APL64 Primitive Functions

Basic Information

This chapter summarizes the APL primitive functions and operators. Each synopsis contains the syntax, a description, and one or more examples. In some examples, a variable or result is shown in display form to illustrate the data structures. The DISPLAY function from the UTILITY workspace and the]display user command in APL64 produce the display form. There are also descriptions of other symbols that APL uses that are neither functions nor operators. This chapter uses the following conventions:

• *res*	the explicit result
• *arg*	the argument of a monadic function
• *larg*	the left argument of a dyadic function
• *rarg*	the right argument of a dyadic function
• *conforming*	the shapes of the left and right arguments must agree in some manner; frequently, they must have the same shape or extend to the same shape
• *i*	a non-negative scalar that designates an axis or a dimension of an array
• *idx*	an index or a variable with valid indices
• *f, g*	any functions, primitive, system, or user-defined, that act as operands to an operator
• *oparray*	an array that acts as an operand to an operator
• *ext*	an external factor that affects the result of an operation (for example, ⎕ct, ⎕rl, ⎕io)
• ←→	equivalence; expressions on both sides of the double arrow return the same result
• scalar	a scalar function calculates the result using one data element at a time, or one datum from the left argument with the corresponding datum from the right argument. Each element of the result is computed independently and depends solely on the corresponding left and right items. The entire result has the same shape as at least one of the arguments.

Note: As with a language dictionary that uses words to provide examples of the usage of other words, this manual uses some primitive functions in the examples of how other primitives work. If you are unfamiliar with APL, you should take particular note of the dyadic function Reshape (ρ) and the monadic function Index generator (ι).

Contents

Numeric Functions

+ Plus

Syntax:

res ← larg + rarg

res +← rarg is equivalent to res ← res + rarg

Description:

Scalar. Add two numbers.

larg, rarg*:* any numeric arrays (conforming).

res*:* each item of *larg* added to corresponding item of rarg.

Example:

```
    ¯2  2  2  +  3.5  1  ¯2
 1.5 3 0
```

+ Conjugate

Syntax:

res ← + arg

Description:

Scalar. Return the value of a number.
arg: any numeric array.
res: same as 0+arg.

Example:

```
    +¯27.34     18      6
 ¯27.34 18 6
```

- Minus

Syntax:

res ← larg - rarg
res -← rarg   res ← res - rarg

Description:
Scalar. Subtract two numbers.
larg, rarg: any numeric arrays (conforming).
res: each item of rarg subtracted from the corresponding item of larg.

The Minus function is not the same as the High minus (¯) symbol, which designates a negative number.
Example:

```
    ¯2 2 2 - 3.5 1 ¯2
 ¯5.5 1 4
```

- Negate
Syntax:

res ← - arg

Description:

Scalar. Change the sign of a number.

arg: any numeric array.

res: each item of arg subtracted from 0.

Example:

```
      - 2 ¯2 1.5
¯2 2 ¯1.5
```

× Times
Syntax:

res ← larg × rarg

res ×← rarg ←→ res ← res × rarg

Description:

Scalar. Multiply two numbers.

larg, rarg: any numeric arrays (conforming).

res: each item of larg multiplied by the corresponding item of rarg.

Example:

```
      ¯2 2 2 × 3.5 0 2
¯7 0 4
```

Note: For monadic ×, see Signum in the "General Functions" section of this chapter.

× Signum
Syntax:

res ← × arg

Description:

Scalar. Determine the sign of a number.

arg: any numeric array.

res: ¯1 if arg is negative, 0 if arg is 0, and 1 if arg is positive.

Example:

```
      × 3 0 ¯0.5
1 0 ¯1
```

÷ Divide
Syntax:

res ← larg ÷ rarg

res ÷← rarg   res ← res ÷ rarg

Description:
Scalar. Divide two numbers.
larg: any numeric array.
rarg: any numeric array (conforming).
res: each item of larg divided by the corresponding item of rarg. Division by zero causes a DOMAIN ERROR except in the case of 0÷0, which returns 1.

Example:

```
     2 ¯3 0 ÷ 1 3 0
2 ¯1 1
     0÷0
1
```

÷ Reciprocal

Syntax:

res ← ÷ arg

Description:
Scalar. Find the reciprocal of a number.
arg: any nonzero numeric array.
res: 1 divided by each item of arg.

Example:

```
     ÷ 2 ¯1 ¯0.5
0.5 ¯1 ¯2
```

⌈ Maximum

Syntax:

res ← larg - rarg

res -← rarg ←→ res ← res - rarg

Description:
Scalar. Select the greater of two numbers.
larg, rarg: any numeric arrays (conforming).
res: the larger of each corresponding pair of numbers in larg and rarg.

Example:

```
     ¯3.2 ¯4.1 - 7 ¯4.2
7 ¯4.1
```

⌈ Ceiling

Syntax:

res ← - arg

Description:
Scalar. Round up to the nearest integer.

arg: any numeric array.
res: smallest integer greater than or equal to arg.
ext: ⎕ct.
Example:

```
    - 3.1416 ¯1.5 6
4 ¯1 6
```

⌊ Minimum

Syntax:

    res ← larg ⌊ rarg

Description:
Scalar. Select the lesser of two numbers.
larg, rarg: any numeric arrays (conforming).
res: the lesser of each corresponding pair of numbers in larg and rarg.
Example:

```
    ¯3.2 ¯4.1 ⌊ 7 ¯4.2
¯3.2 ¯4.2
```

⌊ Floor

Syntax:

    res ← ⌊ arg
    res ⌊← rarg ←→ res ← res ⌊ rarg

Description:
Scalar. Round down to the nearest integer.
arg: any numeric array.
res: largest integer less than or equal to arg.
ext: ⎕ct.
Example:

```
    ⌊ 3.1416 ¯1.5 6
3 ¯2 6
```

* Power

Syntax:

    res ← larg * rarg
    res *← rarg ←→ res ← res * rarg

Description:
Scalar. Raise a number to a specific power.
larg, rarg: any numeric arrays (conforming), except that if rarg is fractional, larg must be non-negative.

res: larg raised to the corresponding rarg power.
Example:

```
    2 49 4 0 * 3 0.5 ¯1 40
8 7 0.25 0
```

* Exponential
Syntax:
        res ← * arg
Description:
Scalar. Raise e to a power.
arg: any numeric array.
res: e (2.71828...) raised to the power specified by each item of arg.
Example:

```
    * 1 ¯1 0
2.718281828 0.3678794412 1
```

? Roll
Syntax:
        res ← ? arg
Description:
Scalar. Select a pseudorandom integer.
arg: any positive integer array.
res: an integer picked at random from the set of numbers given by ι arg[n]; res contains a pseudorandom number for each element of arg where ⎕io ≤ res[n] ≤ arg[n]+⎕io-1.
ext: ⎕io, ⎕rl.
Example:

```
    ? 2000 12 30
1969 2 23
```

? Deal
Syntax:
        res ← larg ? rarg
Description:
Select a set of unique pseudorandom integers.
larg, rarg: non-negative integer scalars where larg ≤ rarg.
res: larg elements selected pseudorandomly without replacement from ιrarg.
ext: ⎕io, ⎕rl.
Example:

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

Note: For additional information, refer to Random Number Generator below in this document.

⌹ Matric Divide

Syntax:

res ← larg ⌹ rarg

Description:

Solve a set of simultaneous equations.

larg, rarg: numeric scalars, vectors, or matrices; the rank of rarg must equal or exceed the rank of larg; if rarg is a matrix, the last dimension must not exceed the first.

res: the solution (or a least squares approximation if rarg has more rows than columns) of the matrix equation: larg   rarg +.× res.

Example:

```
    14 26⌹2 2ρ1 3 4 2
5 3
    14 26 7⌹3 2ρ1 3 4 2 1 1
4.981481481  2.944444444
```

⌹ Matrix Inverse

Syntax:

res ← ⌹ arg

Description:

Calculate the inverse of a matrix.

arg: numeric scalar, vector, or matrix.

res: inverse of arg if arg is nonsingular and square.

Example:

```
    ⌹ 2 2 ρ 1 1 2 3
 3 ¯1
¯2  1
```

⍟ Logarithm

Syntax: res ← larg ⍟ rarg

res ⍟← rarg ←→ res ← res ⍟ rarg

Description:

Scalar. Compute the logarithm of a number.

larg, rarg: any positive numeric arrays (conforming).

res: the logarithm of each element of rarg to the corresponding base in larg.

Example:

```
    2 49 4 ⍟ 8 7 0.25
3 0.5 ¯1
```

⍟ Natural Logarithm

Syntax:

res ← ⍟ arg

Description:

Scalar. Compute the natural logarithm of a number.

arg: any positive numeric array.

res: logarithm (base e) applied to each item of arg.

Example:

```
      ⍟ 1 10 2.7182818284
0 2.302585093 1
```

! Factorial

Syntax:

res ← ! arg

Description:

Scalar. Compute the factorial of a number.

arg: any numeric array, excluding negative integers.

res: if arg is a positive integer, res is the product of all positive integers from 1 through arg. If arg is 0, res is 1. All fractional numbers are computed using the gamma function on arg+1; the function is undefined for negative integers.

Example:

```
      ! 0 4 2.5
1 24 3.32335097
```

! Binomial

Syntax:

res ← larg ! rarg

res !← rarg ←→ res ← res ! rarg

Description:

Scalar. Find the number of possible combinations for a set of objects.

larg, rarg: any positive numeric arrays (conforming).

res: the number of possible combinations of rarg objects selected larg at a a time.

Example:

```
      1 2 5 ! 5 4 5
5 6 1
```

| Magnitude (Absolute Value)

Syntax:

res ← | arg

Description:

Scalar. Compute the absolute value of a number.
arg: any numeric array.
res: absolute value of each element of arg.
Note: For the EN-US keyboard definition, because APL64 is Unicode aware, APL64 distinguishes between □av[124+□io] (decimal Unicode 124) and □av[254+□io] (decimal Unicode 8739). Decimal Unicode 8739 is the Unicode glyph specified for the Magnitude and Residue primitive functions. Therefore, unlike APL+Win, the Magnitude and Residue glyph is available from the US-EN keyboard using only Alt+M. For the EN-US keyboard Shift+\ generates the decimal Unicode 124 glyph. In the APL64 font, the decimal Unicode 124 and 8739 glyphs appear graphically identical.
Example:

```
    ∣2 0 ¯1.6
2 0 1.6
```

∣ Residue

Syntax:

res ← larg ∣ rarg
res ∣← rarg ←→ res ← res ∣ rarg

Description:
Scalar. Find the remainder.
larg, rarg: any numeric arrays (conforming).
res: the remainder after dividing each item of rarg by the corresponding item of larg; that is:
rarg -(⌊rarg÷larg)×larg.
ext: □ct.
Note: For the EN-US keyboard definition, because APL64 is Unicode aware, APL64 distinguishes between □av[124+□io] (decimal Unicode 124) and □av[254+□io] (decimal Unicode 8739). Decimal Unicode 8739 is the Unicode glyph specified for the Magnitude and Residue primitive functions. Therefore, unlike APL+Win, the Magnitude and Residue glyph is available from the US-EN keyboard using only Alt+M. For the EN-US keyboard Shift+\ generates the decimal Unicode 124 glyph. In the APL64 font, the decimal Unicode 124 and 8739 glyphs appear graphically identical.
Example:

```
    2 ¯2 1∣3 3 3.14159
1 ¯1 0.14159
```

○ Trigonometric functions

Syntax:

res ← larg ○ rarg
res ○← rarg ←→ res ← res ○ rarg

Description:
Scalar. Compute a trigonometric function for a number.
larg: any array of integers in the range ¯7 to +7.
rarg: any valid numeric array.
res: the trigonometric function selected by larg applied to each corresponding item in rarg.

Note: All angular right arguments and results are measured in radians. You can think of this function as being 15 different functions, where the left argument is a code that specifies the function you want to use.

larg	**function**	*larg*	**function**
¯7	ARCTANH	7	TANH
¯6	ARCCOSH	6	COSH
¯5	ARCSINH	5	SINH
¯4	(¯1+ rarg⋆2)⋆.5	4	(1+ rarg⋆2)⋆.5
¯3	ARCTAN	3	TAN
¯2	ARCCOS	2	COS
¯1	ARCSIN	1	SIN
0	(1- rarg⋆2)⋆.5		

Examples:

```
    0 ○ .6
0.8

    2 ○ 3.14159
¯1
    ¯3 ○ 0 1 2
0 0.7853981634 1.107148718
    1 2 3 ○ (○÷4)
0.7071067812 0.7071067812 1
```

○ Pi Times

Syntax:

res ← ○ arg

Description:

Scalar. Multiply a number by pi (3.141592...).

arg: any numeric array.

res: arg multiplied by pi.

Example:

```
    ○ 1 2 0
3.141592654 6.283185307 0
```

⊥ Base Value

Syntax: res ← larg ⊥ rarg

Description:

Find the base value of a number whose successive digits are the elements of rarg.

larg, rarg: any numeric arrays (conforming).

res: the expression of rarg in radix larg. The representation of res is decimal.

This function is also known as Decode.

Relevant Definitions:

Base: the number of units in a given digit's place that is required to give 1 in the next higher place.
Radix: the base of a place-value notation.
Example:

```
    10 ⊥ 1 7 7 6       ⍝ Decimal radix
1776
    2⊥1 0 1 0          ⍝ Binary radix
10
    7 24 60 ⊥ 3 12 50  ⍝ Days/Hours/Minutes mixed radix
5090
    10 3 2 10 ⊥ 1 7 7 6  ⍝ Arbitrary mixed radix
276
       2 10 ⊥   7 6    ⍝ Build the above example step by step
76
     3 2 10 ⊥  7 7 6
216
     0 3 2 10 ⊥ 1 7 7 6   ⍝ Note the first element of larg is only
276                 a placeholder to make the arguments conform
```

Note: res ←→ rarg[n] + (rarg[n-1] × larg[n]) + (rarg[n-2] × (larg[n-1] × larg[n])) + . . .
+(rarg[n-m] × (×/ larg[(n-m) + ι(m)])) + . . . + (rarg[1] × (×/ larg[1 + ι(n-1)]))

⊤ Representation

Syntax:

res ← larg ⊤ rarg

Description:
Find the representation of a number or numbers (represented in decimal notation) in another radix.
larg, rarg: any numeric arrays.
res: the expression of each element of rarg represented in a number system described by larg.
This function is also known as Encode.
Example:

```
    10 10 10 10 ⊤ 1776    ⍝ Decimal radix
1 7 7 6
    2 2 2 ⊤ 5          ⍝ Binary radix
1 0 1
    7 24 60 ⊤ 5090       ⍝ Days/Hours/Minutes mixed radix
3 12 50
    0 24 60 ⊤ 5090 6666    ⍝ Independent values in rarg.
 3   4
12   15
50   6
```

⍨ Commute

Syntax: res ← larg f⍨ rarg

Description:
f is a dyadic function.
larg, rarg: any appropriate arrays (conforming).
res: The result of the function with the arguments reversed. (larg f⍨ rarg ←→ rarg f larg)

Note: The use of the commute operator with the / and \ primitive functions will result in a NONCE ERROR. This limitation might be addressed in a future release. In the interim, this suggested workaround is to use the system functions, ⎕repl and ⎕expand, in place of / and \, respectively.

Example:

```
    10 - 7
3
    7 -⍨10
3
    10 minus 7    ⍝ minus is a user-defined function
3
    7 minus⍨ 10
3
    10 20 30 ∘.+ 1 2 3
 11 12 13
 21 22 23
 31 32 33
    10 20 30 ∘.+⍨ 1 2 3
 11 21 31
 12 22 32
 13 23 33
```

Boolean Functions

Boolean arrays consist of only two values, 0 and 1. You can think of these values as representing the absence or presence of a characteristic, no/yes, or false/true. A function that returns a Boolean array describes a relationship between the data arrays that are its arguments. The argument arrays can be of other data types.

Functions that use only Boolean arrays as arguments and also return a Boolean array are called Logical functions. You can also use Boolean arrays as arguments to non-Boolean functions. The results may not be Boolean. For example, you can sum a Boolean vector to find the number of occurrences.

Scalar Functions that Return a Boolean Array

< Less Than

Syntax:

    res ← larg < rarg
    res <← rarg ←→ res ← res < rarg

Description:
Scalar. Compare two numeric arrays.
larg, rarg: any numeric arrays (conforming).
res: 1 for each pair of corresponding values where larg is less than rarg; otherwise, 0.

ext: ⎕ct.
Example:

```
    1 2 3<2 1 3
1 0 0
```

= Equal

Syntax:

res ← larg = rarg

res =← rarg ←→ res ← res = rarg

Description:
Scalar. Compare two arrays for equality.
larg, rarg: any arrays (conforming).
res: 1 for each pair of corresponding values where larg and rarg are equal; otherwise, 0.
ext: ⎕ct.
Example:

```
    'O'='COGNOS CORPORATION'
0 1 0 0 1 0 0 0 1 0 0 1 0 0 0 0 1 0
```

≠ Not Equal

Syntax:

res ← larg ≠ rarg

res ≠← rarg ←→ res ← res ≠ rarg

Description:
Scalar. Compare arrays for inequality.
larg, rarg: any arrays (conforming).
res: 1 for each pair of corresponding values where larg and rarg are not equal; otherwise, 0.
ext: ⎕ct.
Example:

```
    1 2 3≠2 1 3
1 1 0
```

> Greater Than

Syntax:

res ← larg > rarg

res >← rarg ←→ res ← res > rarg

Description:
Scalar. Compare two numeric arrays.
larg, rarg: any numeric arrays (conforming).
res: 1 for each pair of corresponding values where larg is greater than rarg; otherwise, 0.
ext: ⎕ct.
Example:

```
    1 2 3 > 2 1 3
0 1 0
```

≥ Greater Than or Equal

Syntax:

res ← larg ≥ rarg

res ≥← rarg ←→ res ← res ≥ rarg

Description:

Scalar. Compare two numeric arrays.

larg, rarg: any numeric arrays (conforming).

res: 1 for each pair of corresponding values where larg is greater than or equal to rarg; otherwise, 0.

ext: ⎕ct.

Example:

```
    1 2 3 ≥ 2 1 3
0 1 1
```

≤ Less Than or Equal

Syntax:

res ← larg ≤ rarg

res ≤← rarg ←→ res ← res ≤ rarg

Description:

Scalar. Compare two numeric arrays.

larg, rarg: any numeric arrays (conforming).

res: 1 for each pair of corresponding values where larg is less than or equal to rarg; otherwise, 0.

ext: ⎕ct.

Example:

```
    1 2 3 ≤ 2 1 3
1 0 1
```

Non-scalar Functions that Return a Boolean Array

⍷ Find

Syntax:

res ← larg ⍷ rarg

Description:

Search for an array in another array.

larg, rarg: any arrays, nested or heterogeneous.

res: Boolean array of same shape as rarg with 1s at locations of first element of copies of larg.

ext: ⎕ct.

Example:

```
    'ISSI' ⍷ 'MISSISSIPPI'
0 1 0 0 1 0 0 0 0 0 0

    2 3 ⍷ 3 4⍴1 2 3
0 1 0 0
1 0 0 0
0 0 1 0

    (2 2⍴2 3 3 1)⍷3 4⍴1 2 3
0 1 0 0
1 0 0 0
0 0 0 0

    '' ⍷ 'ABCDE'
1 1 1 1 1

    (⊂1,⊂2 3)⍷¨'XYZ',(⊂1,⊂2 3),4,⊂2 3⍴1,⊂2 3
0 0 0  1 0  0   1 0 0
              0 1 0
```

≡ Match

Syntax:

res ← larg ≡ rarg

Description:

Determine the equivalence of two arrays.

larg, rarg: any arrays.

res: 1 if both larg and rarg have the same rank, shape, and values; otherwise, 0.

ext: ⎕ct.

Example:

```
    'XYZZY' ≡ 1 5⍴'XYZZY'
0
    0 ≡ ,0
0
    A←2 3⍴⍳4
    A ≡ A
1
    A ≡ 'A'
0
```

Depth ≡

Syntax:

res ← ≡ arg

Description:

Levels of nesting in an array.

arg: any array.

res: the number of times you must use Pick (dyadic ⊃) to extract the most deeply nested simple scalar from arg.
Example:

```
    ≡3.3
0
    ≡1 2 3
1
    ≡''
1
    ≡ 2 3 4ρι24
1
    ≡(1 2) (2 3) 'AB'
2
    ≡⊂⊂⊂⊂⊂12 12
6
```

∊ Member of
Syntax:

    res ← larg ∊ rarg

Description:
Compare the contents of two arrays.
larg , rarg: any arrays.
res: same size as larg and contains a 1 if larg item is found anywhere in rarg; otherwise, 0.
ext: ⎕ct.
Example:

```
    2 5 ∊ 1 2 3 4
1 0
```

Note: For monadic ∊, see Enlist in the "Structural Functions" section of this chapter.

Scalar Logical Functions that Use and Return Boolean Arrays
The functions described below not only return a Boolean array, they require Boolean arguments. In addition to these functions, you can also use the six scalar functions that return a Boolean array as logical functions if you use Boolean arrays as arguments.

^ AND
Syntax:

    res ← larg ^ rarg
    res ^← rarg ←→ res ← res ^ rarg

Description:
Scalar. Logical AND of two Boolean arrays.
larg, rarg: any Boolean arrays (conforming).
res: 1 if both larg and rarg are 1; otherwise, 0.
Example:

```
    0 0 1 1 ^ 0 1 0 1
0 0 0 1
```

∨ OR

Syntax:

res ← larg ∨ rarg

res ∨← rarg ←→ res ← res ∨ rarg

Description:

Scalar. Logical OR of two Boolean arrays.

larg, rarg: any Boolean arrays (conforming).

res: 1 if either larg or rarg is 1; otherwise, 0.

Example:

```
    0 0 1 1 ∨ 0 1 0 1
0 1 1 1
```

~ Without

Syntax:

res ← larg ~ rarg

Description:

Set difference.

larg: any scalar or vector.

rarg: any array.

res: vector that contains those elements of larg that do not occur anywhere in rarg. ext: ⎕ct.

Example:

```
    1 1 2 2 3 3 4 4~4 2 4
1 1 3 3
    R←'AEIOU'
    L←'ANY ARRAY'
    L~R
NY RRY
    L~L~R
AAA
```

~ NOT

Syntax:

res ← ~ arg

Description:

Scalar. Return the complement of a Boolean array.

arg: any Boolean array.

res: 1 for each item of arg that is 0; 0 for each item that is 1.
Example:

```
    ~ 0 1
1 0
```

≢ Not Match

Syntax:

res ← larg ≢ rarg

Description:
Determine the non-equivalence of two arrays. (larg ≢ rarg ←→ ~ rarg ≡ larg)
larg, rarg: any arrays.
res: 1 if both larg and rarg does not have the same rank, shape, and values; otherwise, 0.
This function is also known as "not match" and "inequivalent".
ext: ⎕ct.
Example:

```
    'bex' ≢ 'b','e','x'
0
    ⍬≢⍳0
0
    ''≢⍳0
1
```

⍱ NOR

Syntax:

res ← larg ⍱ rarg

res ⍱← rarg ←→ res ← res ⍱ rarg

Description:
Scalar. Logical NOR of two Boolean arrays.
larg, rarg: any Boolean arrays (conforming).
res: 1 if both larg and rarg are 0; otherwise, 0; (equivalent to ~ (larg ∨ rarg)).
Example:

```
    0 0 1 1 ⍱ 0 1 0 1
1 0 0 0
```

⍲ NAND

Syntax:

res ← larg ⍲ rarg

res ⍲← rarg ←→ res ← res ⍲ rarg

Description:
Scalar. Logical NAND of two Boolean arrays.
larg, rarg: any Boolean arrays (conforming).

res: 0 if both larg and rarg are 1; otherwise, 1 (equivalent to ~ (larg ^ rarg)).
Example:

```
    0 0 1 1⍲0 1 0 1
1 1 1 0
```

Structural Functions

Structural functions reorganize data in arrays; select subsets of data in arrays; or select and reorganize data in arrays. These functions can change data by replacing items or by restructuring their relationships, such as by creating a nested array, but these functions do not recalculate an individual datum. Some functions create new data, for example, inserting fill items. You can recalculate data at the same time you reorganize them by using numeric functions in the arguments to structural functions. The descriptions in this section are alphabetical.

, Catenate

Syntax:

res ← larg , rarg
res ← larg ,[i] rarg
res ,← rarg (see Note 3)

Description:
Join two arrays along an axis specified by the index to the shape vector. The default axis for the unadorned comma (,) is along the last dimension.
larg, rarg: any arrays (conforming). Conforming arrays for this function do not necessarily have the same shape. The two arrays can have different lengths along the axis you specify for catenation. You can even catenate arrays whose ranks differ by one, if the other dimensions are equal. The result array extends the extra dimension of the higher (rank) array. You can also catenate a scalar to an array of any rank.
i: scalar that indicates the dimension desired.
res: If i is an integer, an array consisting of the two arrays joined along the specified axis. If i is fractional, a new dimension is added to the shape vectors of rarg and larg in the position between ⌊i and -i, and the function acts along that axis. Catenate with a fractional axis is also known as Laminate.
ext: ⎕io (for axis or dimension).

Example:

```
    2 3 5,99
2 3 5 99

     (2 3ρι6),2 2ρ33 333 66 666
1  2  3  33 333
4  5  6  66 666

    B←'HOW' ,[.5] 'NOW'
    B
HOW
NOW
```

```
    'HOW' ,[1.5] 'NOW'
HN
OO
WW
    ⎕io←0
    B←'HOW' ,[¯0.5] 'NOW'
    B
HOW
NOW
```

Note 1: larg ,[⎕io] rarg
larg , rarg

Note 2: Ravel (monadic ,) is also in this section.

Note 3: res ,← rarg   res ← res , rarg
The syntax is similar to the in-place operations in other programming languages like C++ and C#. However, in APL64, besides the shorter notation, this can also provide significant performance gains particularly in cases that involve repetitive catentations with large arrays such as inside an iterative control structure (a :for loop).

⍪ Catenate

Syntax:

res ← larg ⍪ rarg
res ← larg ⍪[i] rarg
res ⍪← rarg (see Note 2)

Description:
Join two arrays along an axis specified by the index to the shape vector. The default axis for catbar (⍪) is along the first dimension; the default axis for the unadorned comma (,) is along the last dimension.
larg, rarg: any arrays (conforming). Conforming arrays for this function do not necessarily have the same shape. The two arrays can have different lengths along the axis you specify for catenation. You can even catenate arrays whose ranks differ by one, if the other dimensions are equal. The result array extends the extra dimension of the higher (rank) array. You can also catenate a scalar to an array of any rank.
i: scalar that indicates the dimension desired.
res: If i is an integer, an array consisting of the two arrays joined along the specified axis. If i is fractional, a new dimension is added to the shape vectors of rarg and larg in the position between ⌊i and -i, and the function acts along that axis. Catenate with a fractional axis is also known as Laminate.
ext: ⎕io (for axis or dimension).

Example:

```
    2 3 5⍪99
2 3 5 99

    'THREE'⍪2 5⍴'BLINDMICE '
THREE
```

```
BLIND
MICE
```

Note 1: larg ⍪ rarg
larg ⍪[i] rarg   (where i is ρρrarg and ⎕io is 1)

Note 2: res ⍪← rarg   res ← res ⍪ rarg
The syntax is similar to the in-place operations in other programming languages like C++ and C#. However, in APL64, besides the shorter notation, this can also provide significant performance gains particularly in cases that involve repetitive catentations with large arrays such as inside an iterative control structure (a :for loop).

⊂ Enclose

Syntax:

res ← ⊂ arg
res ← ⊂[i] arg

Description:
Create a nested scalar from any array that is not a simple scalar.
arg: any array.
i: non-negative integer scalar or vector that indicates the axes of the arguments to be enclosed.

When you use Enclose with axis, you can specify some or all of the axes of an array, yielding a result of lower rank than the right argument that contains the same data. Enclose with Axis is similar to ⎕split but is more general in function.

Example:

```
    C←'A' 'MM' 'SSS'
    ρC
3
   C[2]←⊂2 2 ρι4
    C
A   1 2  SSS
   3 4
    ]DISPLAY C
.→------------.
|   .→--.  .→--.|
|A ↓1  2|  |SSS||
|- |3  4|  '---'|
|   '~--'        |
'∊------------'
     A←3 4ρι12
    ⊂[1]A
 1 5 9  2 6 10  3 7 11  4 8 12
    ⊂[2]A
 1 2 3 4  5 6 7 8  9 10 11 12
    (⊂[1 2]A)≡⊂A
1
```

```
    (⊂[2 1]A)≡⊂⍉A
1
     B←2 3 4ρι24
    ⊂[1 3]B
 1  2  3  4    5  6  7  8    9 10 11 12
13 14 15 16   17 18 19 20   21 22 23 24

 C←⊂[1 2 3]B
    ρB
2 3 4
    ρC

    ≡B
1
    ≡C
2
```

Note 1: See the description of ⎕penclose for the Evolution Level 1 dyadic ⊂, Partitioned enclose.
Note 2: Partition (dyadic ⊂) is also in this section.

⊂ Partition

Syntax:

res ← larg ⊂ rarg
res ← larg ⊂[i] rarg

Description:
Create a nested array from portions of another array. (Evolution Level 2.)
larg: a scalar or vector of non-negative integers that specifies the partioning of rarg along the specified axis; the default axis is the last dimension. If larg is a vector, its length must equal the dimension of the selected axis of rarg. If an element of larg is zero, the corresponding element from rarg is not included; if a non-zero element of larg is greater than the element that precedes it, the function starts a new partition. If the element is less than or equal to its predecessor, the function nests the corresponding element with its predecessor or predecessors. A scalar larg is equivalent to a vector of the appropriate length with all elements having the same value.
rarg: any nonscalar array.
i: non-negative scalar that indicates the dimension desired.
res: a nested array of the elements of rarg, excluding those corresponding to zero elements of larg. The pattern of nesting is determined by elements of larg that are greater than the preceding element. The rank of res is the same as the rank of rarg, but it is nested one level deeper. The items of res are scalars or sub-vectors of the vectors of rarg along the selected axis.
Example:

```
     ]display (10ρ1 2 3)⊂ι10
.→---------------------------.
|.→..→..→--..→..→--..→..→---.|
||1||2||3 4||5||6 7||8||9 10||
|'~''~''~--''~''~--''~''~---'|
'∊---------------------------'

   3 2ρι6
1 2
3 4
5 6

   2 2 3 ⊂[1] 3 2ρι6
.→----------.
↓.→--..→--.|
||1  3||2  4||
|'~--''~--'|
|.→.   .→.   |
||5|   |6|   |
|'~'   '~'   |
'∊----------
```

Note 1: See the description of ⎕penclose in the System Functions manual for the Evolution Level 1 dyadic ⊂, Partitioned enclose. This system function uses a different format in the left argument to produce a similar result.
Note 2: Enclose (monadic ⊂) is also in this section.

⊃ Disclose

Syntax:

res ← ⊃ arg
res ← ⊃[i] arg

Description:
Reduce a level of nesting from an array, raising its rank. (Evolution Level 2:)
arg: any array.
i: non-negative integer scalar or vector that specifies the position in (ρres) where the enclosed dimensions of the items of arg are to be placed. The number of elements in i must match the ranks of all the nonscalar enclosed items in arg. The system pads individual items of arg to the same shape before disclosing them.
res: if arg is a nested scalar, it is expanded back to its enclosed array. Each nonscalar item of arg must have the same rank; a scalar item becomes the first item of an array of the same rank as the nonscalar items. Disclose pads the nested arrays along each dimension to conform to the item being disclosed that is largest along that dimension.

Disclose returns an array whose rank is one more than the common nonscalar rank. When no axis is specified, the first dimension has a length equal to the number of items, and the dimensions of the items of arg become the remaining dimensions of res in the same order.

For example, Disclose turns a nested vector of vectors into a matrix; each item becomes one row in the matrix. If arg is a nested vector of matrices, res is a three- dimensional array with as many planes as there are items; each plane contains as many rows as the nested matrix with the most rows and as many columns as the nested matrix with the most columns.

When you use Disclose with axis, you specify the axis along which to build an array from the items of the argument; arg is a nested array whose items are scalars or arrays of the same rank.

Example:

```
    A←(1 2 3) (2 4 6) (3 6 9)
    ]DISPLAY A
+→--------------------+
|+→----++→----++→----+|
||1  2  3||2  4  6||3  6  9||
|+~----++~----++~----+|
+ϵ--------------------+

    ⊃A
 1 2 3
 2 4 6
 3 6 9

    ⊃(1)(1 2)(1 2 3)
1 0 0
1 2 0
1 2 3

    ⊃[1]'MARY' (ι4) 'STEVE'
M 1 S
A 2 T
R 3 E
Y 4 V
  0 E
    ⊃[1 3](2 5ριι10)(3 4ρ10×ι12)
  1   2   3   4  5
 10  20  30  40  0

  6   7   8   9 10
 50  60  70  80  0

  0   0   0   0  0
 90 100 110 120  0

    ρ⊃[2 3]M←2 3ρ⊂4 5ρι20
2 4 5 3
    ρ⊃[1 4]M
4 2 3 5
    ρ⊃[4 1]M
5 2 3 4
```

Note 1: At Evolution Level 1, monadic Disclose specifies the First function. See the descriptions of the First function (↑) in this section and ⎕first in the System Functions manual.

Note 2: Pick (dyadic ⊃) is also in this section.

⊃ Pick

Syntax:

res ← larg ⊃ arg

Description:

Select a portion of an array.

rarg: any array.

larg: positive integers that describe (left to right) how deep into arg to go to select an item.

res: a subset of arg specified by larg. (Can be used in selective assignment.)

ext: ⎕io.

Example:

```
    A←'ONE' (2 2ρι4) 'SIX'
    ]DISPLAY A
.→----------------.
|.→--. .→--. .→--.|
||ONE| ↓1 2| |SIX||
|'---' |3 4| '---'|
|     '∼--'      |
'∊----------------'
    ρA
3
    2⊃A
1 2
3 4
    3 2⊃A
I
    (2 (2 1))⊃A
3
```

↑ First

Syntax:

res← ↑ rarg

Description:

Return the first item of an array. (Evolution Level 2.)

rarg: any array.

res: if rarg is a nested vector, the first item is selected and expanded into an array. If rarg is empty, res is the prototype of rarg. (See the technical note under the description of Reshape, below.)

Example:

```
    C←'ONE' (2 3 4 5)
    ↑C
ONE
    ρ↑C
3
```

```
 D←0 3ρ ⊂ 2 3ρι6
     ↑D
0 0 0
0 0 0
```

Note 1: At Evolution Level 1, the monadic up arrow specifies the Mix function. See the description of ⎕mix in the System Functions manual.
Note 2: Take (dyadic ↑) is described below.

↑ Take

Syntax:

res ← larg ↑ rarg
res ← larg ↑[i] rarg

Description:
Select a set of elements from an array.
When you use Take with axis ([i]), you can specify some or all of the axes of rarg; the number of elements of larg must equal the number of axes specified in i.
larg: any integer scalar or vector with one element per dimension of rarg.
rarg: any array.
i: integer scalar or vector specifying the axes of rarg along which to select items; the values in i must be unique and less than or equal to the rank of rarg.
res: the subset of rarg elements. larg specifies the shape of res. If larg is negative, the selection starts from the end rather than the beginning; if larg specifies an array larger than rarg, res is padded with the fill item (⎕type ↑ arg; blank or 0 for simple arrays). (Can be used in selective assignment.) The shape along axes not specified in i remains unchanged.

Example:

```
     2↑3 6 2
3 6
     5↑3 6 2
3 6 2 0 0
     ¯3 2↑2 3ρ1 2 3 4 5 6
0 0
1 2
4 5

     B←2 2 3ρ 1 2 3 4 5 6 7 8 9 10 11 12
     2 1↑[3 2]B
 1 2

 7 8
```

↓ Drop

Syntax:

res ← larg ↓ rarg

res ← larg ↓[i] rarg

Description:

Exclude a set of elements from an array. When you use Drop with axis, you can specify some or all of the axes of rarg; the number of elements of larg must equal the number of axes specified in i.

larg: any integer scalar or vector with one element per dimension of rarg.

rarg: any array.

i: integer scalar or vector specifying the axes of rarg along which to drop elements; the values in i must be unique and less than or equal to the rank of rarg.

res: all the elements of rarg except the subset specified by larg. larg specifies the number of elements in each dimension to exclude (starting from the end if larg is negative). If an element of larg is larger in magnitude than the corresponding dimension of rarg, res is empty along that coordinate. (Can be used in selective assignment.) The shape along axes not specified in i remains unchanged.

Example:

```
    5↓1 3 2 7 4 8
8
    A←2 3⍴1 2 3 4 5 6
    0 ¯1↓A
1 2
4 5

    B←2 2 3⍴ 1 2 3 4 5 6 7 8 9 10 11 12
    2 1↓[3 2]B
 6
```

∪ Unique

Syntax:

res ← ∪ arg

Description:

Select the unique elements of a vector.

arg: any vector.

res: a compression of arg with all but the first instance of each distinct element removed.

ext: ⎕ct.

Example:

```
    ∪ 'abracadabra'
abrcd
    ∪ 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3
3 1 4 5 9 2 6 8 7
    ∪ 'George' 'John' 'Thomas' 'James' 'James' 'John' 'Andrew'
George John Thomas James Andrew
```

ρ Shape

Syntax:

res ← ρ arg

Description:

Return the shape of an array.
arg: any array.
res: a vector containing the length of each dimension of arg.

Example:

```
    ρ 2 3 5
3
    ρ 2 3 5 ρι30
2 3 5
    ρ 99

    ρρ 99
```

ρ Reshape

Syntax:

res ← larg ρ rarg

Description:

Create an array of specific shape and content.
larg: non-negative integer scalar or vector.
rarg: any array.
res: the elements of rarg selected in ravel order and formed into the new shape specified by larg. Some rarg elements may be lost (res will have fewer elements than rarg) or duplicated (res will have more elements than rarg) as needed. (Can be used in selective assignment.)

Example:

```
    3 ρ 99
99 99 99
    2 4 ρ 2 3 5
2 3 5 2
3 5 2 3
    2 3ρ1 1 2ρ7 8
7 8 7
8 7 8
```

Note: For monadic ρ, see Shape described above..

Technical Note: Empty Arrays, Type, Prototype, and Fill Items

If you use zero as the left argument to Reshape, the result is an empty array. An empty array has no elements, but it does have a structure whose shape may be complex and deeply nested. To understand the structure, you must know that non-empty arrays have a "type." The type of an array has the same shape and depth as the array with numbers replaced by zeroes and characters replaced by spaces. You can generate the type of an array using the system function ⎕type; since it may contain spaces, it is useful to use the]display user command to understand the structure.

The prototype of an array is the type of its first item (where item is one-level less nested than element). The prototype is the value that is enclosed and stored, nested, in an empty array, and it is this prototype that gives the empty array its structure. (Note that empty arrays can require a great deal of memory.)

The same value is used as the fill item in certain functions, for example, when you Overtake; that is, when you use the Take primitive function with a left argument that is greater than the length of the right argument. However, note that the empty array you create with 0 ρ vector does not match the value of the last element of a vector created with overtake. However, if you apply the First primitive function to the empty array, that result matches the last item of the same vector. Thus, in a somewhat circular definition, fill item is defined in language terms as ↑0 ρ arg .

```
    ρvec
2
    over←3↑vec
    over[3]≡⎕type vec[1]
1
    (0ρvec)≡⎕type vec[1]
0
    (↑0ρvec)≡⎕type ↑vec
1
    (↑ ⎕type vec) ≡ ⎕type ↑vec
1
```

If you create empty arrays of greater rank, such as by 0 3 ρ vec, there is only one prototype. The exact content of the prototype will be of importance only when you materialize it, such as by using Overtake. The shape of the empty array may be extremely important for it to be conforming.

⍉ Transpose

Syntax:

res ← ⍉ arg

Description:

Reverse the axes of an array.

arg: any array.

res: arg with dimensions interchanged. (Can be used in selective assignment.)

Example:

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

```
    ρ⍉B
4 3
```

⊖ Horizontal Rotate

Monadic (Reverse)

Syntax:

    res ← ⊖ arg
    res ← ⊖[i] arg

Description:

Reverse the order of the elements of an array array along an axis specified by the index to the shape vector. The default dimension for ⊖ is the first dimension.
arg: any array.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the elements in arg reversed along the ith dimension. (Can be used in selective assignment.)
ext: ⎕io.

Example:

```
    ⊖ 'RATS LIVE ON'
NO EVIL STAR

    A←3 3ρ'ABCDEFGHI'
    ⊖A
GHI
DEF
ABC
```

Dyadic (Rotate)

Syntax:

    res ← larg ⊖ rarg
    res ← larg ⊖[i] rarg

Description:

Shift the elements of rarg toward the beginning (or end) of one dimension maintaining the same relative order (as if on a chain). When an element is displaced from the beginning, it rotates to become the last element of the dimension; when an element is displaced from the end, it rotates to become the first element of the dimension. The index to the shape vector specifies the dimension along which the elements rotate. The default dimension for ⊖ is the first dimension. larg specifies the direction and the number of positions. If larg is positive, elements shift toward the beginning; if larg is negative, the elements shift toward the end. The magnitude of larg specifies the number of positions to shift.
larg: integer scalar or vector of length equal to the chosen dimension of rarg.
rarg: any array.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the elements in rarg rotated larg places along the ith dimension. (Can be used in selective assignment.)
ext: ⎕io (for dimension).

Example:

```
    2 ⊖ 'TODAY'
DAYTO
    B←3 4ρι12
    B
1  2  3  4
5  6  7  8
9 10 11 12

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

Note: larg ⊖ rarg <-> larg ⌽[⎕io] rarg
larg ⌽ rarg <-> larg ⊖[i] rarg (where i is ρρrarg and ⎕io is 1)

⌽ Vertical Rotate

Monadic (Reverse)

Syntax:

res ← ⌽ arg
res ← ⌽[i] arg

Description:

Reverse the order of the elements of an array array along an axis specified by the index to the shape vector. The default dimension for ⌽ is the last dimension.
arg: any array.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the elements in arg reversed along the ith dimension. (Can be used in selective assignment.)
ext: ⎕io.

Example:

```
    ⌽ 'RATS LIVE ON'
NO EVIL STAR
    A←3 3ρ'ABCDEFGHI'
    ⌽A
CBA
FED
IHG
    ⌽[1]A
GHI
DEF
ABC
```

Note: ⊖ arg <-> ⌽[⎕io] arg
⌽ arg <-> ⊖[i] arg (where i is ρρrarg and ⎕io is 1)

Dyadic (Rotate)

Syntax:

res ← larg ⌽ rarg
res ← larg ⌽[i] rarg

Description:
Shift the elements of rarg toward the beginning (or end) of one dimension maintaining the same relative order (as if on a chain). When an element is displaced from the beginning, it rotates to become the last element of the dimension; when an element is displaced from the end, it rotates to become the first element of the dimension. The index to the shape vector specifies the dimension along which the elements rotate. The default dimension for ⌽ is the last dimension. larg specifies the direction and the number of positions. If larg is positive, elements shift toward the beginning; if larg is negative, the elements shift toward the end. The magnitude of larg specifies the number of positions to shift.
larg: integer scalar or vector of length equal to the chosen dimension of rarg.
rarg: any array.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the elements in rarg rotated larg places along the ith dimension. (Can be used in selective assignment.)
ext: ⎕io (for dimension).

Example:

```
     2 ⌽ 'TODAY'
DAYTO
     B←3 4⍴⍳12
     B
1  2  3  4
5  6  7  8
9 10 11 12

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

Note: larg ⊖ rarg ←→ larg ⌽[⎕io] rarg
larg ⌽ rarg ←→ larg ⊖[i] rarg (where i is ⍴⍴rarg and ⎕io is 1)

⌷ Indexing

Syntax:

res ← larg ⌷ rarg
res ← larg ⌷[i] rarg

Description:
Selects a subset of rarg by using the index values specified in larg along each axis.
rarg: any array
larg: a nested vector or scalar of depth two, or a simple vector or scalar, containing indices to the dimensions of rarg. The length of larg must equal the rank of rarg, except that if rarg is a vector, larg can be either a one-element vector or a scalar. The values of larg must be nonnegative integers less than or equal to the rank of rarg.
res: the elements of rarg specified by their positions in larg.   (Can be used in selective assignment.)
ext: □io.

Examples:

```
      v← 'a' 2 'c' 4 5
      3 ▯ v
c
      (⊂2 4) ▯ v  ⍝ Note that (ρ larg) = ρρ rarg
2 4
      v[2 4]
2 4

      ''≡(⊂ι0) ▯ v
1
      ⍬≡(⊂ι0) ▯ (,9)
1
      (ι0) ▯ 9
9

        m
11 12 13 14
21 22 23 24
31 32 33 34

      (2 3) ▯ m
23
      (2 3) (1 4) ▯ m
21 24
31 34
      (3 2) 1 ▯ m
31 21
      (3 2) (,1) ▯ m
31
21
      ρ(ι0) (ι0) ▯ m
0 0
```

Description:
If you use the indexing function with axis, it is similar to bracket indexing when you omit the index for

one or more axes. In this case, the length of the left argument must equal the number of axes you specify. (Can be used in selective assignment.)

```
    three_D_array
 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 ⌷[2] three_D_array
 1  2  3  4
13 14 15 16

    three_D_array[;1;]
 1  2  3  4
13 14 15 16

    (1 2) (3 4) ⌷[2 3] three_D_array
 3  4
 7  8

15 16
19 20
```

Functions that Return an Index

ι Iota (Index Generator)

Syntax:

res ← ι arg

Description:

Return a set of consecutive integers.

arg: positive integer scalar or array.

res: a vector of arg integers in the sequence ⎕io, ⎕io+1, ⎕io+2, ...

ext: ⎕io.

Example:

```
    ι5
1 2 3 4 5

    ι2 5 3
 1 1 1  1 1 2  1 1 3
 1 2 1  1 2 2  1 2 3
 1 3 1  1 3 2  1 3 3
 1 4 1  1 4 2  1 4 3
 1 5 1  1 5 2  1 5 3

 2 1 1  2 1 2  2 1 3
 2 2 1  2 2 2  2 2 3
 2 3 1  2 3 2  2 3 3
 2 4 1  2 4 2  2 4 3
 2 5 1  2 5 2  2 5 3

      A←2 3⍴⍳6
      A≡A[⍳⍴A]
1
      ⍳⍴A
 1 1  1 2  1 3
 2 1  2 2  2 3
```

Note: For dyadic ⍳, see Index of below.

⍳ Index of

Syntax:

res ← larg ⍳ rarg

Description:

Find the location of items in an array.

larg: any vector.

rarg: any array.

res: for each item of rarg, the corresponding element of res is the index of the item's first occurrence in larg. If larg does not contain the item, the res element is ⎕io+⍴larg.

ext: ⎕io, ⎕ct.

Example:

```
      A←3 4 7 3 8
      A ⍳ 7 4 3 12
3 2 1 6

      'ABCDEF' ⍳ 2 5⍴'DEAD BEEF'
4 5 1 4 7
2 5 5 6 7
```

⍋ Grade Up

Numeric

Syntax:

res ← ⍋ arg

Description:

Return the ascending sort order of a numeric array.

arg: any numeric nonscalar array.

res: a numeric vector containing the indices that arrange the elements of arg in ascending numeric order. The length of res is the same as the first dimension of arg. If arg is a vector, you can use res as a subscript vector. Duplicate values retain their original relative positions.

ext: ⎕io.
If arg is a matrix, the result is formed by considering one column at a time, working from left to right. Grade up sequences the leftmost column as a vector. If the vector has no duplicate values, its ordering becomes the result. If the vector has duplicate values, the elements from the next column to the right are used in an attempt to sequence the duplicates. This process continues until either all duplications are resolved or all columns are used.
Arguments of more than two dimensions are treated as matrices, retaining the original first dimension and combining all the other dimensions into a single second dimension. In effect, the argument is treated as being reshaped as follows:

    ((1↑ρarg) , ×/1↓ρarg) ρ arg

Example:

```
     A←5 2 8
     ⍋A
2 1 3
     A[⍋A]
2 5 8
     Y← 3 4ρ 1 12 25 6 1 15 11 7 1 12 25 5
     Y
1 12 25 6
1 15 11 7
1 12 25 5
     ⍋Y
3 1 2
```

Character or String

Syntax:

res ← larg ⍋ rarg

Description:

Return the ascending sort order of a character array or string elements.
larg, rarg: any character nonscalar arrays; larg is ⎕av by default.
res: a numeric vector containing the indices that arrange the elements of rarg in ascending order according to the collating sequence specified by larg. The length of res is the same as the first dimension of rarg. If rarg is a vector, you can use res as a subscript vector.
ext: ⎕io.

Character grade up associates a numeric value with each character in the right argument based on the elements of larg. The rules of Numeric grade up are then applied to the associated numeric values to produce the result. If the left argument is a vector, then the associated numeric values are equivalent to those produced by dyadic iota. Specifically, V ⍋ A is equivalent to ⍋V⍳A.

For left arguments of rank 2 or greater, each dimension is used independently, working from the last to the first. The numeric ordering value for a character of rarg is the lowest coordinate index along the specified dimension of an occurrence of the character in larg. If the character does not appear in larg, its ordering value is determined like that of dyadic iota. Thus, the initial ordering value of a character is the same as the first column of larg in which it appears. If this ordering contains no duplications, it is used. If duplicates exist, their ordering values are sequenced with respect to the next dimension. This process continues until all duplications are resolved or all dimensions of the left argument are exhausted.

If the following matrix is the left argument

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

the initial ordering using the last dimension results in A and a coming before B and b, and so on. If both A and a are in the right argument, they are duplicates since they have identical coordinate values along the last dimension. A second evaluation, using the first dimension, places A before a.

Example:

```
    'ABC' ⍋ 'CAB'
2 3 1
    A←3 4⍴'FOURFIVESIX '
    A
FOUR
FIVE
SIX
     ⍋A
2 1 3
    A[⍋A;]
FIVE
FOUR
SIX
```

The example below uses three collating sequences (each starting with a blank) to produce the results shown in the table below.

Collating Sequence 1: abcdefABCDEF

Collating Sequence 2: aAbBcCdDeEF

Collating Sequence 3:   abcdefg

Sorting with Collating Sequences

Original Data	Sort with Collating Sequence 1	Sort with Collating Sequence 2	Sort with Collating Sequence 3
Aba	aar	aar	aar
aba	aba	aba	aba
Deaf	abba	abba	Aba
babe	babe	Aba	ABA
deed	bead	ABA	abba
CCC	bA	babe	bA
Deed	deaf	bA	babe
deaf	deed	bead	Bach
bA	Aba	Bach	bead
bead	ABA	CCC	CCC
abba	Bach	deaf	deaf
ABA	CCC	deed	Deaf
Bach	Deaf	Deaf	deed
aar	Deed	Deed	Deed

⍒ Grade Down

Numeric

Syntax:

res ← ⍒ arg

Description:
Return the descending sort order of a numeric array.
arg: any numeric nonscalar array.
res: a numeric vector containing the indices that arrange the elements of arg in descending numeric order. The length of res is the same as the first dimension of arg. If arg is a vector, you can use res as a subscript vector. Duplicate values retain their original relative positions. See the description of Numeric grade up in this chapter for more information.
ext: ⎕io.

Example:

```
    A←37 9 18
    ⍒A
1 3 2
    A[⍒A]
37 18 9
```

Character or String

Syntax:

res ← larg ⍒ rarg

Description:
Return the descending sort order of character array or string elements.
larg, rarg: any character nonscalar arrays; larg is ⎕av by default.
res: a numeric vector containing the indices that arrange the elements of rarg in descending order according to the collating sequence specified by larg. The length of res is the same as the first dimension of rarg. If rarg is a vector, you can use res as a subscript vector. See the description of Numeric grade down in this chapter for more information.
ext: ⎕io.

Example:

```
    ⍒'CAB'
1 3 2
```

Miscellaneous Functions

⍕ Format

Syntax: res ← ⍕ arg

Description:
Represent numeric data in character form.
arg: any array.
res: arg represented in character form.
ext: ⎕pp.

Example:

```
    ⍕ 2 3⍴1 2 3 4 5 6
 1 2 3
 4 5 6
    ⍴ ⍕ 2 3⍴1 2 3 4 5 6
2 6
    'REDUNDANT'≡⍕'REDUNDANT'
1
```

⍕ Pattern Format

Syntax:
res ← larg ⍕ rarg

Description:
Represent numeric data in character form, formatting the result.
larg: integer scalar or vector of pairs; a single pair is replicated as with scalar extension. The first number of each pair specifies the field width for the column; 0 requests a field large enough to accommodate the largest number. The second number specifies the number of decimal places. If the second number is negative, the result is formatted in exponential notation. A pair of numbers for each column specifies different formatting for each column. If only one number is specified, it is assumed to

be the number of decimal places.
rarg: any numeric array.
res: a character representation of rarg formatted as specified by larg.

Example:

```
     1 0 ⍕ 2 3 5
235
     1 ⍕ 2 3 5
 2.0 3.0 5.0
     1 0 4 1 6 2 ⍕ 2 3⍴⍳6
1 2.0  3.00
4 5.0  6.00
```

⍎ *Execute*

Syntax:

⍎ arg
res ← ⍎ arg

Description:
Execute a character string representation of an APL expression.
arg: character scalar or vector.
res: the result, if any, generated by executing the expression.

Note: The APL64 execute primitive function executes the character string representation of an APL expression including new lines (⎕TCNLs) while APL+Win stops evaluation of the character string at the first ⎕TCNL.

Example:

```
     ⍎ '2+3'
5
     N←7
     'V',(⍕?N),'←10×⍳',⍕N
V7←10×⍳7  ⍝ (string is displayed)

     V7
VALUE ERROR
     V7
     ^
     ⍎'V',(⍕N),'←10×⍳',⍕N
 ⍝  (string is executed; no visible output)

     V7
10 20 30 40 50 60 70
```

Operators

¨ Each

Syntax:

```
res ←       f ¨ arg
res ← larg f ¨ rarg
```

Description:
Apply a function to each item.
rarg: any array with items valid for f.
larg: optional; any array with items valid for f (conforming).
res: the collection of all results (each result is a single nested scalar) from applying f to each item of arg one at a time.

Examples:

```
    A←1 2 3 ρ¨ 4 5 6
    A
 4  5 5  6 6 6
    ρA
3
```

This example reads the first five components of a file.

```
    ⎕←TN←99 ,¨ ι5
99 1  99 2  99 3  99 4  99 5
    ρTN
5

    ]display TN
.→----------------------------.
|.→---..→---..→---..→---..→---.|
||99 1||99 2||99 3||99 4||99 5||
|'~---''~---''~---''~---''~---'|
'∊----------------------------'
    FILE←⎕fread¨TN
```

< Enstring

Syntax:

```
res ← < arg
```

Description:
Convert a character vector to a string
arg: any character array
res: a string

Note: For additional information, refer to the document **Help | APL Language | Using ⎕STRING**.

Example:

```
    ⎕dr < «APL64» ⍝ result is a string
164
    ρ < «APL64» ⍝ result is empty because a string is a scalar
```

> Destring

Syntax:

res ← > arg

Description:

Convert a string to a character array.

arg: a string

res: a character array

Note: For additional information, refer to the document **Help | APL Language | Using ⎕STRING**.

Example:

```
    ⎕dr > «APL64» ⍝ result is a string
162
    > «APL64» ⍝ result is a string
APL64
    ρ> «APL64» ⍝ result is a string
5
```

/ Reduction

⌿ Reduction

Syntax:

res ← f ⌿ arg

res ← f / arg

res ← f ⌿[i] arg or res ← f /[i] arg

Description:

Apply a specified function across an array, eliminating the *i*th dimension in the process. The default dimension for ⌿ is the first dimension; the default dimension for / is the last dimension.

arg: any array valid for f. If arg is a scalar, function f is not applied and res is arg.

i: non-negative, integer-valued scalar that indicates the dimension desired.

res: the function f is applied progressively across the array, reducing the number of dimensions.

ext: ⎕io (for dimension).

For the ⌿ operator:

res[1]←arg[1;1] f arg[2;1] . . . f arg[n;1]

res[2]←arg[1;2] f arg[2;2] . . . f arg[n;2]

res[3]←arg[1;3] f arg[2;3] . . . f arg[n;3]

.

.

.

res[m]←arg[1;m] f arg[2;m] . . . f arg[n;m]

For the / operator:

res[1]←arg[1;1] f arg[1;2] . . . f arg[1;m]

res[2]←arg[2;1] f arg[2;2] . . . f arg[2;m]

res[3]←arg[3;1] f arg[3;2] . . . f arg[3;m]

.

.

.

res[n]←arg[n;1] f arg[n;2]. . .f arg[n;m]

Examples:

```
    +⌿ 2 3 5
10
    ,/'ABC' 'DEF' 'GHI'
 ABCDEFGHI

    A←2 3 ρ 1 2 3 4 5 6
    A
 1 2 3
 4 5 6
    ×⌿A
 4 10 18
    ×/A
6 120
    ×/[1]A
 4 10 18
    ×/[2]A
6 120
```

For functions other than scalar primitives, the general case of reduction is defined for vectors (recursively, for j in the range of ιρarg-1) as:

```
    res←⊂(⊃ arg[ρarg-j])f ⊃ f/¯j↑arg
    lc←'abcdefghijklmnopqrstuvwxyz'
    ~/'aeiou'  lc   'which vowels does this vector use?'
 eiou
    ⎕repl/ (6 5 4 3 2 1) (ι3) ('abc' 9 '^')
  abc abc abc abc abc abc 9 9 9 9 9 9 9 9 9 ^^^^^^
```

If the argument to Reduction is an empty array, the derived function is the identity function; that is, the function returns an array that is the mathematical right identity for the operand function. (Zero is the identity element for addition; 1 is the identity element for multiplication; etc.)

```
    ×/ 0 ρ vec
1
```

```
    ÷/ 3 0 ρ vec
1 1 1
```

Some functions do not have meaningful identities; for example ⍟/ ι0 generates a DOMAIN ERROR. Some possible identities are not presently implemented; for example, ,/ 0ρ⊂2 3ρι6 generates a NONCE ERROR. Since Reduction is inherently part of Outer Product (see below), identity functions are also relevant to that operator.

\ Expand

⍀ Expand

Syntax:

res ← oparray ⍀ arg
res ← oparray \ arg
res ← oparray ⍀[i] arg or res ← oparray \[i] arg

Description:
Expand arg by inserting fill items in a pattern specified by oparray. The fill items become new elements of the dimension specified by the index to the shape vector. The default dimension for ⍀ is the first dimension; the default dimension for \ is the last dimension.
oparray: Boolean vector whose sum equals the length of the chosen dimension of arg.
arg: any array; if the chosen dimension has length 1, arg is extended along the expansion axis to the number of positive elements in the operand (+/oparray). Note that this allows you to generate only fill elements for an operand consisting of zeroes and an argument that is a scalar or has length 1 along the chosen dimension.
i: non-negative, integer-valued scalar that indicates the dimension desired.
res: the array arg expanded by adding an additional blank or zero for each corresponding 0 in oparray. (Can be used in selective assignment.)
ext: ⎕io (for dimension).

Examples:

```
    0 0 1 0 1 \ 7 8
0 0 7 0 8
    1 0 1 ⍀ 2 3ρι6
1 2 3
0 0 0
4 5 6
    1 0 1 \ 4
4 0 4
    1 0 1 ⍀ 1 3ρι3
1 2 3
0 0 0
1 2 3
    0 0 \ 5
0 0
    0 0 ⍀ 1 4ρι3
 0 0 0 0
 0 0 0 0

    A←2 3ρ'ABCDEF'
    A
ABC
DEF
    1 0 1 ⍀ A
ABC

DEF
    1 0 0 1 0 1\A
A  B C
D  E F
```

Note 1: oparray ⍀ arg   oparray \[⎕io] arg
        oparray \ arg   oparray ⍀[i] arg   (where i is ρρarg and ⎕io is 1)
Note 2: See also Scan (\) in this section .
            See also ⎕expand in the System Functions manual.

. Inner Product

Syntax:

        res ← larg f.g rarg

Description:
Generalized matrix multiplication.
larg, rarg: conforming arrays valid for f and g where the last dimension of larg is equal to the first dimension of rarg.

res: applying function g between the elements of the last dimension of larg and the corresponding elements of the first dimension of rarg followed by reducing the result using function f. The shape of res is ((¯1↓ρlarg) , 1↓ρrarg). If larg is n by k, and rarg is k by m, then res is n by m:

```
res[1;1]←f/larg[1;] g rarg[;1]
res[1;2]←f/larg[1;] g rarg[;2]
.
.
res[1;m]←f/larg[1;] g rarg[;m]
res[2;1]←f/larg[2;] g rarg[;1]
.
.
.
res[n;1]←f/larg[n;] g rarg[;1]
.
.
res[n;m]←f/larg[n;] g rarg[;m]
```
For vectors larg and rarg: res ← f / larg g rarg

Examples:

```
     2 3 5 +.× 2 3 5
38
     'SPORT'+.='SHOUT'
3
    (3 3ρ'ABCDEFGHI')^.='DEF'
0 1 0

     M←2 3ρι6 ⋄ N←3 4ρι12
     M+.×N (matrix multiplication)
 38 44  50  56
 83 98 113 128
     N^.=⍉N
1 0 0
0 1 0
0 0 1
     'WHEAT GROATS'+.∊'AEIOU'
4
```

∘. Outer Product

Syntax:

res ← larg ∘.g rarg

Description:

Apply a function between every possible pairing of items taking one from each of two arrays.
larg, rarg: any arrays valid for g.
res: if g produces a result, res is an array of size ((ρ larg) , ρ rarg) that consists of the result of applying g between each combination of larg and rarg items. If g does not produce a result, then ∘.g does not return a result.

Examples:

```
     2 3 5 ∘.* 0 1 2 3
1  2   4   8
1  3   9  27
1  5  25 125

     1 2 3 4 5 ∘.- 1 2 3 4 5
1 2 3 4 5
2 2 3 4 5
3 3 3 4 5
4 4 4 4 5
5 5 5 5 5

     'ABC' ∘.= 'ABC'
1 0 0
0 1 0
0 0 1
     'ABC' ∘., '01'
```

```
A0 A1
B0 B1
C0 C1
```

. Inner & Outer Product Operator Delimiter

Inner & Outer Product Operator Delimiter Syntax:

Inner & Outer Product Operator Delimiter Description:

Inner & Outer Product Operator Delimiter Example:

Other Primitive Symbols

Symbols Associated with Values

← Assignment

Use: Assignment (simple, indexed, selective) or Sink

Syntax:
```
 name ← arg
name[idx1;idx2;...] ← arg
(exp) ← arg
      ← arg
```

Description:
Simple assignment: Store a value in a variable.
name: a variable name.
arg: any valid expression that returns a value.

Example:

```
   V ← ⍳5
   V
1 2 3 4 5
   NewName←V+2
   NewName
3 4 5 6 7
```

Description:

Indexed Assignment: Modify a subset of an array specified by an index enclosed in brackets. See the description of Brackets (Index into) in this chapter for information on specifying an index.
name: a variable name.
idx: integer values specifying the items of arg
arg: any valid expression that returns a value.
ext: ⎕io.

Example:

```
    V←2 3ρι6
    V
1 2 3
4 5 6
    V[2;2 3]←7 8
    V
1 2 3
4 7 8
```

Description:

Selective Assignment: Replace a portion of an array selected by an expression that consists of selection primitives and the name of the array.
exp: an expression that defines the portion of the array being replaced.
arg: value to be inserted in the selected portion of the array.

Example:

```
    V←ι6
    V
1 2 3 4 5 6
    (3⊃V)←999
    V
1 2 999 4 5 6
    L←'ABCDEFG'
    M←1 0 1
    (M/3↑L)←'XY'
XBYDEFG
```

To set every data item in a complex nested array to zero, use (∊A)←0; for example:

```
    A←'MARY' (2 3ρι6) 0 'JOE'
    ]DISPLAY A
.→------------------------.
| .→---. .→-----.   .→--. |
| |MARY| ↓ 1 2 3| 0 |JOE| |
| '----' | 4 5 6|   '---' |
|        '~-----'         |
'∊------------------------'

    (∊A)←0
    ]DISPLAY A
.→------------------------.
| .→---. .→-----.   .→--. |
| |0000| ↓ 0 0 0| 0 |000| |
| '----' | 0 0 0|   '---' |
|        '~-----'         |
'∊------------------------'
```

Description:

Sink: Suppress the output from any line of execution in immediate execution (session) and user defined function when executed monadically.
arg: any valid expression that returns a value.

Example:

```
      ← 2 3ρι6
```

¯ High Minus

Use: Negative decoration

Syntax: ¯n (no space)

Description:

The high minus sign placed immediately to the left of a numeric constant makes the number a negative number. This symbol is different from the Minus function; its use eliminates the ambiguities that arise when the same symbol has both meanings. **Note**: The APL64 numeric editor converts minus signs to high minus signs.

Example:

```
      ¯2 2
¯2 2
      -2 2
¯2 ¯2
      -¯2 2
2 ¯2
      ¯ 2
SYNTAX ERROR
```

() Parenthesis

Use: Parentheses have multiple uses as punctuation in APL expressions. They can change the order of execution from the simple right-to-left order that the interpreter would use in their absence; they are required at Evolution Level 2 for certain indexing and multiple assignment statements; and you can use parentheses to create nested vectors (or nested elements of arrays of higher ranks).

Syntax: expression (expression) expression
(variable_string) ← value_string
variable ← value (variable_string) value

Description:
Determine order of execution. When you enclose part of a statement within parentheses, the interpreter evaluates the parenthesized expression and determines a value, which it then evaluates within the full statement.

Example:

```
      19 - 3 + 6 × 2
4
      19 - (3 + 6) × 2
1
      19 (-3 + 6) × 2
38 ¯18
```

Description:
When you want to assign values to multiple variables or index into a strand, you must enclose the left argument or strand in parentheses. The same statements without parentheses generate an EVOLUTION ERROR at Evolution Level 2.
Example:

```
      (C D E) ←(ι3) (4×5 6 7) (8ρ9)
      E
9 9 9 9 9 9 9 9
      (3↑2↓E)←C
      E
9 9 1 2 3 9 9 9
      3↑2↓E←C
3 0 0
      (C D E)[2]
 20 24 28
```

Description:
You can create a nested vector by assigning an enclosed vector to a single element or a vector or enclosing a vector of values within parentheses as part of the assignment.

Example:

```
      A←1 3 5
      A
1 3 5
      A[2]←(⊂2 3 4)
      A
 1  2 3 4  5
      B←1 (2 3 4) 5
      A≡B
1
```

ϑ Zilde

Use: Empty vector

Syntax: ϑ or (ι0)

Description:
A constant that is an empty numeric vector. This is useful in any situation where you want to initialize something or compare something to an empty vector.

Example:

```
    A←ϑ
    ρρA
1
    ρA
0
    B←ι0
    A≡B
1
```

[] Bracket

Use: Index into

Syntax: res ←arg[idx1;idx2; …]
res ←arg[indexarray]

Description:
Select a subset of elements from an array.
arg: any nonscalar array.
index: the index can be either of two forms: a list of simple arrays (idxn) separated by semicolons, or a nested array of enclosed vectors (indexarray).

If the index is a list of simple arrays, it must have the same number of arrays as arg has dimensions; you separate the items with semicolons. Each idx value corresponds to one dimension of arg. An idx value can be an integer array of any shape; its elements specify the values of the dimension you want to select. The elements of idx cannot exceed the length of arg for that dimension. You can leave an index blank, using only the semicolon; in this case, you specify the entire dimension. When the index has the form idx1;...idxn, res is the portion of arg specified by the index arrays. The number of elements in the idx items determine the shape of arg. See the Introduction to the Language chapter in this manual for more examples of this form of indexing.

If the index is an array of enclosed vectors, indexarray can have any shape, but each item of indexarray must be a simple vector of length n, where n is ρρarg. Each item specifies one point in arg; each element of the simple vector specifies the value of the dimension that represents the point. (For example, if arg is a three-dimensional matrix, an item of indexarray would have three scalars, representing the plane, row and column of the point desired.)

When the index has the form indexarray, res is an array whose shape is identical to the shape of indexarray, and whose values represent the points specified.
ext: ⎕io.

Example:

```
    3 4 7 3 8 [3 2 1]
7 4 3
    'ABCD'[3 2]
CB
    (3 4ρι12)[;3]
3 7 11
    'ABC'[4 4ρι3]
ABCA
```

```
BCAB
CABC
ABCA
     (3 4ρι12)[(1 1)(2 2)(3 3)]
1 6 11
```

; Semicolon

Description:

The semicolon has multiple uses as a syntactic designator.

- As a language symbol, the semicolon separates indices (dimensions) in indexing or indexed assignment. For example, MA[2;3] returns the second row, third column of a matrix.
- In writing user-defined functions, you use semicolons in the header to separate variables that you want the system to localize.
- In formatting output, you use the semicolon to separate arguments to the ⎕fmt command. See the Formatting Data chapter in the System Functions manual.
- In contexts that are not specifically part of the language, you use the semicolon as syntactic punctuation to separate arguments, for example, in setting watchpoints for use in debugging.

⎕ Quad

Description:

The quad symbol as the first character of a word designates a system function, a system variable, or a system constant. You can use these functions and values in user-defined functions to perform many actions analogous to system commands. See the System Functions manual for more information.

The quad symbol can also be a utility that allows a user-defined function to request input. The system reads the input as an APL expression and evaluates it. If the expression generates an error, the system reports the error and repeats the prompt (⎕:). The system returns the input as an explicit result. You can also use ⎕ to display the contents of a variable or an expression. **Note**: Quad (⎕) input is available only in the APL64 Developer version which requires the Windows operating system environment.

Example:

You can use the system variable ⎕io to change the index origin from one to zero. Thereafter, the first two positions in an array are numbered 0 and 1 rather than 1 and 2.

Example:

```
     ⎕←X←ι5
1 2 3 4 5
     X
1 2 3 4 5
     ⎕←ι5
1 2 3 4 5
     (⎕←3+4) × ⎕←5+6
11
7
77
```

The following example shows how you might use Quad in a simple demonstration program. The first line is a line of code; the value 10 represents user input.

```
    'How many throws?' ⋄ X←?⎕ρ100 ⋄ X
How many throws?
⎕:
    10
14 76 46 54 22 5 68 68 94 39
```

The Interrupt Input choice on the Options menu enables you to interrupt a program that is requesting Quad (⎕) input.

⍞ Quote-Quad

Use: Character input or output

Description:
A utility that allows a user-defined function to display character output or request character input. If you use quote-quad by itself, the system returns the input as an explicit result. If you assign a value to quote-quad and then request input, the system displays the assigned value with no newline character. If you then use quote-quad to request input, the previous output acts as a prompt. The system returns the combined prompt plus user input as the result. See the description of ⎕pr in the System Functions manual for more information. **Note**: Quote-quad (⍞) input is available only in the APL64 Developer version which requires the Windows operating system environment.

Example:
In this example below, the user types the following phrase: this is what i typed.

```
    ∇ CHARINPUT
  [1]  ⍞←'TYPE HERE: '
  [2]  ZINPUT ← ⍞
  [3]  'This function is done.'
    ∇

    CHARINPUT
TYPE HERE: this is what i typed.
This function is done.
    ZINPUT
        this is what i typed.
```

Note: The Interrupt Input choice on the Options menu enables you to interrupt a program that is requesting Quote-quad (⍞) input.

: Colon (Control Structure Keyword Suffix)

Use:	Syntactic identifier (and argument separator)

Syntax :keyword

label:

: arg

Description:

The colon has multiple uses as an identification symbol.

When using control structures in user-defined functions, the colon is the first character of each keyword that is part of the control structure's outer syntax. See the Writing APL Functions chapter in this manual for more information.

Example:

```
:if X>10
  Y1←Y2+Y3
:elseif X<0
  Y2←Y1+Y3
:else
  Y3←Y1+Y2
:end
```

- In user defined functions, a name with a colon as the last character that appears at the start of a line is a label. You can use a label with the branch arrow to alter the flow of execution. See the example under right arrow, above.
- In user defined functions, a colon prefix followed by a space (colon-space) at the beginning of a statement will suppress the output on any line of execution including the :THEN expression in Inline Control Structures.
- In contexts that are not specifically part of the language, you use the colon as syntactic punctuation to separate arguments, for example, in setting watchpoints for use in debugging.

Symbols Associated with Functions

⍝ Comment Prefix

Use: Comment symbol

Description:
The system does not evaluate or execute anything on a line to the right of the lamp symbol.

Example:

```
    2 + 2 ⍝ 2 good to be true
4
    2 + 2 2 ⍝ good to be true
4 4
```

⋄ Diamond

Use: Statement separator

Description:
In user-defined functions, the diamond separates statements as though they were on separate lines. The system evaluates each APL statement from right to left, but the statements are executed in order from left to right, starting to the left of the first diamond. In the example below, the first line and its result are equivalent to the lines following them.

Example:

```
     B←A←7-3-2 ⋄ B←B+1 ⋄ A+B
13
    A←7-(3-2)
    B←A
    B←B+1
    A+B
13
```

→ Branch (GoTo)

Use: Branch

Description:
In user-defined functions, the branch arrow used with a label or a line number alters the sequence of execution by specifying a different line for the system to execute. If you use the branch arrow with an empty right argument, the system executes the next statement; that is, it does not alter the sequence of execution. Branching to zero or a nonexistent line number exits the function. You can also use the naked branch arrow (with no right argument) to return to immediate execution mode. See the Writing APL Functions chapter in this manual for more information. In immediate execution mode, a naked branch removes one level of immediate execution from the stack.

Example:

```
[0]  BRANCH X
[1]  →(X>10) (X<0) /HI LO
[2]  Y3←Y1+Y2 ⍝ Fall through
[3]  → 0       ⍝ Exit
[4]  HI:
[5]  Y1←Y2+Y3 ⍝ Got first label
[6]  → 0
[7]  LO:
[8]  Y2←Y1+Y3 ⍝ Got second label
```

Symbols that Denote an Action Word

) Right Parenthesis

Description:
The right parenthesis as the first character of a word designates a system command instead of an APL expression. System commands are instructions to APL64 rather than facilities of the language interpreter. The System Commands chapter in this manual describes these commands.

Example:
You can use the system command)si to display the state indicator.

] Right Bracket

Description:
The right bracket as the first character of an immediate execution input line designates a user command

function instead of an APL expression. The User Command Processor and the User Command Descriptions chapters in this manual describe these commands.

Example:
You can use the system command]display to display an array on the screen in a manner that shows its structure as well as the values in the array.

Other Syntax Symbols

" Character vector delimiter

Description:
Character string delimiter. When you specify character text, you can use the double quote just as you use a single quote to delimit the string. You can use pairs of double quotes inside double quotes. If you use double quotes outside, you do not have to double a single quote; similarly, you can use single quotes outside of double quotes. You can double either character to add a level of delimitation.

Example:

```
      T←"Joe's"
      T
Joe's
      'fmA.bnFoo' ⎕wi 'onClick' " 'fmA' ⎕wi 'caption' 'Joe''s quote "" ' "
```

' Character Vector Delimiter

Description:
Character string delimiter. When you specify character text, you use the single quote (also called apostrophe) to delimit the string. If you want an apostrophe in the string, use a pair together.

Example:

```
      T←'Joe''s'
      T
Joe's
```

_ Character in Object Name

Description:
Valid character for object names. When you name a variable or a function, you can use the underscore as you would a numeric character, in any position except the first in the name.

∇ Function definition Prefix & Suffix

Description:
The del has multiple uses as a syntactic designator.

- Tool argument: When defining a Tool for the APL64 window menu, you can use a del in the tool command. When the system executes the tool, it replaces the del with any text that is highlighted in the current window or with the object that is closest to the cursor in the current window if you did not highlight anything before invoking the tool.
- The del, following the lamp symbol (⍝∇), designates public comments.
- The del designates the beginning and end of a user-defined function.

- In APL64, type ∇ or)edit ∇ to open a function Edit window.

⍙ Character in Object Name

Description:

Valid character for object names. When you name a variable or a function, you can use the delta underscore as you would an alphabetic character, in any position in the name.

⍵ Omega

Description:

Tool argument: When defining a Tool for the APL64 window menu, you can use an omega in the tool command. When the system executes the tool, it waits for user input to replace the omega.

⍫ Locked Function Definition Prefix

Note: Deprecated in APL64

⊣ Reserved symbol

⊢ Reserved symbol

∩ Reserved symbol

⍺ Reserved symbol

Random Number Generator

Pseudo-Random Number Algorithms: APL64 Roll and Deal Functions

APL64 is based upon digital technology so truly random numbers cannot be generated in APL64. Even if truly random numbers could be generated in APL64, they might not be useful if an application system employing them needs to be tested since such a truly random number sequence could not be repeated to compare application results between two application system programming modifications.

Instead of generating truly random numbers, APL64, like other programming environments based on digital technology, uses an algorithm to generate pseudo-random numbers. These numbers are pseudo-random because they are generated by a deterministic process which will eventually repeat the sequence of such numbers. Different algorithms have a greater non-repeating sequence period and different algorithms provide for better uniformity of values over the selected target interval. APL64 ⎕rl options provide a means for an application programmer to cause the same sequence of pseudo-random numbers to be generated, which facilitates testing of application system modifications.

No single pseudo-random number generation algorithm is suitable in all situations, so APL64 provides for a selection of these algorithms. This historical pseudo-random number generation algorithm in APL64 remains available and requires no application-system source code modifications. However, by appropriate ⎕rl settings, the APL64 application programmer can select other algorithms. The documentation in this chapter provides information about the available algorithms and how they affect the APL64 roll and deal functions. More information is available about these algorithms on the Internet links provided.

Pseudo-Random Number Algorithms

The APL64 roll (monadic ?, e.g. ?100) and deal (dyadic ?, e.g. 10?100) functions employ pseudo-random number generation algorithms so that programmers can simulate random events or create cryptographic keys. These algorithms are fully determined by their seed value so that the sequence generated can be repeated by using the same seed value. The repeatable property of these algorithms is useful for testing applications involving the roll or deal functions, however this repeatable property is what makes them pseudo-random. The period of a pseudo-random number algorithm is the maximum length, among all possible seed values, of a sequence of values generated by the algorithm which is non-repeating.

Pseudo-random number generator algorithms are analyzed on the following bases:

- Period ‘length’ for specific seed values
- Correlation among successively generated values
- Distribution of generated values among the possible values
- Mathematical ‘complexity’ resulting in processing time variation

The underlying pseudo-random number generator algorithm is used to obtain a pseudo-random number in a specified range. This value is mapped to the integer range space of the APL64 roll function. For example, the range space of the pseudo-random number generator may be values in [0, 1], but the range space of the APL64 roll function is programmer determined, e.g. For ?10 the range space is [1, 10] in index origin 1.

A pseudo-random number generator seed may itself be generated by a seeding generator. This technique is deemed to ‘inject entropy’ into the resulting sequence of pseudo random numbers, e.g. Seed Sequence. Refer to http://www.cplusplus.com/reference/random/seed_seq/. For detailed information about the specification of the Seed Sequence utility, refer to Section 26.5.7.1 of the working draft of the Standard for Programming Language C++. This specification is available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf.

Mersenne Twister

Mersenne Twister algorithm provides fast generation of high-quality pseudorandom integers. The specific variant of Mersenne Twister implemented in APL64 is ‘MT19937’ as provided by the Microsoft operating system. It is based on the Mersenne prime 2^{19937}-1 and produces a sequence of 32-bit numbers with a state size of 19937 bits. Its predefined parameters are:

- 32: word size
- 624: state size
- 397: shift size
- 31: mask bits
- 0x9908b0df: xor mask
- 11:tempering shift parameter u
- 0xffffffff: tempering bitmask parameter d
- 7: tempering shift parameter s
- 0x9d2c5680: tempering bitmask parameter b
- 15: tempering shift parameter t
- 0xefc60000: tempering bitmask parameter c
- 18: tempering shift parameter l
- 1812433253: initialization multiplier

Range of values generated: [0, $2^{32} - 1$]
Default Seed: 5489
More information:

- http://en.wikipedia.org/wiki/Mersenne_twister
- http://www.cplusplus.com/reference/random/mt19937

Linear Congruential with Initial Multiplier 48271

Linear congruential algorithm yields a sequence of numbers computed with a discontinuous piecewise linear equation. The pre-existing version of this algorithm in APL64 uses zero as the linear constant term. The specific variant of Linear Congruential with Initial Multiplier 48271 implemented in APL64 is 'MINSTD_RAND' as provided by the Microsoft operating system. Its predefined parameters are:

- 48271: the multiplier
- 0: the increment
- 2147483647:the modulus

Range of values generated: [Min: 1, ($2^{31} - 1$) – 1]]
Default Seed: 1
Initial State: 48271
More information:

- http://en.wikipedia.org/wiki/Linear_congruential_generator
- http://www.cplusplus.com/reference/random/minstd_rand

Subtract with Carry

Subtract with carry is a lagged Fibonacci algorithm with a state sequence of integer elements, plus one carry value. This algorithm is a generalization of the L'Ecuyer RNG (University of Montreal). The specific version of Subtract with Carry implemented in APL64, as provided by the Microsoft operating system, is 'RANLUX24_BASE' which produces 24-bit numbers. Its predefined parameters are

- 24: number of bit in each word in the state sequence
- 10: number of elements between advances
- 24: value that determines the degree of recurrence in the generated series

Range of values generated: [0, $2^{24} - 1$]
Default Seed: 19780503
More information:

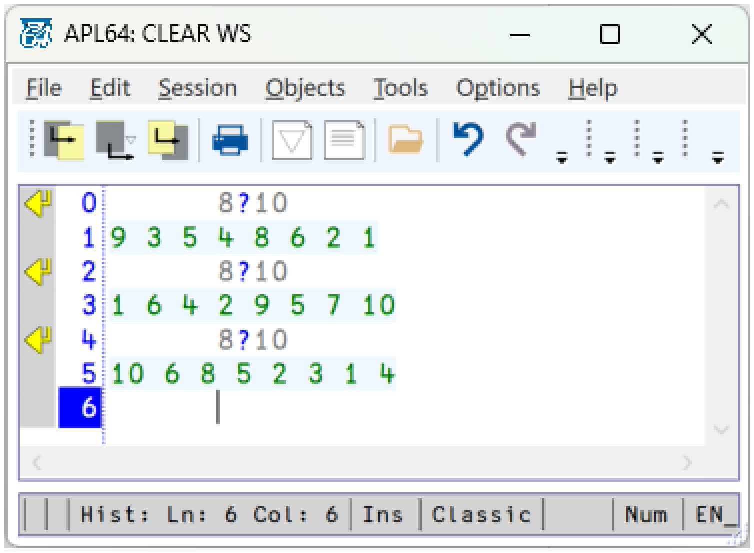
- http://en.wikipedia.org/wiki/Subtract_with_carry
- http://www.cplusplus.com/reference/random/ranlux24_base

Algorithmic Description of the APL64 Deal Function

Verbal description of the deal function

The APL64 deal function is a dyadic syntax function using the ? glyph. The result of L?R is a numeric vector of unique integers which have been pseudo-randomly selected from the range [⎕IO, ⎕IO+R-1]. L and R may be any positive integer scalar where L is not greater than R.

Examples of APL64 deal function



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

APL64: CLEAR WS
File Edit Session Objects Tools Options Help
Hist: Ln: 6 Col: 6 Ins Classic Num EN

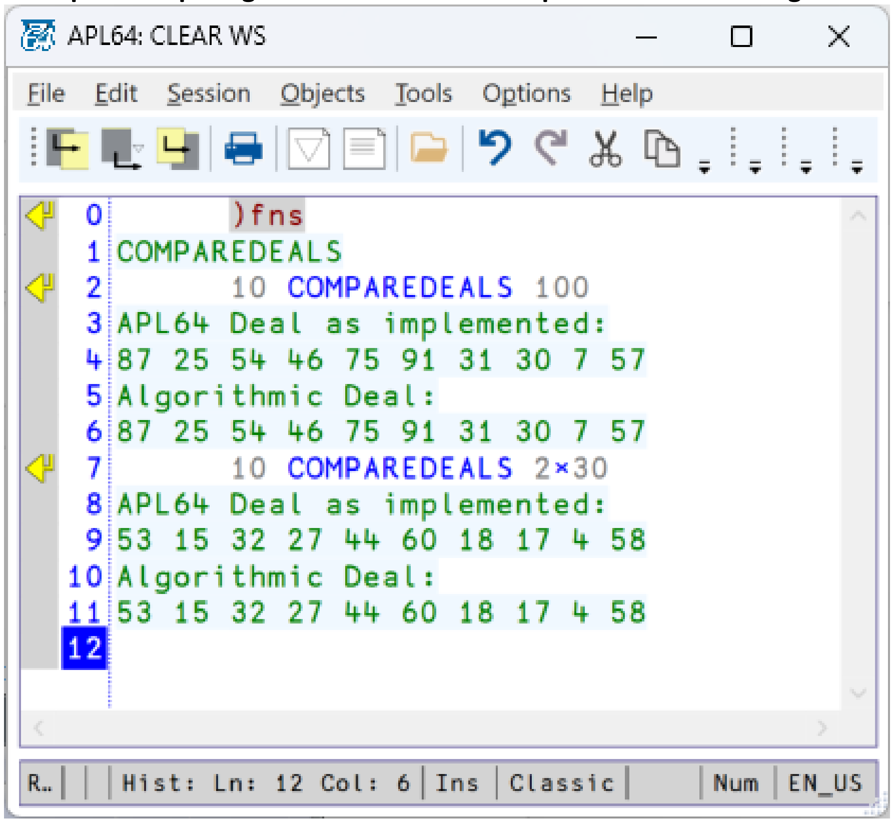
Algorithmic description of the deal function

Step 0: Assume index origin 1
```
    ⎕io←1
```
Step 1: Initialize a vector of integers from 1 to n
```
    LIST←ιn.
```
Step 2: Generate m pseudo-random integers
```
    R←¯1+(ιm)+?n+1-ιm
```
Step 3: In a loop for i∊ιm, swap the i-th and R[i]-th elements of LIST:
```
    :FOR i :IN ιm
        LIST[i,R[i]]←LIST[R[i],i]
    :ENDFOR
```
Step 4: Return the result of m?n as the first (n+1-m) elements of LIST
```
    n+1-m↑LIST
```

APL64 function to compare Implemented and Algorithmic Deal

```
m COMPAREDEALS n
⎕io←1
⎕rl←7*5
⎕←'APL64 Deal as implemented:'
⎕←m?n
⎕rl←7*5
LIST←ιn
R←¯1+(ιm)+?n+1-ιm
:FOR i :IN ιm
  LIST[i,R[i]]←LIST[R[i],i]
:ENDFOR
⎕←'Algorithmic Deal:'
⎕←n+1-m↑LIST
```

Examples comparing the deal function as implemented and as algorithmic


APL64: CLEAR WS
File Edit Session Objects Tools Options Help
R.. Hist: Ln: 12 Col: 6 Ins Classic Num EN_US

```
      )fns
COMPAREDEALS
      10 COMPAREDEALS 100
APL64 Deal as implemented:
87 25 54 46 75 91 31 30 7 57
Algorithmic Deal:
87 25 54 46 75 91 31 30 7 57
      10 COMPAREDEALS 2×30
APL64 Deal as implemented:
53 15 32 27 44 60 18 17 4 58
Algorithmic Deal:
53 15 32 27 44 60 18 17 4 58
```

Conclusions

- The deal function uses a 'shuffle algorithm'. For example see: http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle.
- The selection of the results of the deal function is pseudo-random because a digital computer cannot inherently generate random numbers, so the APL64 pseudo-random number generator is used. Refer to the documentation of ⎕RL for additional information as well as http://engineering.mit.edu/ask/can-computer-generate-truly-random-number.
- Via Step 2, for each i∊⍳m, R[i] is pseudo-randomly distributed between i and n.
- Via Step 3, the algorithm is valid because at for each i∊⍳m, an integer is selected with uniform probability from the set of integers from 1 to n that have not yet been selected.
- As the COMPAREDEALS functions illustrates, the actual implementation of the deal function in APL64 cannot implement the algorithmic description literally, but must instead use better memory management, create the elements of the vector R individually and use dynamic swapping to avoid instantiating the entire LIST and R vectors in memory as m can be equal to n and n can be as large as ¯2+2*31.

Credits:

Thanks to Bruce Levin, Ph.D. [Professor and Past Chair, Department of Biostatistics Columbia University, Mailman School of Public Health] for investigating the algorithmic description of the deal function in APL64.