
| Codepage 285 | $ | £ | ¬ | ` | # | @ | { | } | | | (United Kingdom) | | | | | | | | |
| Codepage 290 | £ | ¥ | ¬ | ` | # | @ | { | } | | | (Katakana) | | | | | | | | |
| Codepage 297 | ° | $ | ^ | µ | £ | à | é | è | | | (France) | | | | | | | | |
| Codepage 500 | [ | $ | ^ | ` | # | @ | { | } | | | (International) | | | | | | | | |

The following characters are not shown in either the ASCII or EBCDIC character set figures (Figure 68 on page 470 and Figure 69 on page 471). They do have an extended ASCII assignments that are honored by APL-ASCII, though the code points may be redefined by other ASCII code pages. But the graphics for the EBCDIC mapping are national language dependent, as this table shows:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ASCII | í | ñ | Ñ | ª | º | ╚ | ╦ | ╠ | ═ | ß | |
| ASCII hex | A1 | A4 | A5 | A6 | A7 | C8 | CB | CC | CD | E1 | |
| EBCDIC hex | A1 | 6A | 9E | E0 | B5 | B9 | 4F | 5A | 9C | FA | |
| | | | | | | | | | | | |
| Codepage 037 | ˜ | ¦ | Æ | \ | § | ¾ | \| | ! | æ | Û | (Canada, US) |
| Codepage 273 | ß | ö | Æ | Ö | @ | ¾ | ! | Ü | æ | ³ | (Austria, Germany) |
| Codepage 275 | ˜ | ç | Æ | \ | § | ¾ | ! | $ | æ | ³ | (Brazil) |
| Codepage 277 | ü | ø | [ | \ | @ | ¾ | ! | ¤ | { | ³ | (Denmark, Norway) |
| Codepage 278 | ü | ö | Æ | É | [ | ¾ | ! | ¤ | æ | ³ | (Finland, Sweden) |
| Codepage 280 | ì | ò | Æ | ç | @ | ¾ | ! | é | æ | ³ | (Italy) |
| Codepage 281 | ¯ | ¦ | Æ | $ | § | ¾ | \| | ! | æ | ³ | (Japan-Latin) |
| Codepage 282 | ç | õ | Æ | Ç | § | ¾ | ! | ] | æ | ³ | (Portugal) |
| Codepage 284 | ¨ | ñ | Æ | \ | § | ¾ | \| | ] | æ | ³ | (Latin America, Spain) |
| Codepage 285 | ¯ | ¦ | Æ | \ | § | ¾ | \| | ! | æ | ³ | (United Kingdom) |
| Codepage 297 | ¨ | ù | Æ | ç | ] | ¾ | ! | § | æ | ³ | (France) |
| Codepage 500 | ˜ | ¦ | Æ | \ | § | ¾ | ! | ] | æ | ³ | (International) |

The character X'FF' (⎕AF 255) has no graphics associated with it and, if used in comments, may not be preserved by the editors. Similar problems may occur for any of the characters X'00' through X'3F' (64↑⎕AV) in EBCDIC, and the characters X'00' through X'1F' (32↑⎕AV) in ASCII, although in some cases graphics are associated with them. In particular, APL systems and their editors often recognize the following as control characters:

| **EBCDIC** | | **ASCII** | | **⎕TC index** | **Usage as a** |
|---|---|---|---|---|---|
| **⎕AF** | **hex** | **⎕AF** | **hex** | **(Origin 1)** | **Control Character** |
| 5 | 05 | 9 | 09 | | Tab |
| 14 | 0E | 14 | 0E | | Shift Out |
| 15 | 0F | 15 | 0F | | Shift In |
| 21 | 15 | 13 | 0D | 2 | Carriage Return |
| 22 | 16 | 8 | 08 | 1 | Backspace |
| 37 | 25 | 10 | 0A | 3 | Line Feed |

Figure 72 on page 482 and Figure 73 on page 482 show complete code point mapping tables from EBCDIC to ASCII and from ASCII to EBCDIC. Hexadecimal source code points are shown in the table margins, with hexadecimal destination code points in the body of the table.

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
    +-----------------------------------------------------+
00  |  00 01 02 03 05 09 15 18 DF 19 B0 0B 0C 1F 0E 0F  |  00
10  |  10 11 12 13 1A 0D 08 DD DB DE B3 BF DA 1D C0 D9  |  10
20  |  B1 A0 1C A2 A3 0A 17 1B DC A8 B2 AD C5 C4 06 07  |  20
30  |  BA B9 16 BB BC 1E AB 04 C9 CA CE C2 14 C3 C1 B4  |  30
40  |  20 7F 80 81 82 83 84 85 86 87 9B 2E 3C 28 2B CB  |  40
50  |  26 88 89 8A 8B 8C 8D 8E 8F 93 CC 24 2A 29 3B AA  |  50
60  |  2D 2F 94 95 96 97 99 9A 9C 9E A4 2C 25 5F 3E 3F  |  60
70  |  D8 5E FE D5 D0 D1 D6 D7 EB 60 3A 23 40 27 3D 22  |  70
80  |  7E 61 62 63 64 65 66 67 68 69 C6 C7 F3 A9 BE B8  |  80
90  |  90 6A 6B 6C 6D 6E 6F 70 71 72 E3 E2 CD EA A5 BD  |  90
A0  |  FD A1 73 74 75 76 77 78 79 7A EF AC 9D 5B F2 F8  |  A0
B0  |  E0 EE EC E6 F9 A7 F5 5C F6 C8 B7 B6 98 5D F4 7C  |  B0
C0  |  7B 41 42 43 44 45 46 47 48 49 E5 E7 D3 E8 D4 ED  |  C0
D0  |  7D 4A 4B 4C 4D 4E 4F 50 51 52 9F 21 FC FB 91 E4  |  D0
E0  |  A6 CF 53 54 55 56 57 58 59 5A F0 F1 D2 E9 92 AE  |  E0
F0  |  30 31 32 33 34 35 36 37 38 39 E1 FA F7 B5 AF FF  |  F0
    +-----------------------------------------------------+
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

*Figure 72. APL2 EBCDIC to ASCII code point mapping*

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
    +-----------------------------------------------------+
00  |  00 01 02 03 37 04 2E 2F 16 05 25 0B 0C 15 0E 0F  |  00
10  |  10 11 12 13 3C 06 32 26 07 09 14 27 22 1D 35 0D  |  10
20  |  40 DB 7F 7B 5B 6C 50 7D 4D 5D 5C 4E 6B 60 4B 61  |  20
30  |  F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 7A 5E 4C 7E 6E 6F  |  30
40  |  7C C1 C2 C3 C4 C5 C6 C7 C8 C9 D1 D2 D3 D4 D5 D6  |  40
50  |  D7 D8 D9 E2 E3 E4 E5 E6 E7 E8 E9 AD B7 BD 71 6D  |  50
60  |  79 81 82 83 84 85 86 87 88 89 91 92 93 94 95 96  |  60
70  |  97 98 99 A2 A3 A4 A5 A6 A7 A8 A9 C0 BF D0 80 41  |  70
80  |  42 43 44 45 46 47 48 49 51 52 53 54 55 56 57 58  |  80
90  |  90 DE EE 59 62 63 64 65 BC 66 67 4A 68 AC 69 DA  |  90
A0  |  21 A1 23 24 6A 9E E0 B5 29 8D 5F 36 AB 2B EF FE  |  A0
B0  |  0A 20 2A 1A 3F FD BB BA 8F 31 30 33 34 9F 8E 1B  |  B0
C0  |  1E 3E 3B 3D 2D 2C 8A 8B B9 38 39 4F 5A 9C 3A E1  |  C0
D0  |  74 75 EC CC CE 73 76 77 70 1F 1C 18 28 17 19 08  |  D0
E0  |  B0 FA 9B 9A DF CA B3 CB CD ED 9D 78 B2 CF B1 AA  |  E0
F0  |  EA EB AE 8C BE B6 B8 FC AF B4 FB DD DC A0 72 FF  |  F0
    +-----------------------------------------------------+
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

*Figure 73. APL2 ASCII to EBCDIC code point mapping*

The appearance of some national-language-dependent EBCDIC characters may be very similar to APL characters even though they are distinct. The specific characters vary by language, but ones that often cause confusion are:

| **EBCDIC hex** | **EBCDIC ⎕AF** | **APL char** | **Some EBCDIC code points that may appear similar** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 | 128 | ~ | 48 | 58 | 59 | A0 | A1 | BC | BD | DC | |
| AD | 173 | [ | 4A | 63 | 70 | 71 | 90 | 9E | B1 | B5 | BA |
| B7 | 183 | \ | 48 | 68 | 71 | B2 | E0 | EC | | | |
| BD | 189 | ] | 51 | 5A | 68 | 80 | 9F | B5 | BB | FC | |
| DB | 219 | ! | 4F | 5A | 5B | BB | | | | | |
| BF | 191 | \| | 4F | BB | | | | | | | |

The above characters can also cause problems when using non-APL facilities (such as terminal emulators or upload programs) to transfer data between an ASCII system and an EBCDIC system. The problems occur if the facility attempts to map between APL and ASCII code points. You should have no problem if you let APL do the translation, or if you use a terminal emulator that supports the 3270 APL feature.

# Appendix B. APL2 Transfer Files and Extended Transfer Formats

Transfer file formats have been defined to permit exchange of APL2 workspaces or workspace objects among all IBM APL2 implementations. In general, users need only be concerned with the APL commands needed to create and read transfer files, and with the physical requirements of moving files from one system to another. APL2 systems implementers, and occasionally writers of sophisticated applications, also need to understand the internal formats of the files. Each of these is covered below.

## Reading and Writing Transfer Files

The APL commands used to create and read transfer files are `)OUT`, `)IN`, and `)PIN`. The APL system function `⎕TF` also converts individual APL objects between their internal format and a character-based representation that can be used in transfer files. Since `)OUT`, `)IN`, and `)PIN` also take care of all of the additional file format requirements, provide support for multiple objects or entire workspaces, and perform the file I/O itself, they are the preferred technique in almost every case.

## Moving Transfer Files from One System to Another

The techniques for physically moving files from one system to another vary greatly depending on the types of systems and what connections exist between them.

- One key issue is that some systems (for example MVS/TSO and VM/CMS) use an EBCDIC character encoding, while others (for example PC/DOS and AIX/6000) use an ASCII encoding. Both ASCII and EBCDIC transfer file formats are defined, and all IBM APL2 systems accept both formats. No data conversion should be attempted within the file itself when transferring it from one system to another. The receiving APL2 system performs any necessary conversion. If the transfer is done electronically through a network connection, the programs controlling that transfer must be told that this is a "binary" rather than "character" file. (The exact terminology used may vary depending on the system and network control programs being used.)
- Some systems use "record oriented" files while others use stream files. If stream files are being transferred to a system that expects record oriented files, an arbitrary record length may be used, but the existing record separators ("CR" or "CR/LF") must be retained. Conversely, separators should not be inserted when record oriented files are being transferred to a system that expects stream files. Again, the receiving APL2 system adjusts to these differences.
- Within these constraints, standard data transmission commands appropriate to the system such as "ftp put", "SEND", "SENDFILE", or "TRANSMIT" may be used for network transmission, with corresponding commands such as "ftp get" or "RECEIVE" as appropriate to the receiving system.
- Because the receiving APL2 system performs all necessary conversions, it is also possible to use shared DASD, remote file systems, removable media, or other such facilities to transport the data.

# Internal Formats of Transfer Files

The remainder of this appendix deals with data formats within transfer files, and is not of concern to most users.

**Note:** The migration and extended forms of objects defined below can be intermixed within a transfer file.

## File and Record Structure

A transfer file is logically structured as a set of 80-byte records, whether or not the file system is record oriented. For stream files each record must be followed by either a CR (ASCII `X'0D'` or EBCDIC `X'15'`) or a CR/LF combination (ASCII `X'0D0A'`, no EBCDIC equivalent).

**Caution:** Any of these three code points may also be embedded within the data. They must be interpreted as record separators only if they appear immediately following the 80th byte of a record. No record in a transfer file can begin with one of these three code points, so a check for them at the end of a record is unambiguous.

Within each 80-byte record the structure is:

| **Columns** | **Content** |
|---|---|
| 1 | Blank in all records which are part of an object except for the last (or only) record of the object, which contains “`X`”. Records which is not part of an object contain “`*`”. See “Records Not Containing Objects” on page 486 for the usage of those records. Note that the only valid code points (in hex) are ASCII `20`, `2A`, or `58`, or EBCDIC `40`, `5C`, or `E7`. Thus the first byte of a file can be used to determine whether it is in ASCII or EBCDIC format. |
| 2-72 | Part or all of the transfer form representation of an object. Column 2 of the first record for an object must indicate the representation type:<br>`A` An array in extended (`2 ⎕TF`) transfer format.<br>`C` A character array in migration (`1 ⎕TF`) transfer format.<br>`F` A defined function or operator in extended (`2 ⎕TF`) transfer format.<br>`N` A numeric array in migration (`1 ⎕TF`) transfer format.<br>**Note:** The “C” or “N” in column 2 is actually the first character produced by `1 ⎕TF`. The “A” or “F” in column 2 precedes the first character produced by `2 ⎕TF`. A transfer file is not allowed to contain a function in `1 ⎕TF` format.<br>Objects not complete by column 72 of a record are continued in column 2 of the next record. Unused bytes in the last record of an object must be blank. |
| 73-80 | These columns may contain sequence numbers or any other desired information. They are not inspected during `)IN` processing. |

## Records Not Containing Objects

Any record beginning with “*” is either a comment or a special directive. The cases are distinguished by the content of column 2.

| | |
|---|---|
| blank | A comment record, ignored by `)IN` and `)PIN`. |
| ( | A time stamp. This should appear immediately preceding a function or operator definition. Its format is<br>`*(year month day hour min sec millisec)`<br>(i.e. `⎕TS` format) for the time when the function was created or last modified. The timestamp should be adjusted to Greenwich Mean Time if possible. Transmitting systems should supply trailing zeroes if timestamp information is not known precisely.<br>Time stamps should be provided for each defined function or operator if the information is available, but receiving systems must be able to accept defined objects without timestamps. If no timestamp is available, but the receiving system maintains a timestamp for the object type, then the time of receipt should be used. |
| I | Reserved for an “imbed” facility. At present `)OUT` must not produce transfer files containing this directive, and `)IN` and `)PIN` need not support it. |

## Migration Transfer Form of an Object

The migration form is one of two forms of objects that can be intermixed within a transfer file. Migration form can be used for simple arrays (unless they contain complex numbers), but cannot be used for functions or for objects unique to APL2. In migration form an object is represented by four character segments:

1. A data type indicator character, “C” or “N” for simple character or numeric data. “F” (for functions) is also supported as `1 ⎕TF` output, but not in a transfer file. Where “F” is supported, the remaining three fields are based on the matrix form of the function produced by `⎕CR`. (Any `⎕FX` execution attributes are ignored.)
2. The name of the object, followed by a single blank. (there is no blank between the type indicator and the name.)
3. A character representation (digits 0-9) of the rank followed by the shape of the array, with one blank following each numeric item. For example:

   | | |
   |---|---|
   | scalar | “`0 `” |
   | 293 element vector | “`1 293 `” |
   | 17 by 11 matrix | “`2 17 11 `” |

4. A character representation (as produced by `⍕`) of the ravel of the array. Numeric conversions are done as if `⎕PP←18`. Note that trailing blanks must be present, even if this requires extra records in the transfer file, and even though they are indistinguishable from record padding blanks.

## Extended Transfer Form of an Object

The extended form is the second of the two forms of objects that can be intermixed within a transfer file. Extended form can be used for all APL2 objects. There are three basic subformats within extended form, each having the attribute that the object can be recreated by applying the ⍎ primitive to the entire representation:

- The **variable** format, described in detail below. It begins with a name followed by "←". This form is used for shared variables and system variables as well as ordinary variables in the workspace.
- The **function** format, used for defined functions and defined operators in the workspace. It begins with ⎕*FX*, preceded by a character representation of the four ⎕*FX* execution attributes if any of them are nonzero, and continues with the lines of the function, each represented as a quoted character constant, with a single blank between each constant.
- The **external object** format, used for all objects external to the workspace regardless of their class. This format is a reconstruction of the ⎕*NA* function originally used to make a workspace association to the object, though with the ⎕*NA* arguments represented as constants regardless of how they may have originally been provided.

Conceptually the **variable** format could contain, to the right of the "←", any valid APL statement which returns a result and has no side effects. In practice only a very limited set of constructs is permitted in transfer files, to allow APL2 systems to optimize processing during )*IN*. For the same reason, redundant blanks and parentheses are prohibited. The overall object representation is:

***name←value***

where ***value*** must be one of the following:

| Construct | Format or Example |
|---|---|
| ***scalar*** constant | e.g. '*Q*' or 3.26*J*1*E*¯5 |
| empty vector | must be '' or 0ρ0 |
| one-element vector | 1ρ***scalar***, e.g. 1ρ'*A*' or 1ρ2 |
| vector constant | e.g. '*ABC*' or 6 ¯2 0 |
| ***progression*** expression | (see below) |
| rank 2 or higher array | ***shape***ρ***data*** (see below), e.g. 2 2ρ'*ABCD*' |
| enclosed value | ⊂***value*** (recursive), e.g. ⊂2 1 4 |

where ***progression*** applies to an array of integers of any rank, 1 or greater, where the difference between all pairs of adjacent values (in raveled order) is a constant. The representation is ***first***-***incr***×⎕*IO*-ι***count***.

| Example Value | Representation |
|---|---|
| 9 8 7 | 9-¯1×⎕*IO*-ι3 |
| ¯3 0 3 6 | ¯3-3×⎕*IO*-ι4 |

and where ***data*** must be one of the following, defining exactly ×/***shape*** items:

| Construct | Format or Example |
|---|---|
| empty vector | must be '' or 0ρ0 |
| scalar or vector constant | e.g. '*ABCDE*' or ¯3.14159 |
| ***progression*** expression | (see above) |
| extended character vector | using ⎕*AF*, e.g. ⎕*AF* 19677897 19677890 |
| enclosed value | ⊂***value*** (recursive), e.g.⊂'*XYZ*' |

| | |
|---|---|
| strand expression | two or more items in any combination from the following:<br>• a scalar or empty character constant<br>• a scalar or vector numeric constant<br>• a parenthesized ***value*** (recursive) |

# Appendix C. System Limitations for APL2

System limitations for APL2 vary depending on the APL2 product. Figure 74 shows the limitations imposed on APL2 on the various systems by the nature of the implementation. Because they interact, a particular limitation may not be attainable.

*Figure 74. Limitations by System*

| Limitation | Workstations | APL2/370 | APL2/PC | |
|---|---|---|---|---|
| | | | **16-Bit** | **32-Bit** |
| Largest and smallest representable numbers in an array | 1.7976931348623158*E*+308<br>and<br>¯1.7976931348623158*E*+308 | 7.2370055773322621*E*75<br>and<br>¯7.2370055773322621*E*75 | 1.7976931348623158*E*+308<br>and<br>¯1.7976931348623158*E*+308 | 1.7976931348623158*E*+308<br>and<br>¯1.7976931348623158*E*+308 |
| Most infinitesimal (near 0) representable numbers in an array | 2.2250738585072014*E*¯308<br>and<br>¯2.2250738585072014*E*¯308 | 5.397605346934027891*E*¯79<br>and<br>¯5.397605346934027891*E*¯79 | 2.2250738585072014*E*¯308<br>and<br>¯2.2250738585072014*E*¯308 | 2.2250738585072014*E*¯308<br>and<br>¯2.2250738585072014*E*¯308 |
| Maximum rank of an array | 63 | 64 | 63 | 63 |
| Maximum length of any axis in an array | ¯1+2*31 (2147483647) | ¯1+2*31 (2147483647) | 65520 | ¯1+2*31 (2147483647) |
| Maximum product of all dimensions in an array | ¯1+2*31 (2147483647) | ¯1+2*31 (2147483647) | 65520 | ¯1+2*31 (2147483647) |
| Maximum depth of an array applied with the primitive functions depth (≡*R*) and match (*L*≡*R*) | 181 | 181 | No limit | No limit |
| Maximum depth of a shared variable | 181 | 181 | 6 | 6 |
| Maximum depth of a copied variable | 181 | 181 | 168 | 168 |
| Maximum number of characters in the name of a shared variable | 255 | 255 | 18 | 18 |
| Maximum number of characters in a comment (minus leading blanks) | 4090 | 32764 | 4089 | 4089 |
| Maximum length of line | 8190 | N/A | 4096 | 4096 |
| Maximum number of lines in a defined function or operator | ¯1+2*15 (32767) | ¯1+2*31 (2147483647) | 700 | 545 |
| Maximum number of labels in a defined function or operator | Limited by number of lines | 32767 | 700 | 545 |
| Maximum number of local names (excluding labels) in a defined function or operator | Limited by lengths of lines and names | 32767 | Limited by lengths of lines and names | Limited by lengths of lines and names |
| Bytes in internal symbol label | 8191 | N/A | 64999 | 4095 |
| Maximum number of slots in the internal symbol table. A slot is required for each unique name, each unique constant, and each ill-formed constant in the workspace. | N/A | 32767 | N/A | N/A |
| Maximum value of ⎕*PW* | 254 | 390 | 254 | 254 |
| Maximum value of ⎕*PP* | 16 | 18 | 16 | 16 |

# Bibliography

## APL2 Publications

- *APL2 Fact Sheet,* GH21-1090
- *APL2/370 Application Environment Licensed Program Specifications,* GH21-1063
- *APL2/370 Licensed Program Specifications,* GH21-1070
- *APL2 for AIX/6000 Licensed Program Specifications,* GC23-3058
- *APL2 for Sun Solaris Licensed Program Specifications,* GC26-3359
- *APL2/370 Installation and Customization under CMS,* SH21-1062
- *APL2/370 Installation and Customization under TSO,* SH21-1055
- *APL2 Migration Guide,* SH21-1069
- *APL2 Programming*: *Language Reference,* SH21-1061
- *APL2/370 Programming*: *Processor Interface Reference,* SH21-1058
- *APL2 Reference Summary,* SX26-3999
- *APL2 Programming*: *An Introduction to APL2,* SH21-1073
- *APL2 for AIX/6000*: *User's Guide,* SC23-3051
- *APL2 for OS/2*: *User's Guide,* SH21-1091
- *APL2 for Sun Solaris*: *User's Guide,* SH21-1092
- *APL2 for the IBM PC*: *User's Guide,* SC33-0600
- *APL2 GRAPHPAK*: *User's Guide and Reference,* SH21-1074
- *APL2 Programming*: *Using Structured Query Language,* SH21-1057
- *APL2/370 Programming*: *Using the Supplied Routines,* SH21-1056
- *APL2/370 Programming*: *System Services Reference,* SH21-1054
- *APL2/370 Diagnosis Guide,* LY27-9601
- *APL2/370 Messages and Codes,* SH21-1059

## Other Books You Might Need

The following book is recommended:

- *APL2 at a Glance*, by James Brown, Sandra Pakin, and Raymond Polivka, published by Prentice-Hall, ISBN 0-13-038670-7 (1988). Copies can be ordered from IBM as SC26-4676.

See your system-specific user's guide for other books you might need for your operating system.

## APL2 Keycaps and Decals

Plastic replacement keyboard keycaps are available from IBM as:

- *APL2 Keycaps (US and UK base set)*, SX80-0270
- *APL2 Keycaps, German upgrade to SX80-0270*, SX23-0452
- *APL2 Keycaps, French upgrade to SX80-0270*, SX23-0453
- *APL2 Keycaps, Italian upgrade to SX80-0270*, SX23-0454

*APL2 Keyboard Decals*, SC33-0604, can also be ordered from IBM.

# Index

# Readers' Comments — We'd Like to Hear from You

**APL2 Programming:**
**Language Reference**

**Publication No. SH21-1061-01**

## Overall, how satisfied are you with the information in this book?

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|---|---|---|---|---|---|
| Overall satisfaction | ☐ | ☐ | ☐ | ☐ | ☐ |

## How satisfied are you that the information in this book is:

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|---|---|---|---|---|---|
| Accurate | ☐ | ☐ | ☐ | ☐ | ☐ |
| Complete | ☐ | ☐ | ☐ | ☐ | ☐ |
| Easy to find | ☐ | ☐ | ☐ | ☐ | ☐ |
| Easy to understand | ☐ | ☐ | ☐ | ☐ | ☐ |
| Well organized | ☐ | ☐ | ☐ | ☐ | ☐ |
| Applicable to your tasks | ☐ | ☐ | ☐ | ☐ | ☐ |

## Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

| | |
|---|---|
| Name | Address |
| Company or Organization | |
| Phone No. | |

## Readers' Comments — We'd Like to Hear from You

SH21-1061-01

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

&@rcfaddr(1)

Fold and Tape **Please do not staple** Fold and Tape

SH21-1061-01

Cut or Fold
Along Line

File Number: S370-40
Program Number: 5688-228

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

**The APL2 Library**

| | |
|---|---|
| GH21-1090 | APL2 Family of Products (fact sheet) |
| SH21-1073 | APL2 Programming: An Introduction to APL2 |
| SH21-1061 | APL2 Programming: Language Reference |
| SX26-3999 | APL2 Reference Summary |
| SH21-1074 | APL2 GRAPHPAK: User's Guide and Reference |
| SH21-1057 | APL2 Programming: Using Structured Query Language |
| SH21-1069 | APL2 Migration Guide |
| SC33-0600 | APL2 for the IBM PC: User's Guide |
| SC33-0601 | APL2 for the IBM PC: Reference Summary |
| SC33-0851 | APL2 for the IBM PC: Reference Card |
| SH21-1091 | APL2 for OS/2: User's Guide |
| GC23-3058 | APL2 for AIX/6000 Licensed Program Specifications |
| SC23-3051 | APL2 for AIX/6000: User's Guide |
| GC26-3359 | APL2 for Sun Solaris Licensed Program Specifications |
| SH21-1092 | APL2 for Sun Solaris: User's Guide |
| GH21-1063 | APL2/370 Application Environment Licensed Program Specifications |
| GH21-1070 | APL2/370 Licensed Program Specifications |
| SH21-1062 | APL2/370 Installation and Customization under CMS |
| SH21-1055 | APL2/370 Installation and Customization under TSO |
| SH21-1054 | APL2/370 Programming: System Services Reference |
| SH21-1056 | APL2/370 Programming: Using the Supplied Routines |
| SH21-1058 | APL2/370 Programming: Processor Interface Reference |
| LY27-9601 | APL2/370 Diagnosis Guide |
| SH21-1059 | APL2/370 Messages and Codes |

SH21-1061-01

*Spine information:*

IBM APL2 Programming: Language Reference