DYALOG APL

The tool of thought for expert programming

Language Reference

Version 12.0

Version 12.0.3

First Edition August 2008

Overview

Contents

C H A P T E R 1

Introduction

Workspaces

APL expressions are evaluated within a workspace. The workspace may contain objects, namely operators, functions and variables defined by the user. APL expressions may include references to operators, functions and variables provided by APL. These objects do not reside in the workspace, but space is required for the actual process of evaluation to accommodate temporary data. During execution, APL records the state of execution through the STATE INDICATOR which is dynamically maintained until the process is complete. Space is also required to identify objects in the workspace in the SYMBOL TABLE. Maintenance of the symbol table is entirely dynamic. It grows and contracts according to the current workspace contents.

Workspaces may be explicitly saved with an identifying name. The workspace may subsequently be loaded, or objects may be selectively copied from a saved workspace into the current workspace.

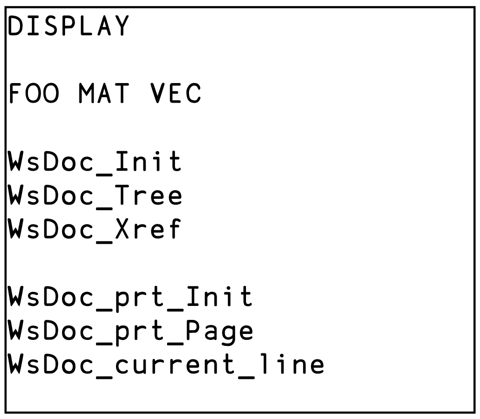
Under UNIX, workspace names must be valid file names, but are otherwise unrestricted. See your UNIX documentation for details.

Under Windows, Dyalog APL workspaces are stored in files with the suffix ".DWS". However, they are referred to from within APL by only the first part of the file name which must conform to Windows file naming rules.

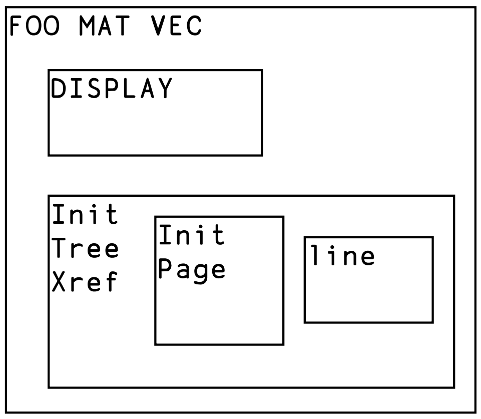
Namespaces

A namespace is a name class 9 object in Dyalog APL. Namespaces are analogous to nested workspaces.

'Flat' APL Workspace



Workspace with Namespaces



They provide the same sort of facility for workspaces as directories do for filing systems. The analogy might prove helpful:

Operation	MS-DOS	Namespace
`Create`	`MKDIR`	`)NS or ⎕NS`
`Change`	`CD`	`)CS`
`Relative name`	`DIR1\DIR2\FILE`	`NS1.NS2.OBJ`
`Absolute name`	`\DIR\FILE`	`#.NS.OBJ`
`Name separator`	`\`	`.`
`Top (Root) object`	`\`	`#`
`Parent object`	`..`	`##`

Major Benefits of Namespaces

Namespaces provide static (as opposed to dynamic) local names. This means that a defined function can use local variables and functions which persist when it exits and which are available next time it is called.

Just as with the provision of directories in a filing system, namespaces allow us to organise the workspace in a tidy fashion. This helps to promote an object oriented programming style.

APL's traditional name-clash problem is ameliorated in several ways.

- Workspaces can be arranged so that there are many fewer names at each namespace level. This means that when copying objects from saved workspaces there is a much reduced chance of a clash with existing names.

- Utility functions in a saved workspace may be coded as a single namespace and therefore on being copied into the active workspace consume only a single name. This avoids the complexity and expense of a solution which is sometimes used in 'flat' workspaces, where such utilities dynamically fix local functions on each call.

- In flat APL, workspace administration functions such as `WSDOC` must share names with their subject namespace. This leads to techniques for trying to avoid name clashes such as using obscure name prefixes like `'⍙⍙L1'`. This problem is now virtually eliminated because such a utility can operate exclusively in its own namespace.

The programming of GUI objects is considerably simplified.

- An object's callback functions may be localised in the namespace of the object itself.

- Static variables used by callback functions to maintain information between calls may be localised within the object.

This means that the object need use only a single name in its namespace.

Arrays

A Dyalog APL data structure is called an array. An array is a rectangular arrangement of items, each of which may be a single number, a single character, a namespace reference (ref), another array, or the ⎕OR of an object. An array which is part of another array is also known as a subarray.

An array has two properties; structure and data type. Structure is identified by rank, shape, and depth.

Rank

An array may have 0 or more axes or dimensions. The number of axes of an array is known as its rank. Dyalog APL supports arrays with a maximum of 15 axes.

- An array with 0 axes (rank 0) is called a scalar.
- An array with 1 axis (rank 1) is called a vector.
- An array with 2 axes (rank 2) is called a matrix.
- An array with more than 2 axes is called a multi-dimensional array.

Shape

Each axis of an array may contain zero or more items. The number of items along each axis of an array is called its shape. The shape of an array is itself a vector. Its first item is the length of the first axis, its second item the length of the second axis, and so on. An array whose length along one or more axes is zero, is called an empty array.

Depth

An array whose items are all simple scalars (i.e. single numbers, characters or refs) is called a simple array. If one or more items of an array is not a simple scalar (i.e. is another array, or a ⎕OR), the array is called a nested array. A nested array may contain items which are themselves nested arrays. The degree of nesting of an array is called its depth. A simple scalar has a depth of 0. A simple vector, matrix, or multi-dimensional array has depth 1. An array whose items are all depth 1 subarrays has depth 2; one whose items are all depth 2 subarrays has depth 3, and so forth.

Type

An array whose elements are all numeric, is called a numeric array; its TYPE is numeric. A character array is one in which all items are characters. An array whose items contain both numeric and character elements is of MIXED type.

Numbers

Numbers are entered or displayed in conventional or scaled decimal form. On entry, a decimal point is optional if there is no fractional part. On output, a number with no fractional part (an integer) is displayed without a decimal point.

The scaled form consists of:

a) an integer or decimal number called the mantissa,

b) the letter `E` or `e`,

c) an integer called the scale, or exponent.

The scale specifies the power of 10 by which the mantissa is to be multiplied.

Example

```
      12 23.24 23.0 2.145E2
12 23.24 23 214.5
```

Negative numbers are preceded by the high minus (`¯`) symbol, not to be confused with the minus (`-`) function. In scaled form, both the mantissa and the scale may be negative.

Example

```
      ¯22 2.145E¯2 ¯10.25
¯22 0.02145 ¯10.25
```

The empty vector (`⍳0`) may be represented by the numeric constant `⍬` called ZILDE.

Characters

Characters are entered within a pair of APL quotes. The surrounding APL quotes are not displayed on output. The APL quote character itself must be entered as a pair of APL quotes.

Examples

```
      'DYALOG APL'
DYALOG APL

      'I DON''T KNOW'
I DON'T KNOW

      '*'
*
```

Enclosed Elements

An array may be enclosed to form a scalar element through any of the following means:

- by the enclose function (⊂)
- by inclusion in vector notation
- as the result of certain functions when applied to arrays

Examples

```
      (⊂1 2 3),⊂'ABC'
 1 2 3  ABC

      (1 2 3) 'ABC'
 1 2 3  ABC

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

Legal Names

APL objects may be given names. A name may be any sequence of characters, starting with an alphabetic character, selected from the following:

```
0123456789 (but not as the 1st character in a name)
ABCDEFGHIJKLMNOPQRSTUVWXYZ_
abcdefghijklmnopqrstuvwxyz
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝß
àáâãäåæçèéêëìíîïðñòóôõöøùúûüþ
∆⍙
A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲
```

Note that using a standard Unicode font (rather than APL385 Unicode used in the table above), the last row above would appear as the circled alphabet, □ to □.

Examples

Legal	Illegal
`THIS∆IS∆A∆NAME`	`BAD NAME`
`X123`	`3+21`
`SALES`	`S!H|PRICE`
`pjb_1`	`1_pjb`

Specification of Variables

A variable is a named array. An undefined name or an existing variable may be assigned an array by specification with the left arrow (←).

Examples

```
      A←'CHIPS WITH EVERYTHING'

      A
CHIPS WITH EVERYTHING

      X Y←'ONE' 'TWO'

      X
ONE
      Y
TWO
```

Vector Notation

A series of two or more adjacent expressions results in a vector whose elements are the enclosed arrays resulting from each expression. This is known as VECTOR (or STRAND) NOTATION. Each expression in the series may consist of one of the following:

a) a single numeric value;

b) a single character, within a pair of quotes;

c) more than one character, within a pair of quotes;

d) the name of a variable;

e) the evaluated input symbol ⎕;

f) the quote-quad symbol ⍞;

g) the name of a niladic, defined function yielding a result;

h) any other APL expression which yields a result, within parentheses.

Examples

```
      ⍴A←2 4 10
3

      ⍴TEXT←'ONE' 'TWO'
2
```

Numbers and characters may be mixed:

```
      ⍴X←'THE ANSWER IS ' 10
2

      X[1]
 THE ANSWER IS

      X[2] + 32
42
```

Blanks, quotes or parentheses must separate adjacent items in vector notation. Redundant blanks and parentheses are permitted. In this manual, the symbol pair '←→' indicates the phrase 'is equivalent to'.

```
      1  2  ←→ (1)(2) ←→ 1  (2)  ←→ (1)  2
      2'X'3 ←→ 2 'X' 3 ←→ (2) ('X') (3)
      1  (2+2) ←→ (1) ((2+2)) ←→ ((1))  (2+2)
```

Vector notation may be used to define an item in vector notation:

```
       ρX ← 1 (2 3 4) ('THIS' 'AND' 'THAT')
3

      X[2]
 2 3 4

      X[3]
  THIS  AND  THAT
```

Expressions within parentheses are evaluated to produce an item in the vector:

```
      Y ← (2+2) 'IS' 4
      Y
4  IS  4
```

The following identity holds:

```
      A  B  C  ←→ (⊂A), (⊂B), ⊂C
```

Restructuring Arrays

A class of primitive functions re-structures arrays in some way. Arrays may be input only in scalar or vector form. Structural functions may produce arrays with a higher rank. The Structural functions are reshape (ρ), ravel, laminate and catenate (,), reversal and rotation (⌽), transpose (⍉), mix and take (↑), split and drop (↓), and enclose (⊂). These functions are described in *Chapter 4*.

Examples

```
      2 2ρ1 2 3 4
1 2
3 4

      2 2 4ρ'ABCDEFGHIJKLMNOP'
ABCD
EFGH

IJKL
MNOP
```

```
      ↓2 4⍴'COWSHENS'
 COWS  HENS
```

Display of Arrays

Simple scalars and vectors are displayed in a single line beginning at the left margin. A number is separated from the next adjacent element by a single space. The number of significant digits to be printed is determined by the system variable `⎕PP` whose default value is 10. The fractional part of the number will be rounded in the last digit if it cannot be represented within the print precision. Trailing zeros after a decimal point and leading zeros will not be printed. An integer number will display without a decimal point.

Examples

```
      0.1 1.0 1.12
0.1 1 1.12

      'A' 2 'B' 'C'
A 2 BC

      ÷3 2 6
0.3333333333 0.5 0.1666666667
```

If a number cannot be fully represented in `⎕PP` significant digits, or if the number requires more than five leading zeros after the decimal point, the number is represented in scaled form. The mantissa will display up to `⎕PP` significant digits, but trailing zeros will not be displayed.

Examples

```
      ⎕PP←3

      123 1234 12345 0.12345 0.00012345 0.00000012345
123 1.23E3 1.23E4 0.123 0.000123 1.23E¯7
```

Simple matrices are displayed in rectangular form, with one line per matrix row. All elements in a given column are displayed in the same format, but the format and width for each column is determined independently of other columns. A column is treated as numeric if it contains any numeric elements. The width of a numeric column is determined such that the decimal points (if any) are aligned; that the `E` characters for scaled formats are aligned, with trailing zeros added to the mantissae if necessary, and that integer forms are right-adjusted one place to the left of the decimal point column (if any). Numeric columns are right-justified; a column which contains no numeric elements is left-justified. Numeric columns are separated from their neighbours by a single column of blanks.

Examples

```
      2 4⍴'HANDFIST'
HAND
FIST

      1 2 3 ∘.× 6 2 5
 6 2  5
12 4 10
18 6 15

      2 3⍴2 4 6.1 8 10.24 12
2  4     6.1
8 10.24 12

      2 4⍴4 'A' 'B' 5 ¯0.000000003 'C' 'D' 123.56
 4E0  AB   5
¯3E¯9 CD 123.56
```

In the display of non-simple arrays, each element is displayed within a rectangle such that the rows and columns of the array are aligned. Simple items within the array are displayed as above. For non-simple items, this rule is applied recursively, with one space added on each side of the enclosed element for each level of nesting.

Examples

```
      ⍳3
1 2 3

      ⊂⍳3
 1 2 3

      ⊂⊂⍳3
  1 2 3

      ('ONE' 1) ('TWO' 2) ('THREE' 3) ('FOUR' 4)
  ONE  1   TWO  2   THREE  3   FOUR  4

      2 4⍴'ONE' 1 'TWO' 2 'THREE' 3 'FOUR' 4
 ONE    1  TWO   2
 THREE  3  FOUR  4
```

Multi-dimensional arrays are displayed in rectangular planes. Planes are separated by one blank line, and hyper-planes of higher dimensions are separated by increasing numbers of blank lines. In all other respects, multi-dimensional arrays are displayed in the same manner as matrices.

Examples

```
      2 3 4⍴⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

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

      3 1 1 3⍴'THEREDFOX'
THE


RED


FOX
```

The power of this form of display is made apparent when formatting reports.

Examples

```
      +AREAS←'West' 'Central' 'East'
 West  Central  East

      +PRODUCTS←'Biscuits' 'Cakes' 'Buns' 'Rolls'
 Biscuits  Cakes  Buns  Rolls

      SALES←50 5.25 75 250 20.15 900 500
      SALES,←80.98 650 1000 90.03 1200
      +SALES←4 3⍴SALES
  50  5.25   75
 250 20.15  900
 500 80.98  650
1000 90.03 1200

      ' ' PRODUCTS ⍪., AREAS SALES
            West  Central  East
 Biscuits     50     5.25    75
 Cakes       250    20.15   900
 Buns        500    80.98   650
 Rolls      1000    90.03  1200
```

If the display of an array is wider than the page width, as set by the system variable ⎕PW, it will be folded at or before ⎕PW and the folded portions indented six spaces. The display of a simple numeric or mixed array may be folded at a width less than ⎕PW so that individual numbers are not split across a page boundary.

Example

```
      ⎕PW←40

      ?3 20⍴100
54 22  5 68 68 94 39 52 84  4  6 53 68
85 53 10 66 42 71 92 77 27  5 74 33 64
66  8 64 89 28 44 77 48 24 28 36 17 49
       1  39  7 42 69 49 94
      76 100 37 25 99 73 76
      90  91  7 91 51 52 32
```

The Display Function

DISPLAY is a defined function distributed with Dyalog APL which may be used to illustrate the structure of an array. DISPLAY is monadic. Its result is a character matrix containing a pictorial representation of its argument. DISPLAY is used throughout this manual to illustrate examples. An array is illustrated with a series of boxes bordering each sub-array. Characters embedded in the border indicate rank and type information. The top and left borders contain symbols that indicate its rank. A symbol in the lower border indicates type. The symbols are defined as follows:

→	Vector.
↓	Matrix or higher rank array.
⊖	Empty along last axis.
⌽	Empty along other than last axis.
∊	Nested array.
~	Numeric data.
-	Character data.
+	Mixed character and numeric data.
∇	⎕OR object.
#	array of refs.

```
      DISPLAY 'ABC' (1 4⍴1 2 3 4)
.→-----------------.
| .→--.  .→------. |
| |ABC|  ↓1 2 3 4| |
| '---'  '~------' |
'∊-----------------'
```

Prototypes and Fill Items

Every array has an associated *prototype* which is derived from the array's first item.

If the first item is a number, the prototype is 0. Otherwise, if the first item is a character, the prototype is `' '` (space). Otherwise, if the first item is a (ref to) an instance of a Class, the prototype is a ref to that Class.

Otherwise (in the nested case, when the first item is other than a simple scalar), the prototype is defined recursively as the prototype of each of the array's first item.

Examples:

Array	Prototype
`1 2 3.4`	`0`
`2 3 5⍴'hello'`	`'hello'`
`99 'b' 66`	`0`
`(1 2)(3 4 5)`	`0 0`
`((1 2)3)(4 5 6)`	`(0 0)0`
`'hello' 'world'`	`'     '`
`⎕NEW MyClass`	`MyClass`
`(88(⎕NEW MyClass)'X')7`	`0 MyClass ' '`

Fill Items

Fill items for an overtake operation, are derived from the argument's prototype. For each `0` or `' '` in the prototype, there is a corresponding `0` or `' '` in the fill item and for each class reference in the prototype, there is a ref to a (newly constucted and distinct) instance of that class that is initialised by the niladic (default) constructor for that class, if defined.

Examples:

```
      4↑1 2
1 2 0 0
      4↑'ab'
ab  
      4↑(1 2)(3 4 5)
 1 2  3 4 5  0 0  0 0
      2↑⎕NEW MyClass
 #.[Instance of MyClass]  #.[Instance of MyClass]
```

In the last example, two distinct instances are constructed (the first by `⎕NEW` and the second by the overtake).

Fill items are used in a number of operations including:

First (`⊃` or `↑`) of an empty array
Fill-elements for overtake
For use with the Each operator on an empty array

Expressions

An expression is a sequence of one or more syntactic tokens which may be symbols or constants or names representing arrays (variables) or functions. An expression which produces an array is called an ARRAY EXPRESSION. An expression which produces a function is called a FUNCTION EXPRESSION. Some expressions do not produce a result.

An expression may be enclosed within parentheses.

Evaluation of an expression proceeds from right to left, unless modified by parentheses. If an entire expression results in an array that is not assigned to a name, then that array value is displayed. (Some system functions and defined functions return an array result only if the result is assigned to a name or if the result is the argument of a function or operator.)

Examples

```
      X←2×3-1

      2×3-1
4
      (2×3)-1
5
```

Either blanks or parentheses are required to separate constants, the names of variables, and the names of defined functions which are adjacent. Excessive blanks or sets of parentheses are redundant, but permitted. If `F` is a function, then:

```
      F 2↔ F(2) ↔ (F)2 ↔ (F) (2) ↔ F  (2) ↔ F ((2))
```

Blanks or parentheses are not needed to separate primitive functions from names or constants, but they are permitted:

```
      -2 ↔ (-)(2) ↔ (-) 2
```

Blanks or parentheses are not needed to separate operators from primitive functions, names or constants. They are permitted with the single exception that a dyadic operator must have its right argument available when encountered. The following syntactical forms are accepted:

```
      (+.×) ↔ (+).× ↔ +.(×)
```

The use of parentheses in the following examples is not accepted:

```
      +(.)×  or    (+.)×
```

Functions

A function is an operation which is performed on zero, one or two array arguments and may produce an array result. Three forms are permitted:

- NILADIC defined for no arguments
- MONADIC defined for a right but not a left argument
- DYADIC defined for a left and a right argument

The number of arguments is referred to as its VALENCE.

The name of a non-niladic function is AMBIVALENT; that is, it potentially represents both a monadic and a dyadic function, though it might not be defined for both. The usage in an expression is determined by syntactical context. If the usage is not defined an error results.

Functions have long SCOPE on the right; that is, the right argument of the function is the result of the entire expression to its right which must be an array. A dyadic function has short scope on the left; that is, the left argument of the function is the array immediately to its left. Left scope may be extended by enclosing an expression in parentheses whence the result must be an array.

For some functions, the explicit result is suppressed if it would otherwise be displayed on completion of evaluation of the expression. This applies on assignment to a variable name. It applies for certain system functions, and may also apply for defined functions.

Examples

```
      10×5-2×4
¯30
      2×4
8
      5-8
¯3
      10×¯3
¯30
      (10×5)-2×4
42
```

Defined Functions

Functions may be defined with the system function `⎕FX`, or with the function editor. A function consists of a HEADER which identifies the syntax of the function, and a BODY in which one or more APL statements are specified.

The header syntax identifies the function name, its (optional) result and its (optional) arguments. If a function is ambivalent, it is defined with two arguments but with the left argument within braces (`{}`). If an ambivalent function is called monadically, the left argument has no value inside the function. If the explicit result is to be suppressed for display purposes, the result is shown within braces. A function need not produce an explicit result. Refer to *Chapter 2* for further details.

Example

```
     ∇ R←{A} FOO B
[1]    R←⊃'MONADIC' 'DYADIC'[⎕IO+0≠⎕NC'A']
[2]  ∇

      FOO 1
MONADIC

      'X' FOO 'Y'
DYADIC
```

Functions may also be created by using assignment (`←`).

Function Assignment & Display

The result of a function-expression may be given a name. This is known as FUNCTION ASSIGNMENT (see also *Dynamic Functions*). If the result of a function-expression is not given a name, its value is displayed. This is termed FUNCTION DISPLAY.

Examples

```
      PLUS←+
      PLUS
+
      SUM←+/
      SUM
+/
```

Function expressions may include defined functions and operators. These are displayed as a ∇ followed by their name.

Example

```
      ∇ R←MEAN X    ⍝ Arithmetic mean
[1]     R←(+/X)÷⍴X
      ∇

      MEAN
 ∇MEAN

      AVERAGE←MEAN
      AVERAGE
 ∇MEAN

      AVG←MEAN∘,
      AVG
 ∇MEAN ∘,
```

Operators

An operator is an operation on one or two operands which produces a function called a DERIVED FUNCTION. An operand may be a function or an array. Operators are not ambivalent. They require either one or two operands as applicable to the particular operator. However, the derived function may be ambivalent. The derived function need not return a result. Operators have higher precedence than functions. Operators have long scope on the left. That is, the left operand is the longest function or array expression on its left. The left operand may be terminated by:

1. the end of the expression
2. the right-most of two consecutive functions
3. a function with an array to its left
4. an array with a function to its left
5. an array or function to the right of a monadic operator.

A dyadic operator has short scope on the right. That is, the right operand of an operator is the single function or array on its right. Right scope may be extended by enclosing an expression in parentheses.

Examples

```
      ⍴¨X←'WILLIAM' 'MARY' 'BELLE'
 7  4  5

      ⍴∘⍴¨X
 1  1  1

      (⍴∘⍴)¨X
 1  1  1

      ⎕∘←∘⎕VR¨'PLUS' 'MINUS'
     ∇ R←A PLUS B
[1]    R←A+B
     ∇
     ∇ R←A MINUS B
[1]    R←A-B
     ∇

      PLUS/1 2 3 4
10
```

Defined Operators

Operators may be defined with the system function ⎕FX, or with the function editor. A defined operator consists of a HEADER which identifies the syntax of the operator, and a BODY in which one or more APL statements are specified.

A defined operator may have one or two operands; and its derived function may have one or two arguments, and may or may not produce a result. The header syntax defines the operator name, its operand(s), the argument(s) to its derived function, and the result (if any) of its derived function. The names of the operator and its operand(s) are separated from the name(s) of the argument(s) to its derived function by parentheses.

Example

```
      ∇ R←A(F AND G)B
[1]     R←(A F B)(A G B)
      ∇
```

The above example shows a dyadic operator called AND with two operands (F and G). The operator produces a derived function which takes two arguments (A and B), and produces a result (R).

```
      12 +AND÷ 4
16 3
```

Operands passed to an operator may be either functions or arrays.

```
      12 (3 AND 5) 4
12 3 4  12 5 4

      12 (× AND 5) 4
48  12 5 4
```

Namespace Syntax

Names within namespaces may be referenced *explicitly* or *implicitly*. An ***explicit*** reference requires that you identify the object by its full or relative pathname using a `'.'` syntax; for example:

```
      X.NUMB ← 88
```

sets the variable `NUMB` in namespace `X` to 88.

```
      88 UTIL.FOO 99
```

calls dyadic function `FOO` in namespace `UTIL` with left and right arguments of 88 and 99 respectively. The interpreter can distinguish between this use of `'.'` and its use as the inner product operator, because the leftmost name: `UTIL` is a (class 9) namespace, rather than a (class 3) function.

The general namespace reference syntax is:

```
  SPACE . SPACE . (...) EXPR
```

Where `SPACE` is an *expression* which resolves to a namespace reference, and `EXPR` is any APL expression to be resolved in the resulting namespace.

There are two special space names:

  `#` is the top level or 'Root' namespace.
  `##` is the parent or space containing the current namespace.

`⎕SE` is a system namespace which is preserved across workspace load and clear.

Examples

```
      WSDOC.PAGE.NO +← 1      ⍝ Increment WSDOC page count

      #.⎕NL 2                    ⍝ Variables in root space

      UTIL.⎕FX 'Z←DUP A' 'Z←A A'    ⍝ Fix remote function

      ##.⎕ED'FOO'          ⍝ Edit function in parent space

      ⎕SE.RECORD ← PERS.RECORD    ⍝ Copy from PERS to ⎕SE

      UTIL.(⎕EX ⎕NL 2)         ⍝ Expunge variables in UTIL


      (⊃⎕SE #).(⍎⊃↓⎕NL 9).(⎕NL 2)     ⍝ Vars in first ⎕SE
                                      ⍝ namespace.

      UTIL.⍎STRING         ⍝ Execute STRING in UTIL space
```

You may also reference a function or operator in a namespace ***implicitly*** using the mechanism provided by ⎕EXPORT and ⎕PATH. If you reference a name that is undefined in the current space, the system searches for it in the list of exported names defined for the namespaces specified by ⎕PATH. See ⎕EXPORT and ⎕PATH for further details.

Evaluation

When the interpreter encounters a namespace reference, it :

- Switches to the namespace.
- Evaluates the name.
- Switches back to the original namespace.

If for example, in the following, the current namespace is #.W, the interpreter evaluates the line:

```
      A ← X.Y.DUP MAT
```

in the following way:

- Evaluate array MAT in current namespace W to produce argument for function.
- Switch to namespace X.Y within W,
- Evaluate function DUP in namespace W.X.Y with argument.
- Switch back to namespace W,
- Assign variable A in namespace W.

Static Vs Dynamic Localisation

The rules for name resolution have been generalised for namespaces.

In flat APL, the interpreter searches the state indicator to resolve names referenced by a defined function or operator. If the name does not appear in the state indicator, then the workspace-global name is assumed.

With namespaces, a defined function or operator is evaluated in its 'home' namespace. When a name is referenced, the interpreter searches only those lines of the state indicator which belong to the home namespace. If the name does not appear in any of these lines, the home namespace-global value is assumed.

For example, if `#.FN1` calls `XX.FN2` calls `#.FN3` calls `XX.FN4`, then:

`FN1`:
- is evaluated in `#`
- can see its own dynamic local names
- can see global names in `#`

`FN2`:
- is evaluated in `XX`
- can see its own dynamic local names
- can see global names in `XX`

`FN3`:
- is evaluated in `#`
- can see its own dynamic local names
- can see dynamic local names in `FN1`
- can see global names in `#`

`FN4`:
- is evaluated in `XX`
- can see its own dynamic local names
- can see dynamic local names in `FN2`
- can see global names in `XX`

Namespace References

A *namespace reference*, or *ref* for short, is a unique data type that is distinct from and in addition to *number* and *character*.

Any expression may result in a *ref*, but the simplest one is the namespace itself:

```
      )NS NS1           ⍝ Make a namespace called NS1
      NS1.A←1           ⍝ and populate it with variables A
      NS1.B←2 3⍴⍳6      ⍝ and B

      NS1               ⍝ expression results in a ref
#.NS1
```

You may assign a ref ; for example:

```
      X←NS1
      X
#.NS1
```

In this case, the display of `X` informs you that `X` refers to the named namespace `#.NS1`.

You may also supply a ref as an argument to a defined or dynamic function:

```
     ∇ FOO ARG
[1]    ARG
     ∇
      FOO NS1
#.NS1
```

The name class of a *ref* is 9.

```
      ⎕NC 'X'
9
```

You may use a ref to a namespace anywhere that you would use the namespace itself. For example:

```
      X.A
1
      X.B
1 2 3
4 5 6
```

Notice that refs are references to namespaces, so that if you make a copy, it is the reference that is copied, not the namespace itself. This is sometimes referred to as a shallow as opposed to a deep copy. It means that if you change a ref, you actually change the namespace that it refers to.

```
      X.A+←1
      X.A
2
      NS1.A
2
```

Similarly, a ref passed to a defined function is call-by-reference, so that modifications to the content or properties of the argument namespace using the passed reference, persist after the function exits. For example:

```
     ∇ FOO nsref
[1]    nsref.B+←nsref.A
     ∇

      FOO NS1
      NS1.B
3 4 5
6 7 8
```

```
      FOO X
      NS1.B
5 6  7
8 9 10
```

Notice that the expression to the right of a dot may be arbitrarily complex and will be executed within the namespace or ref to the left of the dot.

```
      X.(C←A×B)
      X.C
10 12 14
16 18 20
      NS1.C
10 12 14
16 18 20
```

Unnamed Namespaces

The monadic form of `⎕NS` makes a new (and unique) unnamed namespace and returns a ref to it.

One use of unnamed namespaces is to represent hierarchical data structures; for example, a simple employee database:

The first record is represented by `JOHN` which is a ref to an *unnamed* namespace:

```
      JOHN←⎕NS ''
      JOHN
#.[Namespace]

      JOHN.FirstName←'John'
      JOHN.FirstName
John

      JOHN.LastName←'Smith'
      JOHN.Age←50
```

Data variables for the second record, `PAUL`, can be established using strand, or vector, assignment:

```
      PAUL←⎕NS ''
      PAUL.(FirstName LastName Age←'Paul' 'Brown' 44)
```

The function `SHOW` can be used to display the data in each record (the function is split into 2 lines only to fit on the printed page). Notice that its argument is a ref.

```
     ∇ R←SHOW PERSON
[1]    R←PERSON.FirstName,' ',PERSON.LastName
[2]    R, ←' is ',⍕PERSON.Age
     ∇

      SHOW JOHN
John Smith is 50

      SHOW PAUL
Paul Brown is 44
```

An alternative version of the function illustrates the use of the `:With :EndWith` control structure to execute an expression, or block of expressions, within a namespace:

```
     ∇ R←SHOW1 PERSON
[1]    :With PERSON
[2]        R←FirstName,' ',LastName,' is ',(⍕Age)
[3]    :EndWith
     ∇

      SHOW1 JOHN
John Smith is 50
```

In this case, as only a single expression is involved, it can be expressed more simply using parentheses.

```
     ∇ R←SHOW2 PERSON
[1]    R←PERSON.(FirstName,' ',LastName,' is ',(⍕Age))
     ∇

      SHOW2 PAUL
Paul Brown is 44
```

Dynamic functions also accept refs as arguments:

```
      SHOW3←{
        ⍵.(FirstName,' ',LastName,' is ',⍕Age)
      }

      SHOW3 JOHN
John Smith is 50
```

Arrays of Namespace References

You may construct arrays of refs using strand notation, catenate (,) and reshape (⍴).

```
      EMP←JOHN PAUL
      ⍴EMP
2
      EMP
 #.[Namespace]  #.[Namespace]
```

Like any other array, an array of refs has name class 2:

```
      ⎕NC 'EMP'
2
```

Expressions such as indexing and pick return refs that may in turn be used as follows:

```
      EMP[1].FirstName
John
      (2⊃EMP).Age
44
```

The each (¨) operator may be used to apply a function to an array of refs:

```
      SHOW¨EMP
 John Smith is 50  Paul Brown is 44
```

An *array* of namespace references (refs) to the left of a '`.`' is expanded according to the following rule, where `x` and `y` are refs, and `exp` is an arbitrary expression:

```
      (x y).exp → (x.exp)(y.exp)
```

If `exp` evaluates to a function, the items of its argument array(s) are *distributed* to each referenced function. In the dyadic case, there is a 3-way distribution among: left argument, referenced functions and right argument.

Monadic function `f`:	`(x y).f d e → (x.f d)(y.f e)`
Dyadic function `g`:	`a b (x y).g  d e → (a x.g d)(b y.g e)`

An array of refs to the left of an assignment arrow is expanded thus:

```
      (x y).a←c d   →  (x.a←c)(y.a←d)
```

Note that the array of refs can be of any rank. In the limiting case of a simple scalar array, the *array* construct: `refs.exp` is identical to the *scalar* construct: `ref.exp`.

Note that the expression to the right of the '`.`' *pervades* a nested array of refs to its left:

```
      ((u v)(x y)).exp → ((u.exp)(v.exp))((x.exp)(y.exp))
```

Note also that with *successive* expansions `(u v).(x y z).` →, the final number of 'leaf' terms is the *product* of the number of refs at each level.

Examples:

```
      JOHN.Children←⎕NS¨'' ''
      ⍴JOHN.Children
2
      JOHN.Children[1].FirstName←'Andy'
      JOHN.Children[1].Age←23

      JOHN.Children[2].FirstName←'Katherine'
      JOHN.Children[2].Age←19

      PAUL.Children←⎕NS¨'' ''
      PAUL.Children[1].(FirstName Age←'Tom' 25)
      PAUL.Children[2].(FirstName Age←'Jamie' 22)

      ⍴EMP
2
      (⊃EMP).Children.(FirstName Age)
  Andy  23   Katherine  19

      DISPLAY (2⊃EMP).Children.(FirstName Age)
.→----------------------------.
| .→---------. .→-----------. |
| | .→--.    | | .→----.    | |
| | |Tom| 25 | | |Jamie| 22 | |
| | '---'    | | '-----'    | |
| '∊---------' '∊-----------' |
'∊----------------------------'

      EMP.Children ⍝ Is an array of refs
  #.[Namespace]  #.[Namespace]   #.[Namespace]  ...

      EMP.Children.(FirstName Age)
   Andy  23   Katherine  19    Tom  25   Jamie  22
```

Distributed Assignment

Assignment pervades nested strands of names to the left of the arrow. The conformability rules are the same as for scalar (pervasive) dyadic primitive functions such as '+'. The mechanism can be viewed as a way of naming the parts of a *structure*.

Examples:

```
      EMP.(FirstName Age)
 JOHN  43   PAUL  44

      EMP.(FirstName Age)←('Jonathan' 21)('Pauline' 22)

      EMP.(FirstName Age)
 Johnathan  21   Pauline  22

      ⍝ Distributed assignment is pervasive
      JOHN.Children.(FirstName Age)
 Andy  23   Katherine  19

      JOHN.Children.(FirstName Age)←('Andrew' 21)('Kate' 9)

      JOHN.Children.(FirstName Age)
 Andrew  21   Kate  9
```

More Examples:

```
      ((a b)(c d))←(1 2)(3 4)   ⍝ a←1 ⋄ b←2 ⋄ c←3 ⋄ d←4

      ((⎕io ⎕ml)vec)←0 ⎕av      ⍝ ⎕io←0 ⋄ ⎕ml←0 ⋄ vec←⎕av

      (i (j k))+←1 2            ⍝ i+←1 ⋄ j+←2 ⋄ k+←2

⍝ Naming of parts:

      ((first last) sex (street city state))←n⊃pvec

⍝ Distributed assignment in :For loop:

      :For (i j)(k l) :In array

⍝ Ref array expansion:

      (x y).(first last)←('John' 'Doe')('Joe' 'Blow')
      (f1 f2).(b1 b2).Caption←⊂'OK' 'Cancel'
```

```
⍝ Structure rearrangement:

      rotate1←{       ⍝ Simple binary tree rotation.
           (a b c)d e←⍵
           a b(c d e)
      }
      rotate3←{       ⍝ Compound binary tree rotation.
           (a b(c d e))f g←⍵
           (a b c)d(e f g)
      }
```

Distributed Functions

Namespace ref array expansion syntax applies to functions too.

```
      JOHN.PLOT←{↑⍵⍴¨'⎕'}
      JOHN.PLOT ⍳10
⎕
⎕⎕
⎕⎕⎕
⎕⎕⎕⎕
⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

      PAUL.PLOT←{(⍵,¨1)⍴¨'⎕'}
      PAUL.PLOT ⍳10
 ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
    ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
       ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
          ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
             ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
                ⎕  ⎕  ⎕  ⎕  ⎕
                   ⎕  ⎕  ⎕  ⎕
                      ⎕  ⎕  ⎕
                         ⎕  ⎕
                            ⎕

      EMP.PLOT⊂⍳10  ⍝ (temporary vector of functions)
 ⎕            ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕              ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕                ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕                  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕                    ⎕  ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕                      ⎕  ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕                        ⎕  ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕⎕                          ⎕  ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕⎕⎕                            ⎕  ⎕
 ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕                              ⎕
```

```
      (x y).⎕NL 2 3                  ⍝ x:vars, y:fns
 varx  funy

        (x y).⎕NL⊂2 3                  ⍝ x&y: vars&fns
 funx  funy
 varx  vary

        (x y).(⎕NL¨)⊂2 3               ⍝ x&y: separate vars&fns
  varx  funx    vary  funy

        'v'(x y).⎕NL 2 3               ⍝ x:v-vars, y:v-fns
 varx

        'vf'(x y).⎕NL 2 3              ⍝ x:v-vars, y:f-fns
 varx  funy
                                       ⍝ x:v-vars&fns,
        'vf'(x y).⎕NL⊂2 3              ⍝ y:f-vars&fns
 varx  funy

        x.⎕NL 2 3                      ⍝ depth 0 ref
funx
varx

        (x y).⎕NL⊂2 3                  ⍝ depth 1 refs
 funx  funy
 varx  vary

        ((u v)(x y)).⎕NL⊂⊂2 3          ⍝ depth 2 refs
  funu  funv    funx  funy
  varu  varv    varx  vary

        (1 2)3 4(w(x y)z).+1 2(3 4) ⍝ argument distribution.
 2 3  5 5  7 8
```

Operators

A function passed as operand to a primitive or defined operator, carries its namespace context with it. This means that if subsequently, the function operand is applied to an argument, it executes in its home namespace, irrespective of the namespace from which the operator was invoked or defined.

Examples

```
      VAR←99                           ë #.VAR

      )NS X
#.X
      X.VAR←77                         ë X.VAR
      X.⎕FX'Z←FN R' 'Z←R,VAR'

      )NS Y
#.Y
      Y.VAR←88                         ë Y.VAR
      Y.⎕FX'Z←(F OP)R' 'Z←F R'

      X.FN¨⍳3
 1 77  2 77  3 77

      X.FN 'VAR:'
 VAR: 77

      X.FN Y.OP 'VAR:'
 VAR: 77
      ⍎ Y.OP'VAR'
99
```

Summary

Apart from its use as a decimal separator (`3.14`), '`.`' is interpreted by looking at the type or *class* of the expression to its left:

Template	Interpretation	Example
`∘.`	Outer product	`2 3 ∘.× 4 5`
function`.`	Inner product	`2 3 +.× 4 5`
ref`.`	Namespace reference	`2 3 x.foo 4 5`
array`.`	Reference array expansion	`(x y).⎕nc⊂'foo'`

Threads

Overview

Dyalog APL supports multithreading - the ability to run more than one APL expression at the same time.

This unique capability allows you to perform background processing, such as printing, database retrieval, database update, calculations, and so forth while at the same time perform other interactive tasks.

Multithreading may be used to improve throughput and system responsiveness.

A *thread* is a strand of execution in the APL workspace.

A thread is created by calling a function *asynchronously*, using the new primitive operator 'spawn': & or by the asynchronous invocation of a callback function.

With a traditional APL *synchronous* function call, execution of the calling environment is paused, *pendent* on the return of the called function. With an *asynchronous* call, both calling environment and called function proceed to execute concurrently.

An asynchronous function call is said to start a new *thread* of execution. Each thread has a unique *thread number*, with which, for example, its presence can be monitored or its execution terminated.

Any thread can spawn any number of sub-threads, subject only to workspace availability. This implies a hierarchy in which a thread is said to be a *child thread* of its *parent thread*. The *base thread* at the root of this hierarchy has thread number 0.

With multithreading, APL's stack or state indicator can be viewed as a branching tree in which the path from the base to each leaf is a thread.

When a parent thread terminates, any of its children which are still running, become the children of (are 'adopted' by) the parent's parent.

Thread numbers are allocated sequentially from 0 to 2147483647. At this point, the sequence 'wraps around' and numbers are allocated from 0 again avoiding any still in use. The sequence is reinitialised when a `)RESET` command is issued, or the active workspace is cleared, or a new workspace is loaded. A workspace may not be saved with threads other than the base thread: 0, running.

Threads introduce new language elements.

- Primitive operator, spawn: `&`.
- System functions: `⎕TID`, `⎕TCNUMS`, `⎕TNUMS`, `⎕TKILL`, `⎕TSYNC`.
- An extension to the GUI Event syntax to allow asynchronous callbacks.
- A control structure: `:Hold`.
- System commands: `)HOLDS`, `)TID`.
- Extended `)SI` and `)SINL` display.

Running CallBack Functions as Threads

A callback function is associated with a particular event via the Event property of the object concerned. A callback function is executed by `⎕DQ` when the event occurs, or by `⎕NQ`.

If you append the character `&` to the name of the callback function in the `Event` specification, the callback function will be executed asynchronously as a thread when the event occurs. If not, it is executed synchronously as before.

For example, the event specification:

```
      ⎕WS'Event' 'Select' 'DoIt&'
```

tells `⎕DQ` to execute the callback function `DoIt` *asynchronously as a thread* when a Select event occurs on the object.

Thread Switching

Programming with threads requires care.

The interpreter may switch between running threads at the following points:

- Between any two lines of a defined (or dynamic) function or operator.
- While waiting for a `⎕DL` to complete.
- While waiting for a `⎕FHOLD` to complete.
- While awaiting input from:
  `⎕DQ`
  `⎕SR`
  `⎕ED`
  The session prompt or `⎕:` or `⍞`.
- While awaiting the completion of an external operation:
  A call on an external (AP) function.
  A call on a `⎕NA` (DLL) function
  A call on an OLE function.

At any of these points, the interpreter might execute code in other threads. If such threads change the global environment; for example by changing the value of, or expunging a name; then the changes will appear to have happened while the thread in question passes through the switch point. It is the task of the application programmer to organise and contain such behaviour!

You can prevent threads from interacting in critical sections of code by using the `:Hold` control structure.

High Priority Callback Functions

Note that the interpreter cannot perform thread-switching during the execution of a *high-priority callback*. This is a callback function that is invoked by a *high-priority* event which demands that the interpreter must return a result to Windows before it may process any other event. Such high-priority events include Configure, ExitWindows, DateTimeChange, DockStart, DockCancel, DropDown. It is therefore not permitted to use a `:Hold` control structure in a high-priority callback function.

Name Scope

APL’s name scope rules apply whether a function call is synchronous or asynchronous. For example when a defined function is called, names in the calling environment are visible, unless explicitly shadowed in the function header.

Just as with a synchronous call, a function called asynchronously has its own local environment, but can communicate with its parent and ‘sibling’ functions via local names in the parent.

This point is important. It means that siblings can run in parallel without danger of local name clashes. For example, a GUI application can accommodate multiple concurrent instances of its callback functions.

However, with an asynchronous call, as the calling function continues to execute, both child *and parent functions* may modify values in the calling environment. Both functions see such changes immediately they occur.

If a parent function terminates while any of its children are still running, those children will thenceforward ‘see’ local names in the environment that called the parent function. In cases where a child function relies on its parent’s environment (the setting of a local value of `⎕IO` for example), this would be undesirable, and the parent function would normally execute a `⎕TSYNC` in order to wait for its children to complete before itself exiting.

If, on the other hand, after launching an asynchronous child, the parent function calls a *new* function (either synchronously or asynchronously), names in the new function are beyond the purview of the original child. In other words, a function can only ever see its calling stack decrease in size – never increase. This is in order that the parent may call new defined functions without affecting the environment of its asynchronous children.

Using Threads

Put most simply, multithreading allows you to *appear to* run more than one APL function at the same time, just as Windows (or UNIX) *appears to* run more than one application at the same time. In both cases this is something of an illusion, although it does nothing to detract from its usefulness.

Dyalog APL implements an internal timesharing mechanism whereby it shares processing between threads. Although the mechanics are somewhat different, APL multithreading is rather similar to the multitasking provided by Windows 95/98 and NT. If you are running more than one application, Windows switches from one to another, allocating each one a certain *time slice* before switching. At any point in time, only one application is actually running; the others are paused, waiting.

If you execute more than one Dyalog APL thread, only one thread is actually running; the others are paused. Each APL thread has its own State Indicator, or SI stack. When APL switches from one thread to another, it saves the current stack (with all its local variables and function calls), restores the new one, and then continues processing.

Stack Considerations

When you start a thread, it begins with the SI stack of the calling function and sees all of the local variables defined in all the functions down the stack. However, unless the calling function specifically waits for the new thread to terminate (see `⎕TSYNC`), the calling functions will (bit by bit, in their turn) continue to execute. The new thread's view of its calling environment may then change. Consider the following example:

Suppose that you had the following functions: `RUN[3]` calls `INIT` which in turn calls `GETDATA` but as 3 separate threads with 3 different arguments:

```
     ∇ RUN;A;B
[1]    A←1
[2]    B←'Hello World'
[3]    INIT
[4]    CALC
[5]    REPORT
     ∇

     ∇ INIT;C;D
[1]    C←D←0
[2]    GETDATA&'Sales'
[3]    GETDATA&'Costs'
[4]    GETDATA&'Expenses'
     ∇
```

When each GETDATA thread starts, it immediately *sees* (via ⎕SI) that it was called by INIT which was in turn called by RUN, and it *sees* local variables A, B, C and D. However, once INIT[4] has been executed, INIT terminates, and execution of the root thread continues by calling CALC. From then on, each GETDATA thread no longer sees INIT (it thinks that it was called directly from RUN) nor can it see the local variables C and D that INIT had defined. However, it *does* continue to see the locals A and B defined by RUN, until RUN itself terminates.

Note that if CALC were also to define locals A and B, the GETDATA threads would still see the values defined by RUN and not those defined by CALC. However, if CALC were to modify A and B (as globals) without localising them, the GETDATA threads would see the modified values of these variables, whatever they happened to be at the time.

Globals and the Order of Execution

It is important to recognise that any reference or assignment to a global or semi-global object (including GUI objects) is **inherently dangerous** (i.e. a source of programming error) if more than one thread is running. Worse still, programming errors of this sort may not become apparent during testing because they are dependent upon random timing differences. Consider the following example:

```
      ∇ BUG;SEMI_GLOBAL
[1]     SEMI_GLOBAL←0
[2]     FOO& 1
[3]     GOO& 1
      ∇

      ∇ FOO
[1]     :If SEMI_GLOBAL=0
[2]         DO_SOMETHING SEMI_GLOBAL
[3]     :Else
[4]         DO_SOMETHING_ELSE SEMI_GLOBAL
[5]     :EndIf
      ∇

      ∇ GOO
[1]     SEMI_GLOBAL←1
      ∇
```

In this example, it is formally impossible to predict in which order APL will execute statements in `BUG`, `FOO` or `GOO` from `BUG[2]` onwards. For example, the actual sequence of execution may be:

```
      BUG[1] → BUG[2] → FOO[1] → FOO[2] →
               DO_SOMETHING[1]
```

or

```
      BUG[1] → BUG[2] → BUG[3] → GOO[1] →
               FOO[1] → FOO[2] → FOO[3] →
               FOO[4] → DO_SOMETHING_ELSE[1]
```

This is because APL may switch from one thread to another between any two lines in a defined function. In practice, because APL gives each thread a significant time-slice, it is likely to execute many lines, maybe even hundreds of lines, in one thread before switching to another. However, you must not rely on this; **thread-switching may occur at any time between lines in a defined function**.

Secondly, consider the possibility that APL switches from the `FOO` thread to the `GOO` thread after `FOO[1]`. If this happens, the value of `SEMI_GLOBAL` passed to `DO_SOMETHING` will be 1 and not 0. Here is another source of error.

In fact, in this case, there are two ways to resolve the problem. To ensure that the value of `SEMI_GLOBAL` remains the same from `FOO[1]` to `FOO[2]`, you may use diamonds instead of separate statements, e.g.

```
      :If SEMI_GLOBAL=0 ⋄ DO_SOMETHING SEMI_GLOBAL
```

Even better, although less efficient, you may use `:Hold` to synchronise access to the variable, for example:

```
      ∇ FOO
[1]    :Hold 'SEMI_GLOBAL'
[2]        :If SEMI_GLOBAL=0
[3]            DO_SOMETHING SEMI_GLOBAL
[4]        :Else
[5]            DO_SOMETHING_ELSE SEMI_GLOBAL
[6]        :EndIf
[7]    :EndHold
     ∇

     ∇ GOO
[1]    :Hold 'SEMI_GLOBAL'
[2]        SEMI_GLOBAL←1
[3]    :EndHold
     ∇
```

Now, although you still cannot be sure which of `FOO` and `GOO` will run first, you can be sure that `SEMI_GLOBAL` will not change (because `GOO` cuts in) within `FOO`.

Note that the string used as the argument to `:Hold` is completely arbitrary, so long as threads competing for the same resource use the same string.

A Caution

These types of problems are inherent in all multithreading programming languages, and not just with Dyalog APL. ***If you want to take advantage of the additional power provided by multithreading, it is advisable to think carefully about the potential interaction between different threads.***

Threads & Niladic Functions

In common with other operators, the spawn operator `&` may accept monadic or dyadic functions as operands, but not niladic functions. This means that, using spawn, you cannot start a thread that consists only of a niladic function

If you wish to invoke a niladic function asynchronously, you have the following choices:

- Turn your niladic function into a monadic function by giving it a dummy argument which it ignores.
- Call your niladic function with a dynamic function to which you give an argument that is implicitly ignored. For example, if the function `NIL` is niladic, you can call it asynchronously using the expression: `{NIL}& 0`
- Call your function via a dummy monadic function, e.g.

```
     ∇ NIL_M DUMMY
[1]    NIL
     ∇
      NIL_M& ''
```

- Use execute, e.g.

```
      ⍎& 'NIL'
```

Note that niladic functions *can* be invoked asynchronously as callback functions. For example, the statement:

```
      ⎕WS'Event' 'Select' 'NIL&'
```

will execute correctly as a thread, even though `NIL` is niladic. This is because callback functions are invoked directly by `⎕DQ` rather than as an operand to the spawn operator.

Threads & External Functions

External functions in dynamic link libraries (DLLs) defined using the ⎕NA interface may be run in separate C threads. Such threads:

- **take advantage of multiple processors** if the operating system permits.
- allow APL to **continue processing in parallel** during the execution of a ⎕NA function.

When you define an external function using ⎕NA, you may specify that the function be run in a separate C thread by appending an ampersand (&) to the function name, for example:

```
      'beep'⎕NA'user32|MessageBeep& i'
      ⍝ MessageBeep will run in a separate C thread
```

When APL first comes to execute a multi-threaded ⎕NA function, it starts a new C-thread, executes the function within it, and waits for the result. Other APL threads may then run in parallel.

Note that when the ⎕NA call finishes and returns its result, its new C-thread is retained to be re-used by any subsequent multithreaded ⎕NA calls made within the same APL thread. Thus any APL thread that makes any multi-threaded ⎕NA calls maintains a separate C-thread for their execution. This C-thread is discarded when its APL thread finishes.

Note that there is no point in specifying a ⎕NA call to be multi-threaded, unless you wish to execute other APL threads at the same time.

In addition, if your ⎕NA call needs to access an APL GUI object (strictly, a window or other handle) it should normally run within the same C-thread as APL itself, and not in a separate C-thread. This is because Windows associates objects with the C-thread that created them. Although you *can* use a multi-threaded ⎕NA call to access (say) a Dyalog APL Form via its window handle, the effects may be different than if the ⎕NA call was not multi-threaded. In general, ⎕NA calls that access APL (GUI) objects should not be multi-threaded.

If you wish to run the same ⎕NA call in separate APL threads at the same time, you must ensure that the DLL is *thread-safe*. Functions in DLLs which are not *thread-safe*, must be prevented from running concurrently by using the `:Hold` control structure. Note that all the standard Windows API DLLs **are** *thread safe*.

Notice that you may define two separate functions (with different names), one single-threaded and one multi-threaded, associated with the same function in the DLL. This allows you to call it in either way.

Synchronising Threads

Threads may be synchronised using *tokens* and a *token pool*.

An application can synchronise its threads by having one thread add tokens into the pool whilst other threads wait for tokens to become available and retrieve them from the pool.

Tokens posess two separate attributes, a *type* and a *value*.

The *type* of a token is a positive or negative integer scalar. The *value* of a token is any arbitrary array that you might wish to associate with it.

The token pool may contain up to 2*31 tokens; they do not have to be unique neither in terms of their types nor of their values.

The following system functions are used to manage the token pool:

`⎕TPUT`	Puts tokens into the pool.
`⎕TGET`	If necessary waits for, and then retrieves some tokens from the pool.
`⎕TPOOL`	Reports the types of tokens in the pool
`⎕TREQ`	Reports the token requests from specific threads

A simple example of a thread synchronisation requirement occurs when you want one thread to reach a certain point in processing before a second thread can continue. Pehaps the first thread performs a calculation, and the second thread must wait until the result is available before it can be used.

This can be achieved by having the first thread put a specific type of token into the pool using `⎕TPUT`. The second thread waits (if necessary) for the new value to be available by calling `⎕TGET` with the same token type.

Notice that when `⎕TGET` returns, the specifed tokens are *removed* from the pool. However, *negative* token types will satisfy an infinite number of requests for their positive equivalents.

The system is designed to cater for more complex forms of synchronisation. For example, a *semaphore* to control a number of resources can be implemented by keeping that number of tokens in the pool. Each thread will take a token while processing, and return it to the pool when it has finished.

A second complex example is that of a *latch* which holds back a number of threads until the coast is clear. At a signal from another thread, the latch is opened so that all of the threads are released. The latch may (or may not) then be closed again to hold up subsequently arriving threads. A practical example of a latch is a ferry terminal.

Semaphore Example

A *semaphore* to control a number of resources can be implemented by keeping that number of tokens in the pool. Each thread will take a token while processing, and return it to the pool when it has finished.

For example, if we want to restrict the number of threads that can have sockets open at any one time.

```
    sock←99                     ⍝ socket-token
                                  any +ive number will do).
    ⎕TPUT 5/sock                ⍝ add 5 socket-tokens to pool.

    ∇ sock_open ...
[1]   :If sock=⎕TGET sock       ⍝ grap a socket token
[.]       ...                   ⍝ do stuff.
[.]       ⎕TPUT sock            ⍝ release socket token
[.]   :Else
[.]       error'sockets off'    ⍝ sockets switched off by
                                  retract (see below).
[.]   :EndIf
    ∇

    0 ⎕TPUT ⎕treq ⎕tnums        ⍝ retract socket "service"
                                  with 0 value.
```

Latch Example

A *latch* holds back a number of threads until the coast is clear. At a signal from another thread, the latch is opened so that all of the threads are released. The latch may (or may not) then be closed again to hold up subsequently arriving threads.

A visual example of a latch might be a ferry terminal, where cars accumulate in the queue until the ferry arrives. The barrier is then opened and all (up to a maximum number) of the cars are allowed through it and on to the ferry. When the last car is through, the barrier is re-closed.

```
    tkt←6                             ⍝ 6-token: ferry ticket.

    ∇ car ...
[1]   ⎕TGET tkt                       ⍝ await ferry.
[2]   ...

    ∇ ferry ...
[1]   arrives in port
[2]   ⎕TPUT(↑,/⎕treq ⎕tnums)∩tkt  ⍝ ferry tickets for all.
[3]   ...
```

Note that it is easy to modify this example to provide a maximum number of ferry places per trip by inserting `max_places↑` between `⎕TPUT` and its argument. If fewer cars than the ferry capacity are waiting, the `↑` will fill with trailing 0s. This will not cause problems because zero tokens are ignored.

Let us replace the car ferry with a new road bridge. Once the bridge is ready for traffic, the barrier could be opened permanently by putting a *negative* ticket in the pool.:

```
    ⎕TPUT -tkt      ⍝ open ferry barrier permananently.
```

Cars could choose to take the last ferry if there are places:

```
    ∇ car ...
[1]   :Select ⎕TGET tkt
[2]   :Case  tkt ⋄ take the last ferry.
[3]   :Case -tkt ⋄ ferry full: take the new bridge.
[4]   :End
```

The above `:Select` works because by default, `⎕TPUT -tkt` puts a *value* of `-tkt` into the token.

Debugging Threads

If a thread sustains an untrapped error, its execution is *suspended* in the normal way. If the *Pause on Error* option (see User Guide) is set, all other threads are *paused*. If *Pause on Error* option (see User Guide) is not set, other threads will continue running and it is possible for another thread to encounter an error and suspend.

Using the facilities provided by the Tracer and the Threads Tool (see User Guide) it is possible to interrupt (suspend) and restart individual threads, and to pause and resume individual threads, so any thread may be in one of three states - *running*, *suspended* or *paused*.

The Tracer and the Session may be connected with any suspended thread and you can switch the attention of the Session and the Tracer between suspended threads using `)TID` or by clicking on the appropriate tab in the Tracer. At this point, you may:

- Examine and modify local variables for the currently suspended thread.
- Trace and edit functions in the current thread.
- Cut back the stack in the currently suspended thread.
- Restart execution.
- Start new threads

The error message from a thread other than the base is prefixed with its thread number:

```
260:DOMAIN ERROR
Div[2] rslt←num÷div
      ^
```

State indicator displays: `)SI` and `)SINL` have been extended to show threads' tree-like calling structure.

```
      )SI
·   Calc[1]
&5
·   ·   DivSub[1]
·   &7
·   ·   DivSub[1]
·   &6
·   Div[2]*
&4
Sub[3]
Main[4]
```

Here, `Main` has called `Sub`, which has spawned threads `4` and `5` with functions: `Div` and `Calc`. Function `Div`, after spawning `DivSub` in each of threads `6` and `7`, has been suspended at line `[2]`.

Removing stack frames using *Quit* from the Tracer or → from the session affects only the current thread. When the final stack frame in a thread (other than the base thread) is removed, the thread is expunged.

`)RESET` removes all but the base thread.

Note the distinction between a *suspended* thread and a *paused* thread.

A *suspended* thread is stopped at the beginning of a line in a defined function or operator. It may be connected to the Session so that expressions executed in the Session do so in the context of that thread. It may be *restarted* by executing `→line` (typically, `→⎕LC`).

A *paused* thread is an inactive thread that is currently being ignored by the thread scheduler. A paused thread may be paused within a call to `⎕DQ`, a call on an external function, at the beginning of a line, or indeed at any of the thread-switching points described earlier in this chapter.

A paused thread may be *resumed* only by the action of a menu item or button . A paused thread resumes only in the sense that it ceases to be ignored by the thread scheduler and will therefore be switched back to at some point in the future. It does not actually continue executing until the switch occurs.

External Variables

An external variable is a variable whose contents (value) reside not in the workspace, but in a file. An external variable is associated with a file by the system function `⎕XT`. If at the time of association the file exists, the external variable assumes its value from the contents of the file. If the file does not exist, the external variable is defined but a `VALUE ERROR` occurs if it is referenced before assignment. Assignment of an array to the external variable or to an indexed element of the external variable has the effect of updating the file. The value of the external variable or the value of indexed elements of the external variable is made available in the workspace when the external variable occurs in an expression. No special restrictions are placed on the usage of external variables.

Normally, the files associated with external variables remain permanent in that they survive the APL session or the erasing of the external variable from the workspace. External variables may be accessed concurrently by several users, or by different nodes on a network, provided that the appropriate file access controls are established. Multi-user access to an external variable may be controlled with the system function `⎕FHOLD` between co-operating tasks.

Refer to the sections describing the system functions `⎕XT` and `⎕FHOLD` in *Chapter 6* for further details.

Examples

```
      'ARRAY' ⎕XT 'V'

      V←⍳10
      V[2] + 5
7

      ⎕EX'V'

      'ARRAY' ⎕XT 'F'
      F
1 2 3 4 5 6 7 8 9 10
```

Component Files

A component file is a data file maintained by Dyalog APL. It contains a series of APL arrays known as components which are accessed by reference to their relative positions or component number within the file. A set of system functions is provided to perform a range of file operations. (See *Chapter 6*.) These provide facilities to create or delete files, and to read and write components. Facilities are also provided for multi-user access including the capability to determine who may do what, and file locking for concurrent updates. (See *User Guide*.)

Auxiliary Processors

Auxiliary Processors (APs) are non-APL programs which provide Dyalog APL users with additional facilities. They run as separate tasks, and communicate with the Dyalog APL interpreter through pipes (UNIX) or via an area of memory (Windows). Typically, APs are used where speed of execution is critical, such as in screen management software, or for utility libraries. Auxiliary Processors may be written in any compiled language, although 'C' is preferred and is directly supported.

When an Auxiliary Processor is invoked from Dyalog APL, one or more **external functions** are fixed in the active workspace. Each external function behaves as if it were a locked defined function, but is in effect an entry point into the Auxiliary Processor. An external function occupies only a negligible amount of workspace. (See *User Guide*.)

Migration Level

⎕ML determines the degree of migration of the Dyalog APL language towards IBM's APL2. Unless otherwise stated, the manual assumes ⎕ML has a value of 0.

Key to Notation

The following definitions and conventions apply throughout this manual:

`f`	A function, or an operator's left argument when a function.
`g`	A function, or an operator's right argument when a function.
`A`	An operator's left argument when an array.
`B`	An operator's right argument when an array.
`X`	The left argument of a function.
`Y`	The right argument of a function.
`R`	The explicit result of a function.
`[K]`	Axis specification.
`[I]`	Index specification.
`{X}`	The left argument of a function is optional.
`{R}←`	The function may or may not return a result, or the result may be suppressed.

The term *function* may refer to a primitive function, a system function, a defined (canonical, dynamic or assigned) function or a derived (from an operator) function.

CHAPTER 2

Defined Functions & Operators

A defined function is a program that takes 0, 1, or 2 arrays as **arguments** and may produce an array as a result. A defined operator is a program that takes 1 or 2 functions or arrays (known as **operands**) and produces a **derived function** as a result. To simplify the text, the term **operation** is used within this chapter to mean function or operator.

Canonical Representation

Operations may be defined with the system function `⎕FX` (Fix) or by using the editor within definition mode. Applying `⎕CR` to the character array representing the name of an already established operation will produce its canonical representation. A defined operation is composed of lines. The first line (line 0) is called the operation HEADER. Remaining lines are APL statements, called the BODY.

The operation header consists of the following parts:

1. its model syntactical form,
2. an optional list of local names, each preceded by a semi-colon (`;`) character,
3. an optional comment, preceded by the symbol `⍝`.

Only the model is required. If local names and comments are included, they must appear in the prescribed order.

Model Syntax

The model for the defined operation identifies the name of the operation, its valence, and whether or not an explicit result may be returned. Valence is the number of explicit arguments or operands, either 0, 1 or 2; whence the operation is termed NILADIC, MONADIC or DYADIC respectively. Only a defined function may be niladic. There is no relationship between the valence of a defined operator, and the valence of the derived function which it produces. Defined functions and derived functions produced by defined operators may be ambivalent, i.e. may be executed monadically with one argument, or dyadically with two. An ambivalent operation is identified in its model by enclosing the left argument in braces.

The value of a result-returning function or derived function may be suppressed in execution if not explicitly used or assigned by enclosing the result in its model within braces. Such a suppressed result is termed SHY.

Figures 2(i), 2(ii) and 2(iii) show all possible models for defined functions and operators respectively.

Defined Functions

Result	Niladic	Monadic	Dyadic	Ambivalent
None	`f`	`f Y`	`X f Y`	`{X} f Y`
Explicit	`R←f`	`R←f Y`	`R←X f Y`	`R←{X} f Y`
Suppressed	`{R}←f`	`{R}←f Y`	`{R}←X f Y`	`{R}←{X} f Y`

Figure 2(i) : Models for Defined Functions

Note: The right argument Y and/or the result R may be represented by a single name, or as a blank-delimited list of names surrounded by parentheses. For further details, see Namelists.

Derived Functions produced by Monadic Operator

Result	Monadic	Dyadic	Ambivalent
None	`(A op)Y`	`X(A op)Y`	`{X}(A op)Y`
Explicit	`R←(A op)Y`	`R←X(A op)Y`	`R←{X}(A op)Y`
Suppressed	`{R}←(A op)Y`	`{R}←X(A op)Y`	`{R}←{X}(A op)Y`

Figure 2(ii) : Models for Defined Operators (Monadic)

Derived Functions produced by Dyadic Operator

Result	Monadic	Dyadic	Ambivalent
None	`(A op B)Y`	`X(A op B)Y`	`{X}(A op B)Y`
Explicit	`R←(A op B)Y`	`R←X(A op B)Y`	`R←{X}(A op B)Y`
Suppress	`{R}←(A op B)Y`	`{R}←X(A op B)Y`	`{R}←{X}(A op B)Y`

Figure 2(iii) : Models for Defined Operators (Dyadic)

Statements

A statement is a line of characters understood by APL. It may be composed of:

1. a LABEL (which must be followed by a colon `:`), or a CONTROL STATEMENT (which is preceded by a colon), or both,

2. an EXPRESSION (see *Chapter 1*),

3. a SEPARATOR (consisting of the diamond character `⋄` which must separate adjacent expressions),

4. a COMMENT (which must start with the character `⍝`).

Each of the four parts is optional, but if present they must occur in the given order except that successive expressions must be separated by `⋄`. Any characters occurring to the right of the first comment symbol (`⍝`) that is not within quotes is a comment.

Comments are not executed by APL. Expressions in a line separated by `⋄` are taken in left-to-right order as they occur in the line. For output display purposes, each separated expression is treated as a separate statement.

Examples

```
      5×10
50

      MULT: 5×10
50

      MULT: 5×10 ⋄ 2×4
50
8

      MULT: 5×10 ⋄ 2×4  ⍝ MULTIPLICATION
50
8
```

Global & Local Names

The following names, if present, are local to the defined operation:

1. the result,
2. the argument(s) and operand(s),
3. additional names in the header line following the model, each name preceded by a semi-colon character,
4. labels,
5. the argument list of the system function `⎕SHADOW` when executed,
6. a name assigned within a Dynamic Function.

All names in a defined operation must be valid APL names. The same name may be repeated in the header line, including the operation name (whence the name is localised). Normally, the operation name is not a local name.

The same name may not be given to both arguments or operands of a dyadic operation. The name of a label may be the same as a name in the header line. More than one label may have the same name. When the operation is executed, local names in the header line after the model are initially undefined; labels are assigned the values of line numbers on which they occur, taken in order from the last line to the first; the result (if any) is initially undefined.

In the case of a defined function, the left argument (if any) takes the value of the array to the left of the function when called; and the right argument (if any) takes the value of the array to the right of the function when called. In the case of a defined operator, the left operand takes the value of the function or array to the left of the operator when called; and the right operand (if any) takes the value of the function or array to the right of the operator when called.

During execution, a local name temporarily excludes from use an object of the same name with an active definition. This is known as LOCALISATION or SHADOWING. A value or meaning given to a local name will persist only for the duration of execution of the defined operation (including any time whilst the operation is halted). A name which is not local to the operation is said to be GLOBAL. A global name could itself be local to a pendent operation. A global name can be made local to a defined operation during execution by use of the system function `⎕SHADOW`. An object is said to be VISIBLE if there is a definition associated with its name in the active environment.

Examples

```
      A←1

     ∇ F
[1]    A←10
[2]  ∇

      F ⍝ <A> NOT LOCALISED IN <F>, GLOBAL VALUE REPLACED
      A
10
      A←1
      )ERASE F

     ∇ F;A
[1]    A←10
[2]  ∇

      F  ⍝ <A> LOCALISED IN <F>, GLOBAL VALUE RETAINED
      A
1
```

Any statement line in the body of a defined operation may begin with a LABEL. A label is followed by a colon (`:`). A label is a constant whose value is the number of the line in the operation defined by system function `⎕FX` or on closing definition mode.

The value of a label is available on entering an operation when executed, and it may be used but not altered in any expression.

Example

```
      ⎕VR'PLUS'
    ∇ R←{A} PLUS B
[1]    →DYADIC ⍴⍨2=⎕NC'A' ⋄ R←B ⋄ →END
[2]  DYADIC: R←A+B
[3]  END:
    ∇

      1 ⎕STOP'PLUS'

      2 PLUS 2

PLUS[1]
      DYADIC
2

      END
3
```

Namelists

The right argument and the result of a function may be specified in the function header by a single name or by a *Namelist*. In this context, a Namelist is a blank-delimited list of names surrounded by a single set of parentheses.

Names specified in a Namelist are automatically local to the function; there is no need to localise them explicitly using semi-colons.

If the *right argument* of a function is declared as a Namelist, the function will only accept a right argument that is a vector whose length is the same as the number of names in the Namelist. Calling the function with any other argument will result in a `LENGTH ERROR` in the calling statement. Otherwise, the elements of the argument are assigned to the names in the Namelist in the specified order.

Example:

```
     ∇ IDN←Date2IDN(Year Month Day)
[1]    'Year is ',⍕Year
[2]    'Month is ',⍕Month
[3]    'Day is ',⍕Day
[4] ...
     ∇

      Date2IDN 2004 4 30
Year is 2004
Month is 4
Day is 30

      Date2IDN 2004 4
LENGTH ERROR
      Date2IDN 2004 4
     ^
```

Note that if you specify a *single* name in the Namelist, the function may be called only with a 1-element vector or a scalar right argument.

If the *result* of a function is declared as a Namelist, the values of the names will automatically be stranded together in the specified order and returned as the result of the function when the function terminates.

Example:

```
     ∇ (Year Month Day)←Birthday age
[1]    Year←1949+age
[2]    Month←4
[3]    Day←30
     ∇
      Birthday 50
1999 4 30
```

Function Declaration Statements

Certain statements that are used to identify the characteristics of a function in some way. These statements are not executable statements and may appear anywhere in the body of the function.

Access Statement :Access

```
:Access <Private|Public><Instance|Shared>
:Access <WebMethod>
```

The :Access statement is used to specify characteristics for functions that represent Methods in classes (see chapter 3). It is also applicable to Classes and Properties.

Element	Description
`Private|Public`	Specifies whether or not the method is accessible from outside the Class or an Instance of the Class. The default is `Private`.
`Instance|Shared`	Specifies whether the method runs in the Class or Instance. The default is `Instance`.
`WebMethod`	Specifies that the method is exported as a web method. This applies only to a Class that implements a Web Service.
`Overridable`	Applies only to an Instance Method and specifies that the Method may be overridden by a Method in a higher Class. See below.
`Override`	Applies only to an Instance Method and specifies that the Method overrides the corresponding Overridable Method defined in the Base Class. See below

Overridable/Override

Normally, a Method defined in a higher Class replaces a Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*.

However, a Method declared as being `Overridable` is replaced in situ (i.e. within its own Class) by a Method of the same name in a higher Class if that Method is itself declared with the `Override` keyword. For further information, see Superseding Base Class Methods.

WebMethod

Note that `:Access WebMethod` is equivalent to:

```
:Access Public
:Attribute System.Web.Services.WebMethodAttribute
```

Attribute Statement :Attribute

```
:Attribute <Name> [ConstructorArgs]
```

The :Attribute statement is used to attach .Net Attributes to a Method (or Class).

Attributes are descriptive tags that provide additional information about programming elements. Attributes are not used by Dyalog APL but other applications can refer to the extra information in attributes to determine how these items can be used. Attributes are saved with the *metadata* of Dyalog APL .NET assemblies.

Element	Description
`Name`	The name of a .Net attribute
`ConstructorArgs`	Optional arguments for the Attribute constructor

Examples

```
      :Attribute ObsoleteAttribute
      :Attribute ObsoleteAttribute 'Don''t use' 1
```

Implements Statement :Implements

```
:Implements Constructor <[:Base expr]>
:Implements Destructor
:Implements Method <InterfaceName.MethodName>
:Implements Trigger <name1><,name2,name3,...
```

The :Implements statement identifies the function to be one of the following special types.

Element	Description
`Constructor`	Specifies that the function is a class constructor.
`:Base expr`	Specifies that the Base Constructor be called with the result of the expression `expr` as its argument.
`Destructor`	Specifies that the method is a Class Destructor.
`Method`	Specifies that the function implements the Method `MethodName` whose syntax is specified by Interface `InterfaceName`.
`Trigger`	Identifies the function as a Trigger Function which is activated by changes to variables `name1`, `name2`, etc. (see Triggers).

Signature Statement :Signature

```
:Signature <rslttype←><name><arg1type arg1name>,...
```

This statement identifies the name and signature by which a function is exported as a method to be called from outside Dyalog APL. Several :Signature statements may be specified to allow the method to be called with different arguments and/or to specify a different result type.

Element	Description
`rslttype`	Specifies the data type for the result of the method
`name`	Specifies the name of the exported method.
`argntype`	Specifies the data type of the nth parameter
`argnname`	Specifies the name of the nth parameter

Argument and result data types are identified by the names of .Net Types which are defined in the .Net Assemblies specified by `⎕USING` or by a `:USING` statement.

Examples

In the following examples, it is assumed that the .Net Search Path (defined by `:Using` or `⎕USING` includes. `'System'`.

The following statement specifies that the function is exported as a method named `Format` which takes a single parameter of type `System.Object` named `Array`. The data type of the result of the method is an array (vector) of type `System.String`.

```
      :Signature String[]←Format Object Array
```

The next statement specifes that the function is exported as a method named `Catenate` whose result is of type `System.Object` and which takes 3 parameters. The first parameter is of type `System.Double` and is named `Dimension`. The second is of type `System.Object` and is named `Arg1`. The third is of type `System.Object` and is named `Arg2`.

```
      :Signature Object←Catenate Double Dimension,...
                                ...Object Arg1, Object Arg2
```

The next statement specifes that the function is exported as a method named `IndexGen` whose result is an array of type `System.Int32` and which takes 2 parameters. The first parameter is of type `System.Int32` and is named `N`. The second is of type `System.Int32` and is named `Origin`.

```
      :Signature Int32[]←IndexGen Int32 N, Int32 Origin
```

The next block of ststements specifies that the function is exported as a method named `Mix`. The method has 4 different signatures; i.e. it may be called with 4 different parameter/result combinations.

```
      :Signature Int32[,]←Mix Double Dimension, ...
            ...Int32[] Vec1, Int32[] Vec2
      :Signature Int32[,]←Mix Double Dimension,...
            ... Int32[] Vec1, Int32[] Vec2, Int32 Vec3
      :Signature Double[,]←Mix Double Dimension, ...
            ... Double[] Vec1, Double[] Vec2
      :Signature Double[,]←Mix Double Dimension, ...
            ... Double[] Vec1, Double[] Vec2, Double[] Vec3
```

Control Structures

Control structures provide a means to control the flow of execution in your APL programs.

Traditionally, lines of APL code are executed one by one from top to bottom and the only way to alter the flow of execution is using the branch arrow. So how do you handle logical operations of the form “If this, do that; otherwise do the other”?.

In APL this is often not a problem because many logical operations are easily performed using the standard array handling facilities that are absent in other languages. For example, the expression :

```
      STATUS←(1+AGE<16)⊃'Adult' 'Minor'
```

sets `STATUS` to `'Adult'` if `AGE` is 16 or more; otherwise sets `STATUS` to `'Minor'`.

Things become trickier if, depending upon some condition, you wish to execute one set of code instead of another, especially when the code fragments cannot conveniently be packaged as functions. Nevertheless, careful use of array logic, defined operators, the execute primitive function and the branch arrow can produce high quality maintainable and comprehensible APL systems.

Control structures provide an additional mechanism for handling logical operations and decisions. Apart from providing greater affinity with more traditional languages, Control structures may enhance comprehension and reduce programming errors, especially when the logic is complex. Control structures are not, however, a replacement for the standard logical array operations that are so much a part of the APL language.

Control Structures are blocks of code in which the execution of APL statements follows certain rules and conditions. Control structures are implemented using a set of *control words* that all start with the colon symbol (:). Control Words are case-insensitive.

There are eight different types of control structures defined by the control words, `:If`, `:While`, `:Repeat`, `:For`, `:Select`, `:With`, `:Trap` and `:Hold` . Each one of these control words may occur only at the beginning of an APL statement and indicates the start of a particular type of control structure.

Within a control structure, certain other control words are used as qualifiers. These are `:Else`, :`ElseIf`, `:AndIf`, `:OrIf`, `:Until`, `:Case` and `:CaseList`.

A third set of control words is used to identify the end of a particular control structure. These are `:EndIf`, `:EndWhile`, `:EndRepeat`, `:EndFor`, `:EndSelect`, `:EndWith`, `:EndTrap` and `:EndHold`. Although formally distinct, these control words may all be abbreviated to `:End`.

Finally, the `:GoTo`, `:Return`, `:Leave` and `:Continue` control words may be used to conditionally alter the flow of execution within a control structure.

Control words, including qualifiers such as `:Else` and :`ElseIf`, may occur only at the beginning of a line or expression in a diamond-separated statement. The only exceptions are `:In` and `:InEach` which must appear on the same line within a `:For` expression.

Key to Notation

The following notation is used to describe Control Structures within this section:

`aexp`	an expression returning an array,
`bexp`	an expression returning a single Boolean value (0 or 1),
`var`	loop variable used by `:For` control structure,
`code`	0 or more lines of APL code, including other (nested) control structures,
`andor`	**either** one or more `:AndIf` statements, **or** one or more `:OrIf` statements.

```
|
.----------------------.
|                      |
|<--------------.      |<---------------.
|               |      |                |
:AndIf bexp-----'      :OrIf bexp------'
|                      |
|<---------------------'
|
```

If Statement :If bexp

The simplest `:If` control structure is a single condition of the form:

```
[1]   :If AGE<21
[2]       expr 1
[3]       expr 2
[5]   :EndIf
```

If the test condition (in this case `AGE<21`) is true, the statements between the `:If` and the `:EndIf` will be executed. If the condition is false, none of these statements will be run and execution resumes after the `:EndIf`. Note that the test condition to the right of `:If` must return a single element Boolean value 1 (true) or 0 (false).

`:If` control structures may be considerably more complex. For example, the following code will execute the statements on lines `[2-3]` if `AGE<21` is 1 (true), **or alternatively**, the statement on line `[6]` if `AGE<21` is 0 (false).

```
[1]   :If AGE<21
[2]       expr 1
[3]       expr 2
[5]   :Else
[6]       expr 3
[7]   :EndIf
```

Instead of a single condition, it is possible to have multiple conditions using the `:ElseIf` control word. For example,

```
[1]   :If WINEAGE<5
[2]       'Too young to drink'
[5]   :ElseIf WINEAGE<10
[6]       'Just Right'
[7]   :ElseIf WINEAGE<15
[8]       'A bit past its prime'
[9]   :Else
[10]     'Definitely over the hill'
[11]  :EndIf
```

Notice that APL executes the expression(s) associated with the **first** condition that is true or those following the `:Else` if **none** of the conditions are true.

The `:AndIf` and `:OrIf` control words may be used to define a block of conditions and so refine the logic still further. You may qualify an `:If` or an `:ElseIf` with one or more `:AndIf` statements **or** with one or more `:OrIf` statements. You may not however mix `:AndIf` and `:OrIf` in the same conditional block. For example:

```
:If WINE.NAME≡'Chateau Lafitte'
:AndIf WINE.YEAR∊1962 1967 1970
    'The greatest?'
:ElseIf WINE.NAME≡'Chateau Latour'
:Orif WINE.NAME≡'Chateau Margaux'
:Orif WINE.PRICE>100
    'Almost as good'
:Else
    'Everyday stuff'
:EndIf
```

Please note that in a `:If` control structure, the conditions associated with each of the condition blocks are executed in order until an entire condition block evaluates to true. At that point, the APL statements following this condition block are executed. None of the conditions associated with any other condition block are executed. Furthermore, if an `:AndIf` condition yields 0 (false), it means that the entire block must evaluate to false so the system moves immediately on to the next block without executing the other conditions following the failing `:AndIf`. Likewise, if an `:OrIf` condition yields 1 (true), the entire block is at that point deemed to yield true and none of the following `:OrIf` conditions in the same block are executed.

:If Statement

```
|
:If bexp
|
.-------.
|       |
|       andor
|       |
|<------'
|
code
|
|<------------------------------.
|                               |
.-------.-------.               |
|       |       |               |
|       :Else   :ElseIf bexp    |
|       |       |               |
|       |       .-------.       |
|       |       |       |       |
|       |       |       andor   |
|       |       |       |       |
|       |       |<------'       |
|       |       |               |
|       code    code            |
|       |       |               |
|<------'       `---------------'
|
:End[If]
|
```

While Statement :While bexp

The simplest `:While` loop is :

```
[1]   I←0
[2]   :While I<100
[3]       expr1
[4]       expr2
[5]       I←I+1
[6]   :EndWhile
```

Unless `expr1` or `expr2` alter the value of `I`, the above code will execute lines `[3-4]` 100 times. This loop has a single condition; the value of `I`. The purpose of the `:EndWhile` statement is solely to mark the end of the iteration. It acts the same as if it were a branch statement, branching back to the `:While` line.

An alternative way to terminate a `:While` structure is to use a `:Until` statement. This allows you to add a second condition. The following example reads a native file sequentially as 80-byte records until it finds one starting with the string `'Widget'` or reaches the end of the file.

```
[1]   I←0
[2]   :While I<⎕NSIZE ¯1
[3]       REC←⎕NREAD ¯1 82 80
[4]       I←I+⍴REC
[5]   :Until 'Widget'≡6⍴REC
```

Instead of single conditions, the tests at the beginning and end of the loop may be defined by more complex ones using `:AndIf` and `:OrIf`. For example:

```
[1]   :While 100>i
[2]   :AndIf 100>j
[3]       i j←foo i j
[4]   :Until 100<i+j
[5]   :OrIf i<0
[6]   :OrIf j<0
```

In this example, there are complex conditions at both the start and the end of the iteration. Each time around the loop, the system tests that both `i` and `j` are less than or equal to 100. If either test fails, the iteration stops. Then, after `i` and `j` have been recalculated by `foo`, the iteration stops if `i+j` is equal to or greater than 100, or if either `i` or `j` is negative.

:While Statement

```
|
:While bexp
|
.-------.
|       |
|       andor
|       |
|<------'
|
code
|
.---------------.
|               |
:End[While]     :Until bexp
|               |
|               .-------.
|               |       |
|               |       andor
|               |       |
|               |<------'
|               |
|<--------------'
|
```

Repeat Statement :Repeat

The simplest type of `:Repeat` loop is as follows. This example executes lines `[3-5]` 100 times. Notice that as there is no conditional test at the beginning of a `:Repeat` structure, its code statements are executed at least once.

```
[1]   I←0
[2]   :Repeat
[3]       expr1
[4]       expr2
[5]       I←I+1
[6]   :Until I=100
```

You can have multiple conditional tests at the end of the loop by adding `:AndIf` or `:OrIf` expressions. The following example will read data from a native file as 80-character records until it reaches one beginning with the text string `'Widget'` or reaches the end of the file.

```
[1]   :Repeat
[2]       REC←⎕NREAD ¯1 82 80
[3]   :Until 'Widget'≡6ρREC
[4]   :OrIf 0=ρREC
```

A `:Repeat` structure may be terminated by an `:EndRepeat` (or `:End`) statement in place of a conditional expression. If so, your code must explicitly jump out of the loop using a `:Leave` statement or by branching. For example:

```
[1]   :Repeat
[2]       REC←⎕NREAD ¯1 82 80
[3]      :If 0=ρREC
[4]      :OrIf 'Widget'≡6ρREC
[5]         :Leave
[6]      :EndIf
[7]   :EndRepeat
```

:Repeat Statement

```
|
:Repeat
|
code
|
.----------------.
|                |
:End[Repeat]     :Until bexp
|                |
|                .-------.
|                |       |
|                |       andor
|                |       |
|                |<------'
|                |
|<---------------'
|
```

For Statement `:For var :In[Each] aexp`

Single Control Variable

The `:For` loop is used to execute a block of code for a series of values of a particular control variable. For example, the following would execute lines `[2-3]` successively for values of `I` from `⎕IO` to 10

```
[1]   :For I :In ⍳10
[2]       expr1 I
[3]       expr2 I
[4]   :EndFor
```

The way a `:For` loop operates is as follows. On encountering the `:For`, the expression to the right of `:In` is evaluated and the result stored. This is the *control array*. The *control variable*, named to the right of the `:For`, is then assigned the first value in the control array, and the code between `:For` and `:EndFor` is executed. On encountering the `:EndFor`, the control variable is assigned the next value of the control array and execution of the code is performed again, starting at the first line after the `:For`. This process is repeated for each value in the control array.

Note that if the control array is empty, the code in the `:For` structure is not executed. Note too that the control array may be any rank and shape, but that its elements are assigned to the control variable in ravel order.

The control array may contain any type of data. For example, the following code resizes (and compacts) all your component files

```
[1]   :For FILE :In (↓⎕FLIB '')~¨' '
[2]       FILE ⎕FTIE 1
[3]       ⎕FRESIZE 1
[4]       ⎕FUNTIE 1
[5]   :EndFor
```

You may also nest `:For` loops. For example, the following expression finds the timestamp of the most recently updated component in all your component files.

```
[1]   TS←0
[2]   :For FILE :In (↓⎕FLIB '')~¨' '
[3]       FILE ⎕FTIE 1
[4]       START END←2⍴⎕FSIZE 1
[5]       :For COMP :In (START-1)↓⍳END-1
[6]           TS⌈←¯1↑⎕FREAD FILE COMP
[7]       :EndFor
[8]       ⎕FUNTIE 1
[9]   :EndFor
```

Multiple Control Variables

The `:For` control structure can also take multiple variables. This has the effect of doing a strand assignment each time around the loop.

For example `:For a b c :in (1 2 3)(4 5 6)`, sets `a b c←1 2 3`, first time around the loop and `a b c←4 5 6`, the second time.

Another example is `:For i j :In ⍳⍴Matrix`, which sets `i` and `j` to each row and column index of `Matrix`.

:InEach Control Word

```
      :For var ... :InEach value ...
```

In a `:For` control structure, the keyword `:InEach` is an alternative to `:In`.

For a single control variable, the effect of the keywords is identical but for multiple control variables the values vector is inverted.

The distinction is best illustrated by the following equivalent examples:

```
      :For a b c :In (1 2 3)(3 4 5)(5 6 7)(7 8 9)
          ⎕←a b c
      :EndFor


      :For a b c :InEach (1 3 5 7)(2 4 6 8)(3 5 7 9)
          ⎕←a b c
      :EndFor
```

In each case, the output from the loop is:

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

Notice that in the second case, the number of items in the values vector is the same as the number of control variables. A more typical example might be.

```
      :For a b c :InEach avec bvec cvec
          ...
      :EndFor
```

Here, each time around the loop, control variable `a` is set to the next item of `avec`, `b` to the next item of `bvec` and `c` to the next item of `cvec`.

:For Statement

```
|
:For var :In[Each] aexp
|
code
|
:End[For]
|
```

Select Statement :Select aexp

A :Select structure is used to execute alternative blocks of code depending upon the value of an array. For example, the following displays 'I is 1' if the variable I has the value 1, 'I is 2' if it is 2,or 'I is neither 1 nor 2' if it has some other value.

```
[1]   :Select I
[2]   :Case 1
[3]       'I is 1'
[4]   :Case 2
[5]       'I is 2'
[6]   :Else
[7]       'I is neither 1 nor 2'
[8]   :EndSelect
```

In this case, the system compares the value of the array expression to the right of the :Select statement with each of the expressions to the right of the :Case statements and executes the block of code following the one that matches. If none match, it executes the code following the :Else (which is optional). Note that comparisons are performed using the ≡ primitive function, so the arrays must match exactly. Note also that not all of the :Case expressions are necessarily evaluated because the process stops as soon as a matching expression is found.

Instead of a :Case statement, you may also use a :CaseList statement. If so, the *enclose of* the array expression to the right of :Select is tested for membership of the array expression to the right of the :CaseList using the ∊ primitive function.

Example

```
[1]   :Select ?6 6
[2]   :Case 6 6
[3]       'Box Cars'
[4]   :Case 1 1
[5]       'Snake Eyes'
[6]   :CaseList 2⍴¨⍳6
[7]       'Pair'
[8]   :CaseList (⍳6),¨⌽⍳6
[9]       'Seven'
[10]  :Else
[11]      'Unlucky'
[12]  :EndSelect
```

:Select Statement

```
        |
        :Select aexp
        |
        |<----------------------------------------------.
        |                                               |
        .-------.-------.---------------.               |
        |       |       |               |               |
        |       :Else   :Case aexp      :CaseList aexp  |
        |       |       |               |               |
        |       |       |<--------------'               |
        |       |       |                               |
        |       code    code                            |
        |       |       |                               |
        |<------'       `-------------------------------'
        |
        :End[Select]
```

With Statement :With obj

`:With` is a control structure that may be used to simplify a series of references to an object or namespace. `:With` changes into the specified namespace for the duration of the control structure, and is terminated by `:End[With]`. For example, you could update several properties of a Grid object `F.G` as follows:

```
:With F.G
     Values←4 3⍴0
     RowTitles←'North' 'South' 'East' 'West'
     ColTitles←'Cakes' 'Buns' 'Biscuits'
:EndWith
```

`:With` is analogous to `⎕CS` in the following senses:

- The namespace argument to `:With` is interpreted relative to the current space.
- Local names in the containing defined function continue to be visible in the new space.
- Global references from within the `:With` control structure are to names in the new space.
- Exiting the defined function from within a `:With` control structure causes the space to revert to the one from which the function was called.

On leaving the `:With` control structure, execution reverts to the original namespace. Notice however that the interpreter does not detect branches `(→)` out of the control structure. `:With` control structures can be nested in the normal fashion:

```
[1]   :With 'x'           ⍝ Change to #.x
[2]       :With 'y'       ⍝ Change to #.x.y
[3]           :With ⎕SE   ⍝ Change to ⎕SE
[4]               ...     ⍝ ... in ⎕SE
[5]           :EndWith    ⍝ Back to #.x.y
[6]       :EndWith        ⍝ Back to #.x
[7]   :EndWith            ⍝ Back to #
```

:With Statement

```
|
:With namespace (ref or name)
|
code
|
:End[With]
|
```

Hold Statement :Hold tkns

Whenever more than one thread tries to access the same piece of data or shared resource at the same time, you need some type of synchronisation to control access to that data. This is provided by `:Hold`.

`:Hold` provides a mechanism to control thread entry into a critical section of code. `tkns` must be a simple character vector or scalar, or a vector of character vectors. `tkns` represents a set of 'tokens', all of which must be acquired before the thread can continue into the control structure. `:Hold` is analogous to the component file system `⎕FHOLD`.

Within the whole active workspace, a token with a particular value may be held only once. If the hold succeeds, the current thread *acquires* the tokens and execution continues with the first phrase in the control structure. On exit from the structure, the tokens are released for use by other threads. If the hold fails, because one or more of the tokens is already in use:

1. If there is no `:Else` clause in the control structure, execution of the thread is blocked until the requested tokens become available.

2. Otherwise, acquisition of the tokens is abandoned and execution resumed immediately at the first phrase in the `:Else` clause.

`tkns` can be either a single token:

```
      'a'
      'Red'
      '#.Util'
      ''
      'Program Files'
```

… or a number of tokens:

```
      'red' 'green' 'blue'
      'doe' 'a' 'deer'
      ,¨'abc'
      ↓⎕nl 9
```

Pre-processing removes trailing blanks from each token before comparison, so that, for example, the following two statements are equivalent:

```
      :Hold 'Red' 'Green'

      :Hold ↓2 5⍴'Red  Green'
```

Unlike `⎕FHOLD`, a thread does not release all existing tokens before attempting to acquire new ones. This enables the nesting of holds, which can be useful when multiple threads are concurrently updating parts of a complex data structure.

In the following example, a thread updates a critical structure in a child namespace, and then updates a structure in its parent space. The holds will allow all ‘sibling’ namespaces to update concurrently, but will constrain updates to the parent structure to be executed one at a time.

```
:Hold ⎕cs''            ⍝ Hold child space
    ...                ⍝ Update child space
    :Hold ##.⎕cs''     ⍝ Hold parent space
        ...            ⍝ Update Parent space
    :EndHold
    ...
:EndHold
```

However, with the nesting of holds comes the possibility of a ‘deadlock’. For example, consider the two threads:

Thread 1

```
:Hold 'red'
    ...
    :Hold 'green'
        ...
    :EndHold
:EndHold
```

Thread 2

```
:Hold 'green'
    ...
    :Hold 'red'
        ...
    :EndHold
:EndHold
```

In this case if both threads succeed in acquiring their first hold, they will both block waiting for the other to release its token. Fortunately, the interpreter detects such cases and issues an error `(1008) DEADLOCK`. You can avoid deadlock by ensuring that threads always attempt to acquire tokens in the same chronological order, and that threads never attempt to acquire tokens that they already own.

Note that token acquisition for any particular `:Hold` is atomic, that is, either *all* of the tokens or *none* of them are acquired. The following example *cannot* deadlock:

Thread 1

```
:Hold 'red'
    ...
    :Hold 'green'
        ...
    :EndHold
:EndHold
```

Thread 2

```
:Hold 'green' 'red'
    ...
:EndHold
```

Examples

`:Hold` could be used for example, during the update of a complex data structure that might take several lines of code. In this case, an appropriate value for the token would be the name of the data structure variable itself, although this is just a programming convention: the interpreter does not associate the token value with the data variable.

```
    :Hold'Struct'
        ...             ⍝ Update Struct
        Struct ← ...
    :EndHold
```

The next example guarantees exclusive use of the current namespace:

```
    :Hold ⎕CS''         ⍝ Hold current space
        ...
    :EndHold
```

The following example shows code that holds two positions in a vector while the contents are exchanged.

```
    :Hold ⍕¨to fm
        :If >/vec[fm to]
            vec[fm to]←vec[to fm]
        :End
    :End
```

Between obtaining the next available file tie number and using it:

```
    :Hold '⎕FNUMS'
        tie←1+⌈/0,⎕FNUMS
        fname ⎕FSTIE tie
    :End
```

The above hold is not necessary if the code is combined into a single line:

```
    fname ⎕FSTIE tie←1+⌈/0,⎕FNUMS
```

or,

```
    tie←fname ⎕FSTIE 0
```

Note that `:Hold`, like its component file system counterpart `⎕FHOLD`, is a device to enable *co-operating* threads to synchronise their operation.

`:Hold` does not *prevent* threads from updating the same data structures concurrently, it prevents threads only from `:Hold`-ing the same tokens.

:Hold Statement

```
      |
      :Hold token(s)
      |
code
|
|-------.
|       |
|       :Else
|       |
|       code
|       |
|<------.
|
:End[Hold]
|
```

Trap Statement :Trap ecode

`:Trap` is an error trapping mechanism that can be used in conjunction with, or as an alternative to, the `⎕TRAP` system variable. It is equivalent to APL2's `⎕EA`, except that the code to be executed is not restricted to a single expression and is not contained within quotes (and so is slightly more efficient).

Operation

The segment of code immediately following the `:Trap` keyword is executed. On completion of this segment, if no error occurs, control passes to the code following `:End[Trap]`.

If an error does occur, the event code (error number) is noted and:

- If the error occurred within a sub-function, the system cuts the execution stack back to the function containing the `:Trap` keyword. In this respect, `:Trap` behaves like `⎕TRAP` with a `'C'` qualifier.
- The system searches for a `:Case[List]` representing the event code.
- If there is such a `:Case[List]`, or failing that, an `:Else` keyword, execution continues from this point.

Otherwise, control passes to the code following `:End[Trap]` and no error processing occurs.

Note that the error trapping is in effect **only** during execution of the initial code segment. It is disabled (or surrendered to outer level `:Trap`s or `⎕TRAP`s) immediately a trapped error occurs. In particular, the error trap is no longer in effect during processing of `:Case[List]`'s argument or in the code following the `:Case[List]` or `:Else` statement. This avoids the situation sometimes encountered with `⎕TRAP` where an infinite 'trap loop' occurs. If an error which is not specified occurs, it is processed by outer `:Trap`s, `⎕TRAP`s, or default system processing in the normal fashion.

Examples

```
     ∇ lx
[1]    :Trap 1000        ⍝ Cutback and exit on interrupt
[2]        Main ...
[3]    :EndTrap
     ∇

     ∇ ftie←Fcreate file      ⍝ Create null component file
[1]    ftie←1+⌈/0,⎕fnums      ⍝ next tie number.
[2]    :Trap 22               ⍝ Trap FILE NAME ERROR
[3]        file ⎕fcreate ftie ⍝ Try to create file.
[4]    :Else
[5]        file ⎕ftie ftie    ⍝ Tie the file.
[6]        file ⎕ferase ftie  ⍝ Drop the file.
[7]        file ⎕fcreate ftie ⍝ Create new file.
[8]    :EndTrap
     ∇

     ∇ lx ⍝ Distinguish various cases
[1]    :Trap 0 1000
[2]        Main ...
[3]    :Case 1002
[4]        'Interrupted ...'
[5]    :CaseList 1 10 72 76
[6]        'Not enough resources'
[7]    :CaseList 17+⍳20
[8]        'File System Problem'
[9]    :Else
[10]       'Unexpected Error'
[11]   :EndTrap
     ∇
```

Note that `:Traps` can be nested:

```
     ∇ ntie←Ntie file                   ⍝ Tie native file
[1]    ntie←¯1+⌊/0,⎕nnums               ⍝ Next native tie num
[2]    :Trap 22                         ⍝ Trap FILE NAME ERROR
[3]        file ⎕ntie ntie              ⍝ Try to tie file
[4]    :Else
[5]        :Trap 22                     ⍝ Trap FILE NAME ERROR
[6]            (file,'.txt')⎕ntie ntie ⍝ Try with .txt extn
[7]        :Else
[8]            file ⎕ncreate ntie       ⍝ Create null file.
[9]        :EndTrap
[10]   :EndTrap
     ∇
```

:Trap Statement

```
|
:Trap <ecode>
|
code
|
|<-----------------------------------.
|                                    |
.-------.-------.                    |
|       |       |                    |
|       :Else   :Case[List] <ecode>  |
|       |       |                    |
|       |       |                    |
|       |       |                    |
|       code    code                 |
|       |       |                    |
|<------'       `--------------------'
|
:End[Trap]
|
```

Where `ecode` is a scalar or vector of `⎕TRAP` event codes (see *Chapter 6*).

Note that within the `:Trap` control structure, `:Case` is used for a single event code and `:CaseList` for a vector of event codes.

GoTo Statement :GoTo aexp

A `:GoTo` statement is a direct alternative to `→` (branch) and causes execution to jump to the line specified by the first element of `aexp`.

The following are equivalent. See Branch for further details.

```
      →Exit
      :GoTo Exit

      →(N<I←I+1)/End
      :GoTo (N<I←I+1)/End

      →1+⎕LC
      :GoTo 1+⎕LC

      →10
      :GoTo 10
```

Return Statement :Return

A `:Return` statement causes a function to terminate and has exactly the same effect as `→0`.

The `:Return` control word takes no argument.

A `:Return` statement may occur anywhere in a function or operator.

Leave Statement :Leave

A :Leave statement is used to explicitly terminate the execution of a block of statements within a `:For`, `:Repeat` or `:While` control structure.

The `:Leave` control word takes no argument.

Continue Statement :Continue

A `:Continue` statement starts the next iteration of the immediately surrounding `:For`, `:Repeat` or `:While` control loop.

When executed within a `:For` loop, the effect is to start the body of the loop with the next value of the iteration variable.

When executed within a `:Repeat` or `:While` loop, if there is a trailing test that test is executed and, if the result is true, the loop is terminated. Otherwise the leading test is executed in the normal fashion.

Triggers

Triggers provide the ability to have a function called automatically whenever a variable or a Field is assigned. Triggers are actioned by all forms of assignment (←), but only by assignment.

Triggers are designed to allow a class to perform some action when a field is modified – without having to turn the field into a property and use the property setter function to achieve this. Avoiding the use of a property allows the full use of the APL language to manipulate data in a field, without having to copy field data in and out of the class through get and set functions.

Triggers *can* also be applied to variables outside a class, and there will be situations where this is very useful. However, dynamically attaching and detaching a trigger from a variable is a little tricky at present.

The function that is called when a variable or Field changes is referred to as the *Trigger Function*. The name of a variable or Field which has an associated Trigger Function is termed a *Trigger*.

A function is declared as aTrigger function by including the statement:

```
      :Implements Trigger Name1,Name2,Name3, ...
```

where `Name1`, `Name2` etc are the Triggers.

When a Trigger function is invoked, it is passed an Instance of the internal Class `TriggerArguments`. This Class has 3 Fields:

Member	**Description**
`Name`	Name of the Trigger whose change in value has caused the Trigger Function to be invoked.
`NewValue`	The newly assigned value of the Trigger
`OldValue`	The previous value of the Trigger. If the Trigger was not previously defined, a reference to this Field causes a `VALUE ERROR`.

A Trigger Function is called *as soon as possible* after the value of a Trigger was assigned; typically by the end of the currently executing line of APL code. The precise timing is not guaranteed and may not be consistent because internal workspace management operations can occur at any time.

If the value of a Trigger is changed more than once by a line of code, the Trigger Function will be called at least once, but the number of times is not guaranteed.

A Trigger Function is not called when the Trigger is expunged.

Expunging a Trigger disconnects the name from the Trigger Function and the Trigger Function will not be invoked when the Trigger is reassigned. The connection may be re-established by re-fixing the Trigger Function.

A Trigger may have only a single Trigger Function. If the Trigger is named in more than one Trigger Function, the Trigger Function that was last fixed will apply.

In general, it is inadvisable for a Trigger function to modify its own Trigger, as this will potentially cause the Trigger to be invoked repeatedly and forever.

To associate a Trigger function with a *local* name, it is necessary to dynamically fix the Trigger function in the function in which the Trigger is localised; for example:

```
     ∇ TRIG arg
[1]    :Implements Trigger A
[2]    ...

     ∇ TEST;A
[1]    ⎕FX ⎕OR'TRIG'
[2]    A←10
[3]    ...
```

Example

The following function displays information when the value of variables `A` or `B` changes.

```
     ∇ TRIG arg
[1]    :Implements Trigger A,B
[2]    arg.Name'is now 'arg.NewValue
[3]    :Trap 6 ⍝ VALUE ERROR
[4]        arg.Name'was    'arg.OldValue
[5]    :Else
[6]        arg.Name' was     [undefined]'
[7]    :EndTrap
     ∇
```

Note that on the very first assignment to `A`, when the variable was previously undefined, `arg.OldValue` is a `VALUE ERROR`.

```
      A←10
 A  is now   10
 A   was     [undefined]

      A+←10
 A  is now   20
 A  was      10

      A←'Hello World'
 A  is now   Hello World
 A  was      20

      A[1]←⊂2 3⍴⍳6
 A  is now    1 2 3 ello World
              4 5 6
 A  was      Hello World

      B←⌽¨A
 B  is now    3 2 1 ello World
              6 5 4
 B   was     [undefined]

      A←⎕NEW MyClass
 A  is now   #.[Instance of MyClass]
 A  was       1 2 3 ello World
              4 5 6

      'F'⎕WC'Form'
      A←F
 A  is now   #.F
 A  was      #.[Instance of MyClass]
```

Note that Trigger functions are actioned only by assignment, so changing `A` to a Form using `⎕WC` does not invoke `TRIG`.

```
      'A'⎕WC'FORM' ⍝ Note that Trigger Function is not
invoked
```

However, the connection (between `A` and `TRIG`) remains and the Trigger Function will be invoked if and when the Trigger is re-assigned.

```
      A←99
 A  is now   99
 A  was      #.A
```

See Trigger Fields for information on how a Field (in a Class) may be used as a Trigger.

Idiom Recognition

Idioms are commonly used expressions that are recognised and evaluated internally, providing a significant performance improvement.

For example, the idiom `BV/⍳⍴A` (where `BV` is a Boolean vector and `A` is an array) would (in earlier Versions of Dyalog APL) have been evaluated in 3 steps as follows:

1. Evaluate `⍴A` and store result in temporary variable `temp1` (`temp1` is just an arbitrary name for the purposes of this explanation)
2. Evaluate `⍳temp1` and store result in temporary variable `temp2`.
3. Evaluate `BV/temp2`
4. Discard temporary variables

In the current Version of Dyalog APL, the expression is recognised in its entirety and processed in a single step as if it were a single primitive function. In this case, the resultant improvement in performance is between 2 and 4.5.

Idiom recognition is precise; an expression that is almost identical but not exactly identical to an expression given in the Idiom List table will not be recognised.

For example, `⎕AV⍳` will be recognised as an idiom, but `(⎕AV)⍳` will not. Similarly, `(,)/` would not be recognized as the Join idiom.

Idiom List

In the following table, arguments to the idiom have types and ranks as follows:

Type	Description	Rank	Description
C	Character	S	Scalar or 1-item vector
B	Boolean	V	Vector
N	Numeric	M	Matrix
P	Nested (pointer)	A	Array (any rank)
A	Any type		

For example: NV: numeric vector, CM: character matrix, PV: nested vector.

Expression	**Description**
`⍴⍴A`	Rank
`BV/⍳NS`	Sequence selection
`BV/⍳⍴A`	Index selection
`NA⊃¨⊂A`	Array selection
`A{}A`	Sink
`A{⍺}A`	Left (Lev)
`A{⍵}A`	Right (Dex)
`A{⍺ ⍵}A`	Link
`{0}A`	Zero
`{0}¨A`	Zero Each
`,/PV`	Join
`⊃⌽A`	Upper right item (`⎕ml<2`)
`↑⌽A`	Upper right item (`⎕ml≥2`)
`⊃⌽,A`	Lower right item (`⎕ml<2`)
`↑⌽,A`	Lower right item (`⎕ml≥2`)
`0=⍴V`	Zero shape
`0=⍴⍴A`	Zero rank
`0=≡A`	Zero depth
`⎕AV⍳CA`	Atomic vector index (**Classic Edition only**; use `⎕UCS`)
`M{(↓⍺)⍳↓⍵}M`	Matrix Iota
`↓⍉↑PV`	Nested vector transpose (`⎕ml<2`)
`↓⍉⊃PV`	Nested vector transpose (`⎕ml≥2`)
`^\' '=CA`	Mask of leading blanks.
`+/^\' '=CA`	Number of leading blanks
`+/^\BA`	Number of leading ones
`{(∨\' '≠⍵)/⍵}CV`	Trim leading blanks
`{(+/^\' '=⍵)↓⍵}CV`	Trim leading blanks

`~∘' '¨↓CA`	No-blank split
`{(+/∨\' '≠⌽⍵)↑¨↓⍵}CA`	No-trailing-blank split
`⊃∘⍴¨A`	Length of first axis of each sub-array (`⎕ml<2`)
`↑∘⍴¨A`	Length of first axis of each sub-array (`⎕ml≥2`)
`V,←A`	Catenate To

Notes

Sequence Selection `/⍳` and **Index Selection** `/⍳⍴`, as well as providing an execution time advantage, reduce intermediate workspace usage and consequently, the incidence of memory compactions and the likelihood of a WS FULL.

Array Selection `NV⊃¨⊂A`, is implemented as `A[NV]`, which is significantly faster. The two are equivalent but the former may now be used as a matter of taste with no performance penalty.

Join `,/` is currently special-cased only for vectors of vectors or scalars. Otherwise, the expression is evaluated as a series of concatenations. Recognition of this idiom turns **join** from an *n-squared* algorithm into a linear one. In other words, the improvement factor is proportional to the size of the argument vector.

Upper and **Lower Right Item** now take constant time. Without idiom recognition, the time taken depends linearly on the number of items in the argument.

Zero Depth `0=≡` takes a small constant time. Without idiom recognition, time taken would depend on the size and depth of the argument, which in the case of a deeply nested array, could be significant.

Nested vector transpose `↓⍉↑` is special-cased only for a vector of nested vectors, each of whose items is of the same length.

Matrix Iota `{(↓⍺)⍳↓⍵}`. As well as being quicker, the Matrix Iota idiom can accommodate much larger matrices. It is particularly effective when bound with a left argument using the compose operator:

```
      find←mat∘{(↓⍺)⍳↓⍵}     ⍝ find rows in mat table.
```

In this case, the internal hash table for `mat` is retained so that it does not need to be generated each time the monadic derived function `find` is applied to a matrix argument.

Trim leading blanks `{(∨\' '≠⍵)/⍵}` and `{(+/∧\' '=⍵)↓⍵}` are two codings of the same idiom. Both use the same C code for evaluation.

No-blank split `~∘' '¨↓` typically takes a character matrix argument and returns a vector of character vectors from which, all blanks have been removed. An example might be the character matrix of names returned by the system function `⎕NL`. In general, this idiom accommodates character arrays of any rank.

No-trailing-blank split `{(+/∨\' '≠⌽⍵)↑¨↓⍵}` typically takes a character matrix argument and returns a vector of character vectors. Any embedded blanks in each row are preserved but trailing blanks are removed. In general, this idiom accommodates character arrays of any rank.

Lengths `⊃∘⍴¨A` (`⎕ml<2`) or `↑∘⍴¨A` (`⎕ml>2`) avoids having to create an intermediate nested array of shape vectors.

For an array of vectors, this idiom quickly returns a *simple array* of the length of each vector.

```
      ⊃∘⍴¨ 'Hi' 'Pete' ⍝ Vector Lengths
2 4
```

For an array of matrices, it returns a simple array of the number of rows in each matrix.

```
      ⊃∘⍴¨⎕CR¨↓⎕NL 3   ⍝ Lines in functions
5 21...
```

Catenate To `V,←A` optimises the catenation of a scalar or a vector to a vector. This idiom optimises *repeated* catenation of a scalar or vector to an existing vector.

```
      props,←⊂ 'Posn' 0 0
      props,←⊂'Size' 50 50
      vector,←2+4
```

Note that the idiom is not applied if the value of vector `V` is shared with another symbol in the workspace, as illustated in the following examples:

In this first example, the idiom *is* used to perform the catenation to V1.

```
      V1←⍳10
      V1,←11
```

In the second example, the idiom *is not* used to perform the catenation to `V1`, because its value is at that point shared with `V2`.

```
      V1←⍳10
      V2←V1
      V1,←11
```

In the third example, the idiom *is not* used to perform the catenation to `V` in `Join[1]` because its value is at that point shared with the array used to call the function.

```
      ∇ V←V Join A
[1]     V,←A
      ∇
       (⍳10) Join 11
1 2 3 4 5 6 7 8 9 10 11
```

Search Functions and Hash Tables

Primitive dyadic *search* functions, such as `⍳` (index of) and `∊` (membership) have a *principal* argument in which items of the other *subject* argument are located.

In the case of `⍳`, the principal argument is the one on the left and in the case of `∊`, it is the one on the right. The following table shows the principal (P) and subject (s) arguments for each of the functions.

`P ⍳ s`	Index of
`s ∊ P`	Membership
`s ∩ P`	Intersection
`s ∪ P`	Union
`s ~ P`	Without
`P {(↓⍺)⍳↓⍵} s`	Matrix Iota (idiom)

The Dyalog APL implementation of these functions already uses a technique known as *hashing* to improve performance over a simple linear search. (Note that `⍷` (find) does not employ the same hashing technique, and is excluded from this discussion.)

Building a *hash table* for the principal argument takes a significant time but is rewarded by a considerably quicker search for each item in the subject. Unfortunately, the hash table is discarded each time the function completes and must be reconstructed for a subsequent call (even if its principal argument is identical to that in the previous one).

For optimal performance of *repeated* search operations, the hash table may be retained between calls, by binding the function with its principal argument using the primitive `∘` (compose) operator. The retained hash table is then used directly whenever this monadic derived function is applied to a subject argument.

Notice that retaining the hash table pays off only on a second or subsequent application of the derived function. This usually occurs in one of two ways: either the derived function is named for later (and repeated) use, as in the first example below or it is applied repeatedly as the operand of a primitive or defined operator, as in the second example.

Example: naming a derived function.

```
      words←'red' 'ylo' 'grn' 'brn' 'blu' 'pnk' 'blk'

      find←words∘⍳                     ⍝ monadic find function
      find'blk' 'blu' 'grn' 'ylo'      ⍝
7 5 3 2
      find'grn' 'brn' 'ylo' 'red'      ⍝ fast find
3 4 2 1
```

Example: repeated application by (¨) each operator.

```
      ∊∘⎕A¨'This' 'And' 'That'
 1 0 0 0  1 0 0  1 0 0 0
```

Locked Functions & Operators

A defined operation may be locked by the system function `⎕LOCK`. A locked operation may not be displayed or edited. The system function `⎕CR` returns an empty matrix of shape 0 0 and the system functions `⎕NR` and `⎕VR` return an empty vector for a locked operation.

Stop and trace vectors may be set by the system functions `⎕STOP` and `⎕TRACE` respectively. Existing stop or trace settings are cancelled when an operation is locked.

A locked operation may not be suspended, nor may a locked operation remain pendent when execution is suspended. The state indicator is cut back as described below.

The State Indicator

The state of execution is dynamically recorded in the STATE INDICATOR. The state indicator identifies the chain of execution for operators, functions and the evaluated or character input/output system variables (⎕ and ⍞). At the top of the state indicator is the most recently activated operation.

Execution may be suspended by an interrupt, induced by the user, the system, or by a signal induced by the system function ⎕SIGNAL or by a stop control set by the system function ⎕STOP. If the interrupt (or event which caused the interrupt) is not defined as a trappable event by the system variable ⎕TRAP, the state indicator is cut back to the first of either a defined operation or the evaluated input prompt (⎕) such that there is no locked defined operation in the state indicator. The topmost operation left in the state indicator is said to be SUSPENDED. Other operations in the chain of execution are said to be PENDENT.

The state indicator may be examined when execution is suspended by the system commands)SI and)SINL. The line numbers of defined operations in the state indicator are given also by the system variable ⎕LC.

Suspended execution may be resumed by use of the Branch function (see *Chapter 4*). Whilst execution is suspended, it is permitted to enter any APL expression for evaluation, thereby adding to the existing state indicator. Therefore, there may be more than one LEVEL OF SUSPENSION in the state indicator. If the state indicator is cut back when execution is suspended, it is cut back no further than the prior level of suspension (if any).

Examples

```
     ∇ F
[1]    G
     ∇

     ∇ G
[1]    'FUNCTION G'+
     ∇

      ⍎'F'
SYNTAX ERROR
G[1] 'FUNCTION G'+
     ^

      )SI
G[1]*
F[1]
⍎
```

```
      ⎕LOCK'G'

      ⍎'F'
SYNTAX ERROR
F[1] G
     ^

      )SI
F[1]*
⍎
G[1]*
F[1]
⍎
```

A suspended or pendent operation may be edited by the system editor or redefined using `⎕FX` provided that it is visible and unlocked. However, pendent operations retain their original definition until they complete, or are cleared from the State Indicator. When a new definition is applied, the state indicator is repaired if necessary to reflect changes to the operations, model syntax, local names, or labels.

Dynamic Functions & Operators

A Dynamic Function (operator) is an alternative function definition style suitable for defining small to medium sized functions. It bridges the gap between operator expressions: `rank←⍴∘⍴` and full 'header style' definitions such as:

```
    ∇ rslt←larg func rarg;local...
```

In its simplest form, a dynamic function is an APL expression enclosed in curly braces `{}` possibly including the special characters `⍺` and `⍵` to represent the left and right arguments of the function respectively. For example:

```
      {(+/⍵)÷⍴⍵} 1 2 3 4    ⍝ Arithmetic Mean (Average)
2.5
      3 {⍵*÷⍺} 64           ⍝ ⍺th root
4
```

Dynamic functions can be named in the normal fashion:

```
      mean←{(+/⍵)÷⍴⍵}
      mean¨(2 3)(4 5)
 2.5  4.5
```

Dynamic Functions can be defined and used in any context where an APL function may be found, in particular:

- In immediate execution mode as in the examples above.
- Within a defined function or operator.
- As the operand of an operator such as each (`¨`).
- Within another dynamic function.

The last point means that it is easy to define nested local functions.

Multi-Line Dynamic Functions

The single expression which provides the result of the Dynamic Function may be preceded by any number of assignment statements. Each such statement introduces a name which is local to the function.

For example in the following, the expressions `sum←` and `num←` create **local** variables `sum` and `num`.

```
mean←{          ⍝ Arithmetic mean
    sum←+/⍵     ⍝ Sum of elements
    num←⍴⍵      ⍝ Number of elements

    sum÷num     ⍝ Mean
}
```

Note that Dynamic Functions may be commented in the usual way using `⍝`.

When the interpreter encounters a local definition, a new local name is created. The name is shadowed dynamically exactly as if the assignment had been preceded by: `⎕shadow` *name* `⋄`.

It is **important** to note the distinction between the two types of statement above. There can be **many** assignment statements, each introducing a new local variable, but only a **single** expression where the result is not assigned. As soon as the interpreter encounters such an expression, it is evaluated and the result returned immediately as the result of the function.

For example, in the following,

```
mean←{          ⍝ Arithmetic mean
    sum←+/⍵     ⍝ Sum of elements
    num←⍴⍵      ⍝ Number of elements

    sum,num     ⍝ Attempt to show sum,num (wrong)!

    sum÷num     ⍝ ... and return result.
}
```

... as soon as the interpreter encounters the expression `sum,num`, the function terminates with the two element result (`sum,num`) and the following line is not evaluated.

To display arrays to the session from within a Dynamic function, you can use the explicit display forms ⎕← or ⍞← as in:

```
mean←{             ⍝ Arithmetic mean
    sum←+/⍵        ⍝ Sum of elements
    num←⍴⍵         ⍝ Number of elements

    ⎕←sum,num      ⍝ show sum,num.

    sum÷num        ⍝ ... and return result.
}
```

Note that local definitions can be used to specify local nested Dynamic Functions:

```
rms←{                      ⍝ Root Mean Square
    root←{⍵*0.5}           ⍝ ∇ Square root
    mean←{(+/⍵)÷⍴⍵}        ⍝ ∇ Mean
    square←{⍵×⍵}           ⍝ ∇ Square

    root mean square ⍵
}
```

Default Left Argument

The special syntax: `⍺←expr` is used to give a default value to the left argument if a Dynamic Function is called monadically. For example:

```
root←{        ⍝ ⍺th root
    ⍺←2       ⍝ default to sqrt
    ⍵*÷⍺
}
```

The expression to the right of `⍺←` is evaluated *only* if its Dynamic Function is called with no left argument.

Guards

A Guard is a Boolean-single valued expression followed on the right by a `':'`. For example:

```
0≡≡⍵:          ⍝ Right arg simple scalar

⍺<0:           ⍝ Left arg negative
```

The guard is followed by a single APL expression: the result of the function.

```
⍵≥0: ⍵*0.5     ⍝ Square root if non-negative.
```

A Dynamic function may contain any number of guarded expressions each on a separate line (or collected on the same line separated by diamonds). Guards are evaluated in turn until one of them yields a 1. The corresponding expression to the right of the guard is then evaluated as the result of the function.

If an expression occurs without a guard, it is evaluated immediately as the default result of the function. For example:

```
sign←{
    ⍵>0: '+ve'      ⍝ Positive
    ⍵=0: 'zero'     ⍝ zero
         '-ve'      ⍝ Negative (Default)
}
```

Local definitions and guards can be interleaved in any order.

Note again that any code following the first unguarded expression (which terminates the function) could never be executed and would therefore be redundant.

Shy Result

Dynamic Functions are usually 'pure' functions that take arguments and return explicit results. Occasionally, however, the main purpose of the function might be a side-effect such as the display of information in the session, or the updating of a file, and the value of a result, a secondary consideration. In such circumstances, you might want to make the result 'shy', so that it is discarded unless the calling context requires it. This can be achieved by assigning a dummy variable after a (true) guard:

```
log←{                     ⍝ Append ⍵ to file ⍺.
    tie←⍺ ⎕fstie 0        ⍝ tie number for file,
    cno←⍵ ⎕fappend tie    ⍝ new component number,
    tie←⎕funtie tie       ⍝ untie file,
    1:rslt←cno            ⍝ comp number as shy result.
}
```

Static Name Scope

When an inner (nested) Dynamic Function refers to a name, the interpreter searches for it by looking outwards through enclosing Dynamic Functions, rather than searching back along the execution stack. This regime, which is more appropriate for nested functions, is said to employ **static scope** instead of APL's usual **dynamic scope**. This distinction becomes apparent only if a call is made to a function defined at an outer level. For the more usual inward calls, the two systems are indistinguishable.

For example, in the following function, variable `type` is defined both within `which` itself and within the inner function `fn1`. When `fn1` calls outward to `fn2` and `fn2` refers to `type`, it finds the outer one (with value `'static'`) rather than the one defined in `fn1`:

```
      which←{

          type←'static'

          fn1←{
              type←'dynamic'
              fn2 ⍵
          }

          fn2←{
              type ⍵
          }

          fn1 ⍵
      }

      which'scope'
 static  scope
```

Tail Calls

A novel feature of the implementation of Dynamic Functions is the way in which tail calls are optimised.

When a Dynamic Function calls a sub-function, the result of the call may or may not be modified by the calling function before being returned. A call where the result is passed back immediately without modification is termed a tail call.

For example in the following, the first call on function `fact` is a tail call because the result of `fact` is the result of the whole expression, whereas the second call isn't because the result is subsequently multiplied by `⍵`.

```
      (⍺×⍵)fact ⍵-1        ⍝ Tail call on fact.

      ⍵×fact ⍵-1           ⍝ Embedded call on fact.
```

Tail calls occur frequently in Dynamic Functions, and the interpreter optimises them by re-using the current stack frame instead of creating a new one. This gives a significant saving in both time and workspace usage. It is easy to check whether a call is a tail call by tracing it. An embedded call will pop up a new trace window for the called function, whereas a tail call will re-use the current one.

Using tail calls can improve code performance considerably, although at first the technique might appear obscure. A simple way to think of a tail call is as a **branch with arguments**. The tail call, in effect, branches to the first line of the function after installing new values for ⍵ and ⍺.

Iterative algorithms can almost always be coded using tail calls.

In general, when coding a loop, we use the following steps, possibly in a different order depending on whether we want to test at the 'top' or the 'bottom' of the loop.

1.	Initialise loop control variable(s).	⍝ init
2.	Test loop control variable.	⍝ test
3.	Process body of loop.	⍝ proc
4.	Modify loop control variable for next iteration.	⍝ mod
5.	Branch to step 2.	⍝ jump

For example, in classical APL you might find the following:

```
    ∇ value←limit loop value    ⍝ init
[1]  top:→(⎕CT>value-limit)/0   ⍝ test
[2]   value←Next value          ⍝ proc, mod
[3]   →top                      ⍝ jump
    ∇
```

Control structures help us to package these steps:

```
    ∇ value←limit loop value    ⍝ init
[1]   :While ⎕CT<value-limit    ⍝ test
[2]       value←Next value      ⍝ proc, mod
[3]   :EndWhile                 ⍝ jump
    ∇
```

Using tail calls:

```
loop←{                          ⍝ init
    ⎕CT>⍺-⍵:⍵                   ⍝ test
    ⍺ ∇ Next ⍵                  ⍝ proc, mod, jump
}
```

Error-Guards

An **error-guard** is (an expression that evaluates to) a vector of error numbers, followed by the digraph: `::`, followed by an expression, the *body* of the guard, to be evaluated as the result of the function. For example:

```
      11 5 :: ω×0 ⍝ Trap DOMAIN and LENGTH errors.
```

In common with `:Trap` and `⎕TRAP`, error numbers 0 and 1000 are catchalls for synchronous errors and interrupts respectively.

When an error is generated, the system searches statically upwards and outwards for an error-guard that matches the error. If one is found, the execution environment is unwound to its state immediately *prior* to the error-guard's execution and the body of the error-guard is evaluated as the result of the function. This means that, during evaluation of the body, the guard is no longer in effect and so the danger of a hang caused by an infinite 'trap loop', is avoided.

Notice that you can provide 'cascading' error trapping in the following way:

```
0::try_2nd
0::try_1st
   expr
```

In this case, if `expr` generates an error, its immediately preceding: `0::` catches it and evaluates `try_1st` leaving the remaining error-guard in scope. If `try_1st` fails, the environment is unwound once again and `try_2nd` is evaluated, this time with no error-guards in scope.

Examples:

`Open` returns a handle for a component file. If the exclusive tie fails, it attempts a share-tie and if this fails, it creates a new file. Finally, if all else fails, a handle of 0 is returned.

```
open←{                      ⍝ Handle for component file ω.
    0::0                    ⍝ Fails:: return 0 handle.
    22::ω ⎕FCREATE 0        ⍝ FILE NAME:: create new one.
    24 25::ω ⎕FSTIE 0       ⍝ FILE TIED:: try share tie.
           ω ⎕FTIE 0        ⍝ Attempt to open file.
}
```

An error in `div` causes it to be called recursively with *improved* arguments.

```
div←{                       ⍝ Tolerant division:: ⍺÷0 → ⍺.
    ⍺←1                     ⍝ default numerator.
    5::↑∇/↓↑⍺ ⍵             ⍝ LENGTH:: stretch to fit.
    11::⍺ ∇ ⍵+⍵=0           ⍝ DOMAIN:: increase divisor.
    ⍺÷⍵                     ⍝ attempt division.
}
```

Notice that some arguments may cause `div` to recur twice:

```
      6 4 2 div 3 2
→     6 4 2 div 3 2 0
→     6 4 2 div 3 2 1
→     2 2 2
```

The final example shows the unwinding of the local environment before the error-guard's body is evaluated. Local name `trap` is set to describe the domain of its following error-guard. When an error occurs, the environment is unwound to expose `trap`'s statically correct value.

```
      add←{
          trap←'domain' ⋄ 11::trap
          trap←'length' ⋄  5::trap
          ⍺+⍵
      }

      2 add 3          ⍝ Addition succeeds
5

      2 add 'three'    ⍝ DOMAIN ERROR generated.
domain

      2 3 add 4 5 6    ⍝ LENGTH ERROR generated.
length
```

Dynamic Operators

The operator equivalent of a dynamic function is distinguished by the presence of either of the compound symbols '⍺⍺' or '⍵⍵' anywhere in its definition. ⍺⍺ and ⍵⍵ represent the left and right operand of the operator respectively.

Example

The following monadic `each` operator applies its function operand only to unique elements of its argument. It then distributes the result to match the original argument. This can deliver a performance improvement over the primitive each (¨) operator if the operand function is costly and the argument contains a significant number of duplicate elements. Note however, that if the operand function causes side effects, the operation of dynamic and primitive versions will be different.

```
      each←{                  ⍝ Fast each:
          shp←⍴⍵              ⍝ Shape and ...
          vec←,⍵              ⍝ ... ravel of arg.
          nub←∪vec            ⍝ Vector of unique elements.
          res←⍺⍺¨nub          ⍝ Result for unique elts.
          idx←nub⍳vec         ⍝ Indices of arg in nub ...
          shp⍴idx⊃¨⊂res       ⍝ ... distribute result.
      }
```

The dyadic `else` operator applies its left (else right) operand to its right argument depending on its left argument.

```
      else←{
          ⍺: ⍺⍺ ⍵     ⍝ True: apply Left operand
             ⍵⍵ ⍵     ⍝ Else,  ..   Right   ..
      }

      0 1 ⌈else⌊¨ 2.5         ⍝ Try both false and true.
2 3
```

Recursion

A recursive Dynamic Function can refer to itself using its name explicitly, but because we allow unnamed functions, we also need a special symbol for implicit self-reference: '∇'. For example:

```
      fact←{           ⍝ Factorial ⍵.
          ⍵≤1: 1       ⍝ Small ⍵, finished,
          ⍵×∇ ⍵-1      ⍝ Otherwise recur.
      }
```

Implicit self-reference using '∇' has the further advantage that it incurs less interpretative overhead and is therefore quicker. Tail calls using '∇' are particularly efficient.

Recursive Dynamic Operators refer to their derived functions, that is the operator bound with its operand(s) using ∇ or the operator itself using the compound symbol: ∇∇. The first form of self reference is by far the more frequently used.

```
      pow←{            ⍝ Function power.
          ⍺=0:⍵        ⍝ Apply function operand ⍺ times.
          (⍺-1)∇ ⍺⍺ ⍵ ⍝ ⍺⍺ ⍺⍺ ⍺⍺ ... ⍵
      }
```

The following example shows a rather contrived use of the second form of (operator) self reference. The exp operator composes its function operand with itself on each recursive call. This gives the effect of an exponential application of the original operand function:

```
      exp←{                ⍝ Exponential fn application.
          ⍺=0:⍺⍺ ⍵         ⍝ Apply operand 2*⍺ times.
          (⍺-1)⍺⍺∘⍺⍺ ∇∇ ⍵ ⍝ (⍺⍺∘⍺⍺)∘( ... ) ... ⍵
      }

      succ←{1+⍵}           ⍝ Successor (increment).

      10 succ exp 0
1024
```

Example: Pythagorean triples

The following sequence shows an example of combining Dynamic Functions and Operators in an attempt to find Pythagorean triples: `(3 4 5)(5 12 13)` ...

```
      sqrt←{⍵*0.5}                ⍝ Square root.

      sqrt 9 16 25
3 4 5

      hyp←{sqrt+/⊃⍵*2}            ⍝ Hypoteneuse of triangle.

      hyp(3 4)(4 5)(5 12)
5 6.403124237 13

      intg←{⍵=⌊⍵}                 ⍝ Whole number?

      intg 2.5 3 4.5
0 1 0

      pyth←{intg hyp ⍵}           ⍝ Pythagorean pair?

      pyth(3 4)(4 9)(5 12)
1 0 1

      pairs←{,⍳⍵ ⍵}               ⍝ Pairs of numbers 1..⍵.

      pairs 3
 1 1  1 2  1 3  2 1  2 2  2 3  3 1  3 2  3 3

      filter←{(⍺⍺ ⍵)/⍵}           ⍝ Op: ⍵ filtered by ⍺⍺.

      pyth filter pairs 12        ⍝ Pythagorean pairs 1..12
 3 4  4 3  5 12  6 8  8 6  9 12  12 5  12 9
```

So far, so good, but we have some duplicates: `(6 8)` is just double `(3 4)`.

```
      rpm←{                         ⍝ Relatively prime?
          ⍵=0:⍺=1                   ⍝ C.f. Euclid's gcd.
          ⍵ ∇ ⍵|⍺
      }/¨                           ⍝ Note the /¨

      rpm(2 4)(3 4)(6 8)(16 27)
0 1 0 1

      rpm filter pyth filter pairs 20
 3 4  4 3  5 12  8 15  12 5  15 8
```

We can use an operator to combine the tests:

```
      and←{                         ⍝ Lazy parallel 'And'.
          mask←⍺⍺ ⍵                 ⍝ Left predicate selects...
          mask\⍵⍵ mask/⍵            ⍝ args for right predicate.
      }

      pyth and rpm filter pairs 20
 3 4  4 3  5 12  8 15  12 5  15 8
```

Better, but we still have some duplicates: `(3 4) (4 3).`

```
      less←{</⊃⍵}
      less(3 4)(4 3)
1 0

      less and pyth and rpm filter pairs 40
 3 4  5 12  7 24  8 15  9 40  12 35  20 21
```

And finally, as promised, triples:

```
      {⍵,hyp ⍵}¨less and pyth and rpm filter pairs 35
 3 4 5  5 12 13  7 24 25  8 15 17  12 35 37  20 21 29
```

A Larger Example

Function `tokens` uses nested local D-Fns to split an APL expression into its constituent tokens. Note that all calls on the inner functions: `lex`, `acc`, and the unnamed D-Fn in each token case, are *tail calls*. In fact, the *only* stack calls are those on function: `all`, and the unnamed function: `{ω∨¯1⌽ω}`, within the ‘Char literal’ case.

```
    tokens←{                        ⍝ Lex of APL src line.
        alph←⎕A,⎕Á,'_∆⍙',26↑17↓⎕AV ⍝ Alphabet for names.
        all←{+/^\α∊ω}               ⍝ No. of leading α∊ω.
        acc←{(α,↑/ω)lex⊃↓/ω}        ⍝ Accumulate tokens.
        lex←{
            0=⍴ω:α ⋄ hd←↑ω          ⍝ Next char else done.

            hd=' ':α{               ⍝ White Space.
                size←ω all' '
                α acc size ω
            }ω

            hd∊alph:α{              ⍝ Name
                size←ω all alph,⎕D
                α acc size ω
            }ω

            hd∊'⎕:':α{              ⍝ System Name/Keyword
                size←ω all hd,alph
                α acc size ω
            }ω

            hd='''':α{              ⍝ Char literal
                size←+/^\{ω∨¯1⌽ω}≠\hd=ω
                α acc size ω
            }ω

            hd∊⎕D,'¯':α{            ⍝ Numeric literal
                size←ω all ⎕D,'.¯E'
                α acc size ω
            }ω

            hd='⍝':α acc(⍴ω)ω       ⍝ Comment
            α acc 1 ω               ⍝ Single char token.
        }
        (0⍴⊂'')lex,ω
    }

      display tokens'xtok←size↑srce ⍝ Next token'
.→-------------------------------------------------.
| .→---. .→. .→---. .→. .→---. .→-. .→-----------. |
| |xtok| |←| |size| |↑| |srce| |  | |⍝ Next token| |
| '----' '-' '----' '-' '----' '--' '------------' |
'∊-------------------------------------------------'
```

Restrictions

Currently **multi-line** Dynamic Functions can't be typed directly into the session. The interpreter attempts to evaluate the first line with its trailing left brace and a `SYNTAX ERROR` results.

Dynamic Functions need not return a result. However even a non-result-returning expression will terminate the function, so you can't, for example, call a non-result-returning function from the middle of a Dynamic Function.

You can trace a Dynamic Function **only** if it is defined on more than one line. Otherwise it is executed atomically in the same way as an execute (`⍎`) expression. This deliberate restriction is intended to avoid the confusion caused by tracing a line and seeing nothing change on the screen.

Dynamic Functions do not currently support `⎕cs`.

Supplied Workspaces

You can find more examples of dynamic functions and operators in workspaces in the **samples\dfns** directory.

DFNS.DWS - a selection of utility functions.

MIN.DWS - an example application.

APL Line Editor

The APL Line Editor described herein is included for completeness and for adherence to the ISO APL standard. See *User Guide* for a description of the more powerful full-screen editor, `⎕ED`.

Using the APL Line Editor, functions and operators are defined by entering Definition Mode. This mode is opened and closed by the del symbol , `∇`. Within this mode, all evaluation of input is deferred. The standard APL line editor (described below) is used to create and edit operations within definition mode.

Operations may also be defined using the system function `⎕FX` (implicit in a `⎕ED` fix) which acts upon the canonical (character), vector, nested or object representation form of an operation. (See *Chapter 6* and *User Guide* for details.)

Functions may also be created dynamically or by function assignment. (See above and *Chapter 4*.)

The line editor recognises three forms for the opening request.

Creating Defined Operation

The opening `∇` symbol is followed by the header line of a defined operation. Redundant blanks in the request are permitted except within names. If acceptable, the editor prompts for the first statement of the operation body with the line-number 1 enclosed in brackets. On successful completion of editing, the defined operation becomes the active definition in the workspace.

Example

```
      ∇R←FOO
[1]   R←10
[2]   ∇

      FOO
10
```

The given operation name must not have an active referent in the workspace, otherwise the system reports `defn error` and the system editor is not invoked:

```
      )VARS
SALES  X  Y

      ∇R←SALES Y
defn error
```

The header line of the operation must be syntactically correct, otherwise the system reports `defn error` and the system editor is not invoked:

```
      ∇R←A B C D:G
defn error
```

Listing Defined Operation

The `∇` symbol followed by the name of a defined operation and then by a closing `∇`, causes the display of the named operation. Omitting the function name causes the suspended operation (i.e. the one at the top of the state indicator) to be displayed and opened for editing.

Example

```
      ∇FOO∇
     ∇ R←FOO
[1]    R←10
     ∇

      )SI
FOO[1] *

      ∇
     ∇ R←FOO
[1]    R←10
[2]
```

Editing Active Defined Operation

Definition mode is entered by typing `∇` followed optionally by a name and editing directive.

The `∇` symbol on its own causes the suspended operation (i.e. the one at the top of the state indicator) to be displayed. The editor then prompts for a statement or editing directive with a line-number one greater than the highest line-number in the function. If the state indicator is empty, the system reports **`defn error`** and definition mode is not entered.

The `∇` symbol followed by the name of an active defined operation causes the display of the named operation. The editor then prompts for input as described above. If the name given is not the name of an active referent in the workspace, the opening request is taken to be the creation of a new operation as described in paragraph 1. If the name refers to a pendent operation, the editor issues the message `warning pendent operation` prior to displaying the operation. If the name refers to a locked operation, the system reports defn error and definition mode is not entered.

The `∇` symbol followed by the name of an active defined operation and an editing directive causes the operation to be opened for editing and the editing directive actioned. If the editing directive is invalid, it is ignored by the editor which then prompts with a line-number one greater than the highest line-number in the operation. If the name refers to a pendent operation, the editor issues the message `warning pendent operation` prior to actioning the editing directive. If the name refers to a locked operation, the system reports `defn error` and definition mode is not entered.

On successful completion of editing, the defined operation becomes the active definition in the workspace which may replace an existing version of the function. Monitors, and stop and trace vectors are removed.

Example

```
      ∇FOO[2]
[2]  R←R*2
[3]  ∇
```

Editing Directives

Editing directives, summarised in Figure 2(iv) are permitted as the first non-blank characters either after the operation name on opening definition mode for an active defined function, or after a line-number prompt.

Syntax	Description
`∇`	`Closes definition mode`
`[⎕]`	`Displays the entire operation`
`[⎕n]`	`Displays the operation starting at line n`
`[n⎕]`	`Displays only line n`
`[∆n]`	`Deletes line n`
`[n∆m]`	`Deletes m lines starting at line n`
`[n]`	`Prompts for input at line n`
`[n]s`	`Replaces or inserts a statement at line n`
`[n⎕m]`	`Edits line n placing the cursor at character position m`
	`Edit control symbols are:`
	`    /   - deletes character above`
	` 1 to 9 - inserts that number of spaces`
	` A to Z - inserts multiples of 5 spaces`
	` .text  - inserts the text prior to the character above '.'`
	` ,text  - inserts the text as above but continues the edit`

Figure 2(iv) : Editing directives

Line Numbers

Line numbers are associated with lines in the operation. Initially, numbers are assigned as consecutive integers, beginning with `[0]` for the header line. The number associated with an operation line remains the same for the duration of the definition mode unless altered by editing directives. Additional lines may be inserted by decimal numbering. Up to three places of decimal are permitted. On closing definition mode, operation lines are re-numbered as consecutive integers.

The editor always prompts with a line number. The response may be a statement line or an editing directive. A statement line replaces the existing line (if there is one) or becomes an additional line in the operation:

```
      ∇R←A PLUS B
[1]  R←A+B
[2]
```

Position

The editing directive `[n]`, where n is a line number, causes the editor to prompt for input at that line number. A statement or another editing directive may be entered. If a statement is entered, the next line number to be prompted is the previous number incremented by a unit of the display form of the last decimal digit. Trailing zeros are not displayed in the fractional part of a line number:

```
[2]   [0.8]
[0.8] ⍝ MONADIC OR DYADIC +
[0.9] ⍝ A ←→ OPTIONAL ARGUMENT
[1]
```

The editing directive `[n]`s, where `n` is a line number and s is a statement, causes the statement to replace the current contents of line `n`, or to insert line n if there is none:

```
[1] [0] R←{A} PLUS B
[1]
```

Delete

The editing directive `[∆n]`, where `n` is a line number, causes the statement line to be deleted. The form `[n∆m]`, where `n` is a line number and `m` is a positive integer, causes `m` consecutive statement lines starting from line number `n` to be deleted.

Edit

The editing directive `[n⎕m]`, where `n` is a line number and `m` is an integer number, causes line number `n` to be displayed and the cursor placed beneath the `m`{th} character on a new line for editing. The response is taken to be edit control symbols selected from:

`/`	- to delete the character immediately above the symbol.
1 to 9	- to insert from 1 to 9 spaces immediately prior to the character above the digit.
A to Z	- to insert multiples of 5 spaces immediately prior to the character above the letter, where A = 5, B = 10, C = 15 and so forth.
`,`	- to insert the text after the comma, including explicitly entered trailing spaces, prior to the character above the comma, and then re-display the line for further editing with the text inserted and any preceding deletions or space insertions also effected.
`.`	- to insert the text after the comma, including explicitly entered trailing spaces, prior to the character above the comma, and then complete the edit of the line with the text inserted and any preceding deletions or space insertions also effected.

Invalid edit symbols are ignored. If there are no valid edit symbols entered, or if there are only deletion or space insertion symbols, the statement line is re-displayed with characters deleted and spaces inserted as specified. The cursor is placed at the first inserted space position or at the end of the line if none. Characters may be added to the line which is then interpreted as seen.

The line number may be edited.

Examples

```
[1]    [1⎕7]
[1]    R←A+B
      ,→(0=⎕NC'A')ρ1↓⎕LC ⋄
[1]    →(0=⎕NC'A')ρ1↓⎕LC ⋄ R←A+B
                            .⋄→END
[2]    R←B
[3]    END:
[4]
```

The form `[n⎕0]` causes the line number `n` to be displayed and the cursor to be positioned at the end of the displayed line, omitting the edit phase.

Display

The editing directive `[⎕]`causes the entire operation to be displayed. The form `[⎕n]` causes all lines from line number n to be displayed. The form `[n⎕]` causes only line number `n` to be displayed:

```
[4]    [0⎕]
[0]    R←{A} PLUS B
[0]
[0]    [⎕]
[0]    R←{A} PLUS B
[0.1] ⍝ MONADIC OR DYADIC +
[1]    →(0=⎕NC'A')ρ1+⎕LC ⋄ R←A+B ⋄→END
[2]    R←B
[3]    'END:
[4]
```

Close Definition Mode

The editing directive `∇` causes definition mode to be closed. The new definition of the operation becomes the active version in the workspace. If the name in the operation header (which may or may not be the name used to enter definition mode) refers to a pendent operation, the editor issues the message `warning pendent operation` before exiting. The new definition becomes the active version, but the original one will continue to be referenced until the operation completes or is cleared from the State Indicator.

If the name in the operation header is the name of a visible variable or label, the editor reports `defn error` and remains in definition mode. It is then necessary to edit the header line or quit.

If the header line is changed such that it is syntactically incorrect, the system reports `defn error`, and re-displays the line leaving the cursor beyond the end of the text on the line. Backspace/linefeed editing may be used to alter or cancel the change:

```
[3]   [0⎕]             - display line 0
[0]   R←{A} PLUS B
[0]   R←{A} PLUS B:G;H - put syntax error in line 0
defn error
[0]   R←{A} PLUS B:G;H - line redisplayed
                  ;G;H - backspace/linefeed editing
[1]
```

Local names may be repeated. However, the line editor reports warning messages as follows:

1. If a name is repeated in the header line, the system reports "warning duplicate name" immediately.

2. If a label has the same name as a name in the header line, the system reports "warning label name present in line 0" on closing definition mode.

3. If a label has the same name as another label, the system reports "warning duplicate label" on closing definition mode.

Improper syntax in expressions within statement lines of the function is not detected by the system editor with the following exceptions:

- If the number of opening parentheses in each entire expression does not equal the number of closing parentheses, the system reports "warning unmatched parentheses", but accepts the line.
- If the number of opening brackets in each entire expression does not equal the number of closing brackets, the system reports "warning unmatched brackets", but accepts the line.

These errors are not detected if they occur in a comment or within quotes. Other syntactical errors in statement lines will remain undetected until the operation is executed.

Example

```
[4]   R←(A[;1)=2)⌿⍎EXP,'×2
warning unmatched parentheses
warning unmatched brackets
[5]
```

Note that there is an imbalance in the number of quotes. This will result an a `SYNTAX ERROR` when this operation is executed.

Quit Definition Mode

The user may quit definition mode by typing the INTERRUPT character. The active version of the operation (if any) remains unchanged.

CHAPTER 3

Object Oriented Programing

Introducing Classes

A Class is a blueprint from which one or more *Instances* of the Class can be created (instances are sometimes also referred to as *Objects).*

A Class may optionally derive from another Class, which is referred to as its Base Class.

A Class may contain *Methods*, *Properties* and *Fields* (commonly referred to together as *Members*) which are defined within the body of the class script or are inherited from other Classes. This version of Dyalog APL does not support *Events* although it is intended that these will be supported in a future release. However, Classes that are derived from .Net types may generate events using `4 ⎕NQ`.

A Class that is defined to derive from another Class automatically acquires the set of Properties, Methods and Fields that are defined by its Base Class. This mechanism is described as inheritance.

A Class may extend the functionality of its Base Class by adding new Properties, Methods and Fields or by substituting those in the Base Class by providing new versions with the same names as those in the Base Class.

Members may be defined to be Private or Public. A Public member may be used or accessed from outside the Class or an Instance of the Class. A Private member is internal to the Class and (in general) may not be referenced from outside.

Although Classes are generally used as blueprints for the creation of instances, a class can have Shared members which can be used without first creating an instance

Defining Classes

A Class is defined by a script that may be entered and changed using the editor. A class script may also be constructed from a vector of character vectors, and fixed using `⎕FIX`.

A class script begins with a `:Class` statement and ends with a `:EndClass` statement.

For example, using the editor:

```
      )CLEAR
clear ws
      )ED ○Animal
```

[an edit window opens containing the following skeleton Class script →]

```
:Class Animal
:EndClass
```

[the user edits and fixes the Class script]

```
      )CLASSES
Animal
      ⎕NC⊂'Animal'
9.4
```

Editing Classes

Between the `:Class` and `:EndClass` statements, you may insert any number of function bodies, Property definitions, and other elements. When you fix the Class Script from the editor, these items will be fixed inside the Class namespace.

Note that the contents of the Class Script defines the Class *in its entirety*. You may not add or alter functions by editing them independently and you may not add variables by assignment or remove objects with `⎕EX`.

When you *re-fix* a Class Script using the Editor or with `⎕FIX`, the original Class is discarded and the new definition, as specified by the Script, replaces the old one in its entirety.

Note:

Associated with a Class (or an instance of a class) there is a completely separate namespace which *surrounds* the class and can contain functions, variables and so forth that are created by actions external to the class.

For example, if `X` is *not* a public member of the class `MyClass`, then the following expression will insert a variable `X` into the namespace which surrounds the class:

```
      MyClass.X←99
```

The namespace is analogous to the namespace associated with a GUI object and will be re-initialised (emptied) whenever the Class is re-fixed. Objects in this parallel namespace are not visible from inside the Class or an Instance of the Class.

Inheritance

If you want a Class to derive from another Class, you simply add the name of that Class to the `:Class` statement using colon+space as a separator.

The following example specifies that `CLASS2` derives from `CLASS1`.

```
:Class CLASS2: CLASS1
:EndClass
```

Note that `CLASS1` is referred to as the *Base Class* of `CLASS2`.

If a Class has a Base Class, it automatically acquires all of the **Public** Properties, Methods and Fields defined for its Base Class unless it replaces them with its own members of the same name. This principle of inheritance applies throughout the Class hierarchy. Note that **Private** members are **not** subject to inheritance.

Warning: When a class is fixed, it keeps a reference (a pointer) to its base class. If the global name of the base class is expunged, the derived class will still have the base class reference, and the base class will therefore be kept *alive* in the workspace. The derived class will be fully functional, but attempts to edit it will fail when it attempts to locate the base class as the new definition is fixed.

At this point, if a new class with the original base class name is created, the derived class has no way of detecting this, and it will continue to use the *old and invisible* version of the base class. Only when the derived class is refixed, will the new base class be detected.

If you edit, refix or copy an existing base class, APL will take care to patch up the references, but if the base class is expunged first and recreated later, APL is unable to detect the substitution. You can recover from this situation by editing or refixing the derived class(es) after the base class has been substituted.

Classes that derive from .Net Types

You may define a Class that derives from any of the .Net Types by specifying the name of the .Net Type and including a `:USING` statement that provides a path to the .Net Assembly in which the .Net Type is located.

Example

```
:Class APLGreg: GregorianCalendar
:Using System.Globalization
...
:EndClass
```

Classes that derive from the Dyalog GUI

You may define a Class that derives from any of the Dyalog APL GUI objects by specifying the *name* of the Dyalog APL GUI Class in quotes.

For example, to define a Class named `Duck` that derives from a `Poly` object, the Class specification would be:

```
:Class Duck:'Poly'
:EndClass
```

The Base Constructor for such a Class is the `⎕WC` system function.

For further details see Writing Classes Based on the Dyalog GUI.

Instances

A Class is generally used as a blueprint or model from which one or more Instances of the Class are constructed. Note however that a class can have Shared members which can be used directly without first creating an instance.

You create an instance of a Class using the `⎕NEW` system function which is monadic.

The 1-or 2-item argument to `⎕NEW` contains a reference to the Class and, optionally, arguments for its Constructor function.

When `⎕NEW` executes, it first creates an empty instance namespace and tags it with an internal pointer to its Class.

When `⎕NEW` executes, it creates a regular APL namespace to contain the Instance, and within that it creates an Instance space, which is populated with any Instance Fields defined by the class (with default values if specified), and pointers to the Instance Method and Property definitions specified by the Class.

If a monadic Constructor is defined, it is called with the arguments specified in the second item of the argument to `⎕NEW`. If `⎕NEW` was called without Constructor arguments, and the class has a niladic Constructor, this is called instead.

The Constructor function is typically used to initialise the instance and may establish variables in the instance namespace.

The result of `⎕NEW` is a reference to the instance namespace. Instances of Classes exhibit the same set of Properties, Methods and Fields that are defined for the Class.

Constructors

A Constructor is a special function defined in the Class script that is to be run when an Instance of the Class is created by `⎕NEW`. Typically, the job of a Constructor is to initialise the new Instance in some way.

A Constructor is identified by a `:Implements Constructor` statement. This statement may appear anywhere in the body of the function after the function header. The significance of this is discussed below.

Note that it is also *essential* to define the Constructor to be *Public*, with a `:Access Public` statement, because like all Class members, Constructors default to being *Private*. Private Constructors currently have no use or purpose, but It is intended that they will be supported in a future release of Dyalog APL.

A Constructor function may be niladic or monadic and must not return a result.

A Class may specify any number of different Constructors of which one (and only one) may be niladic. This is also referred to as the *default* Constructor.

There may be any number of monadic Constructors, but each must have a differently defined argument list which specifies the number of items expected in the Constructor argument. See Constructor Overloading for details.

A Constructor function may not call another Constructor function and a constructor function may not be called directly from outside the Class. The only way a Constructor function may be invoked is by `⎕NEW`. See Base Constructors for further details.

When `⎕NEW` is executed *with a 2-item argument,* the appropriate monadic Constructor is called with the second item of the `⎕NEW` argument.

The niladic (default) Constructor is called when `⎕NEW` is executed with a 1-item argument, a Class reference alone, or whenever APL needs to create a fill item for the Class.

Note that `⎕NEW` first creates a new instance of the specified Class, and then executes the Constructor inside the instance.

Example

The `DomesticParrot` Class defines a Constructor function `egg` that initialises the Instance by storing its name (supplied as the 2nd item of the argument to `⎕NEW`) in a Public Field called `Name`.

```
:Class DomesticParrot:Parrot
    :Field Public Name

    ∇ egg name
      :Implements Constructor
      :Access Public
      Name←name
    ∇
    ...
:EndClass ⍝ DomesticParrot

      pol←⎕NEW DomesticParrot 'Polly'
      pol.Name
Polly
```

Constructor Overloading

NameList header syntax is used to define different versions of a Constructor each with a different number of parameters, referred to as its *signature*. The Clover Class illustrates this principle.

In deciding which Constructor to call, APL matches the shape of the Constructor argument with the signature of each of the Constructors that are defined. If a constructor with the same number of arguments exists (remembering that 0 arguments will match a niladic Constructor), it is called. If there is no exact match, and there is a Constructor with a general signature (an un-parenthesised right argument), it is called. If no suitable constructor is found, a `LENGTH ERROR` is reported.

There may be one and only one constructor with a particular signature.

A Constructor function may not call another Constructor function and a constructor function may not be called directly from outside the Class. The only way a Constructor function may be invoked is by `⎕NEW`. See Base Constructors for further details.

In the Clover Class example Class, the following Constructors are defined:

Constructor	**Implied argument**
`Make1`	1-item vector
`Make2`	2-item vector
`Make3`	3-item vector
`Make0`	No argument
`MakeAny`	Any array accepted

Clover Class Example

```
:Class Clover ⍝ Constructor Overload Example
    :Field Public Con
    ∇ Make0
      :Access Public
      :Implements Constructor
      make 0
    ∇
    ∇ Make1(arg)
      :Access Public
      :Implements Constructor
      make arg
    ∇
    ∇ Make2(arg1 arg2)
      :Access Public
      :Implements Constructor
      make arg1 arg2
    ∇
    ∇ Make3(arg1 arg2 arg3)
      :Access Public
      :Implements Constructor
      make arg1 arg2 arg3
    ∇
    ∇ MakeAny args
      :Access Public
      :Implements Constructor
      make args
    ∇
    ∇ make args
      Con←(⍴args)(2⊃⎕SI)args
    ∇
:EndClass ⍝ Clover
```

In the following examples, the `Make` function (see Clover Class listing for details) displays:

```
<shape of argument> <name of Constructor called><argument>
(see function Make)
```

Creating a new Instance of Clover with a 1-element vector as the Constructor argument, causes the system to choose the `Make1` Constructor.Note that, although the argument to `Make1` is a 1-element vector, this is disclosed as the list of arguments is unpacked into the (single) variable `arg1`.

```
      (⎕NEW Clover(,1)).Con
   Make1  1
```

Creating a new Instance of Clover with a 2- or 3-element vector as the Constructor argument causes the system to choose `Make2`, or `Make3` respectively.

```
      (⎕NEW Clover(1 2)).Con
 2  Make2  1 2
      (⎕NEW Clover(1 2 3)).Con
 3  Make3  1 2 3
```

Creating an Instance with any other Constructor argument causes the system to choose `MakeAny`.

```
      (⎕NEW Clover(⍳10)).Con
 10  MakeAny  1 2 3 4 5 6 7 8 9 10
      (⎕NEW Clover(2 2⍴⍳4)).Con
 2 2  MakeAny  1 2
               3 4
```

Note that a scalar argument will call `MakeAny` and not `Make1`.

```
      (⎕NEW Clover 1).Con
   MakeAny  1
```

and finally, creating an Instance without a Constructor argument causes the system to choose `Make0`.

```
      (⎕NEW Clover).Con
   Make0  0
```

Niladic (Default) Constructors

A Class may define a niladic Constructor and/or one or more Monadic Constructors. The niladic Constructor acts as the default Constructor that is used when `⎕NEW` is invoked without arguments and when APL needs a fill item.

```
:Class Bird
    :Field Public Species

    ∇ egg spec
      :Access Public Instance
      :Implements Constructor
      Species←spec
    ∇
    ∇ default
      :Access Public Instance
      :Implements Constructor
      Species←'Default Bird'
    ∇
    ∇ R←Speak
      :Access Public
      R←'Tweet, tweet!'
    ∇

:EndClass ⍝ Bird
```

The niladic Constructor (in this example, the function `default`) is invoked when `⎕NEW` is called without Constructor arguments. In this case, the Instance created is no different to one created by the monadic Constructor `egg`, except that the value of the `Species` Field is set to `'Default Bird'`.

```
      Birdy←⎕NEW Bird
      Birdy.Species
Default Bird
```

The niladic Constructor is also used when APL needs to make a fill item of the Class. For example, in the expression `(3↑Birdy)`, APL has to create two fill items of `Birdy` (one for each of the elements required to pad the array to length 3) and will in fact call the niladic Constructor twice.

In the following statement:

```
      TweetyPie←3⊃10↑Birdy
```

The `10↑` (temporarily) ceates a 10-element array comprising the single entity `Birdy` padded with 9 fill-elements of Class `Bird`. To obtain the 9 fill-elements, APL calls the niladic Constructor 9 times, one for each separate prototypical Instance that it is required to make.

```
      TweetyPie.Species
Default Bird
```

Empty Arrays of Instances: Why ?

In APL it is natural to use *arrays* of Instances. For example, consider the following example.

```
:Class Cheese
    :Field Public Name←''
    :Field Public Strength←θ
    ∇ make2(name strength)
      :Access Public
      :Implements Constructor
      Name Strength←name strength
    ∇
    ∇ make1 name
      :Access Public
      :Implements Constructor
      Name Strength←name 1
    ∇
    ∇ make_excuse
      :Access Public
      :Implements Constructor
      ⎕←'The cat ate the last one!'
    ∇
:EndClass
```

We might create an array of Instances of the Cheese Class as follows:

```
      cdata←('Camembert' 5)('Caephilly' 2) 'Mild Cheddar'
      cheeses←{⎕NEW Cheese ⍵}¨cdata
```

Suppose we want a range of medium-strength cheese for our cheese board.

```
      board←(cheeses.Strength<3)/cheeses
      board.Name
 Caephilly  Mild Cheddar
```

But look what happens when we try to select really strong cheese:

```
      board←(cheeses.Strength>5)/cheeses
      board.Name
The cat ate the last one!
```

Note that this message is not the result of the expression, but was explicitly displayed by the `make_excuse` function. The clue to this behaviour is the shape of `board`; it is empty!

```
      ⍴board
0
```

When a reference is made to an empty array of Instances (strictly speaking, a reference that requires a *prototype*), APL creates a new Instance by calling the *niladic* (default) Constructor, uses the new Instance to satisfy the reference, and then discards it. Hence, in this example, the reference:

```
      board.Name
```

caused APL to run the *niladic* Constructor `make_excuse`, which displayed:

```
The cat ate the last one!
```

Notice that the behaviour of empty arrays of Instances is modelled VERY closely after the behaviour of empty arrays in general. In particular, the Class designer is given the task of deciding what the type of the members of the prototype are.

Empty Arrays of Instances: How?

To cater for the need to handle empty arrays of Instances as easily as non-empty arrays, a reference to an empty array of Class Instances is handled in a special way.

Whenever a reference or an assignment is made to the content of an *empty array of Instances*, the following steps are performed:

1. APL creates a *new Instance* of the same Class of which the empty Instance belongs.
2. the default (niladic) Constructor is run in the new Instance
3. the appropriate value is obtained or assigned:
    a. if it is a reference is to a Field, the value of the Field is obtained
    b. if it is a reference is to a Property, the PropertyGet function is run
    c. if it is a reference is to a Method, the method is executed
    d. if it is an assignment, the assignment is performed or the PropertySet function is run
4. if it is a reference, the result of step 3 is used to generate an empty result array with a suitable prototype by the application of the function `{0⍴⊂⍵}` to it
5. the Class Destructor (if any) is run in the new Instance
6. the New Instance is deleted

Example

```
:Class Bird
    :Field Public Species

    ∇ egg spec
      :Access Public Instance
      :Implements Constructor
      ⎕DF Species←spec
    ∇
    ∇ default
      :Access Public Instance
      :Implements Constructor
      ⎕DF Species←'Default Bird'
      #.DISPLAY Species
    ∇
    ∇ R←Speak
      :Access Public
      #.DISPLAY R←'Tweet, Tweet, Tweet'
    ∇

:EndClass ⍝ Bird
```

First, we can create an empty array of Instances of Bird using `0⍴`.

```
      Empty←0⍴⎕NEW Bird 'Robin'
```

A reference to `Empty.Species` causes APL to create a new Instance and invoke the niladic Constructor `default`. This function sets `Species` to `'Default Bird'` *and* calls `#.DISPLAY` which displays output to the Session.

```
      DISPLAY Empty.Species
.→-----------.
|Default Bird|
'------------'
```

APL then retrieves the value of `Species` (`'Default Bird'`), applies the function `{0⍴⊂⍵}` to it and returns this as the result of the expression.

```
.⊖---------------.
| .→-----------. |
| |            | |
| '------------' |
'∊---------------'
```

A reference to `Empty.Speak` causes APL to create a new Instance and invoke the niladic Constructor `default`. This function sets `Species` to `'Default Bird'` *and* calls `#.DISPLAY` which displays output to the Session.

```
      DISPLAY Empty.Speak
.→-----------.
|Default Bird|
'------------'
```

APL then involes function `Speak` which displays `'Tweet, Tweet, Tweet'` and returns this as the result of the function.

```
.→-----------------.
|Tweet, Tweet, Tweet|
'-------------------'
```

APL then applies the function `{0ρ⊂ω}` to it and returns this as the result of the expression.

```
.⊖---------------------.
| .→-----------------. |
| |                  | |
| '------------------' |
'∊---------------------'
```

Base Constructors

Constructors in a Class hierarchy, are not inherited in the same way as other members. However, there is a mechanism for all the Classes in the Class inheritance tree to participate in the initialisation of an Instance.

Every Constructor function contains a `:Implements Constructor` statement which may appear anywhere in the function body. The statement may optionally be followed by the `:Base` control word and an arbitrary expression.

The statement:

```
      :Implements Constructor :Base expr
```

calls *a monadic* Constructor in the Base Class. The choice of Constructor depends upon the rank and shape of the result of `expr` (see Constructor Overloading for details).

Whereas, the statement:

```
      :Implements Constructor
```
or
```
      :Implements Constructor :Base
```

calls *the niladic* Constructor in the Base Class.

Note that during the instantiation of an Instance, these calls potentially takes place in every Class in the Class hierarchy.

If, anywhere down the hierarchy, there is a *monadic* call and there is no matching monadic Constructor, the operation fails with a `LENGTH ERROR`.

If there is a *niladic* call on a Class that defines **no Constructors**, the niladic call is simply repeated in the next Class along the hierarchy.

However, if a Class defines a monadic Constructor and no niladic Constructor it implies that that Class **cannot be instantiated without Constructor arguments**. Therefore, if there is a call to a niladic Constructor in such a Class, the operation fails with a `LENGTH ERROR`. Note that it is therefore impossible for APL to instantiate a fill item or process a reference to an empty array for such a Class or any Class that is based upon it.

A Constructor function may not call another Constructor function and a constructor function may not be called directly from outside the Class or Instance. The only way a Constructor function may be invoked is by `⎕NEW`. The fundamental reason for these restrictions is that there must be one and only one call on the Base Constructor when a new Instance is instantiated. If Constructor functions were allowed to call one another, there would be several calls on the Base Constructor. Similarly, if a Constructor could be called directly it would potentially duplicate the Bse Constructor call.

Niladic Example

In the following example, `DomesticParrot` is derived from `Parrot` which is derived from `Bird`. They all share the Field `Desc` (inherited from `Bird`). Each of the 3 Classes has its own *niladic* Constructor called `egg0`.

```
:Class Bird
    :Field Public Desc
    ∇ egg0
      :Access Public
      :Implements Constructor
      Desc←'Bird'
    ∇
:EndClass ⍝ Bird

:Class Parrot: Bird
    ∇ egg0
      :Access Public
      :Implements Constructor
      Desc,←'→Parrot'
    ∇
:EndClass ⍝ Parrot

:Class DomesticParrot: Parrot
    ∇ egg0
      :Access Public
      :Implements Constructor
      Desc,←'→DomesticParrot'
    ∇
:EndClass ⍝ DomesticParrot

      (⎕NEW DomesticParrot).Desc
Bird→Parrot→DomesticParrot
```

Explanation

`⎕NEW` creates the new instance and runs the niladic Constructor `DomesticParrot.egg0`. As soon as the line:

```
:Implements Constructor
```

is encountered, `⎕NEW` calls the niladic constructor in the Base Class `Parrot.egg0`

`Parrot.egg0` starts to execute and as soon as the line:

```
:Implements Constructor
```

is encountered, `⎕NEW` calls the niladic constructor in the Base Class `Bird.egg0`.

When the line:

```
:Implements Constructor
```

is encountered, `⎕NEW` cannot call the niladic constructor in the Base Class (there is none) so the chain of Constructors ends. Then, as the State Indicator unwinds →

`Bird.egg0`	executes	`Desc←'Bird''`
`Parrot.egg0`	executes	`Desc,←'→Parrot''`
`DomesticParrot.egg0`	execute	`Desc,←'→DomesticParrot''`

Monadic Example

In the following example, `DomesticParrot` is derived from `Parrot` which is derived from `Bird`. They all share the Field `Species` (inherited from `Bird`) but only a `DomesticParrot` has a Field `Name`. Each of the 3 Classes has its own Constructor called `egg`.

```
:Class Bird
    :Field Public Species
    ∇ egg spec
      :Access Public Instance
      :Implements Constructor
      Species←spec
    ∇
    ...
:EndClass ⍝ Bird

:Class Parrot: Bird
    ∇ egg species
      :Access Public Instance
      :Implements Constructor :Base 'Parrot: ',species
    ∇
    ...
:EndClass ⍝ Parrot

:Class DomesticParrot: Parrot
    :Field Public Name
    ∇ egg(name species)
      :Access Public Instance
      :Implements Constructor :Base species
      ⎕DF Name←name
    ∇
    ...
:EndClass ⍝ DomesticParrot

      pol←⎕NEW DomesticParrot('Polly' 'Scarlet Macaw')
      pol.Name
Polly
      pol.Species
Parrot: Scarlet Macaw
```

Explanation

⎕NEW creates the new instance and runs the Constructor DomesticParrot.egg. The egg header splits the argument into two items name and species. As soon as the line:

```
:Implements Constructor :Base species
```

is encountered, ⎕NEW calls the Base Class constructor Parrot.egg, passing it the result of the expression to the right, which in this case is simply the value in species.

Parrot.egg starts to execute and as soon as the line:

```
:Implements Constructor :Base 'Parrot: ',species
```

is encountered, ⎕NEW calls *its* Base Class constructor Bird.egg, passing it the result of the expression to the right, which in this case is the character vector 'Parrot: ' catenated with the value in species.

Bird.egg assigns its argument to the Public Field Species.

At this point, the State Indicator would be:

```
      )SI
[#.[Instance of DomesticParrot]] #.Bird.egg[3]*
[constructor]
:base
[#.[Instance of DomesticParrot]] #.Parrot.egg[2]
[constructor]
:base
[#.[Instance of DomesticParrot]] #.DomesticParrot.egg[2]
[constructor]
```

Bird.egg then returns to Parrot.egg which returns to DomesticParrot.egg.

Finally, DomesticParrot.egg[3] is executed, which establishes Field Name and the Display Format (⎕DF) for the instance.

Destructors

A *Destructor* is a function that is called just before an Instance of a Class ceases to exist and is typically used to close files or release external resources associated with an Instance.

An Instance of a Class is destroyed when:

- The Instance is expunged using `⎕EX` or `)ERASE`.
- A function, in which the Instance is localised, exits.

But be aware that a destructor will also be called if:

- The Instance is re-assigned (see below)
- The result of `⎕NEW` is not assigned (the instance gets created then immediately destroyed).
- APL creates (and then destroys) a new Instance as a result of a reference to a member of an empty Instance. The destructor is called after APL has obtained the appropriate value from the instance and no longer needs it.
- The constructor function fails. Note that the Instance is actually created before the constructor is run (inside it), and if the constructor fails, the fledgling Instance is discarded. Note too that this means a destructor *may* need to deal with a partially constructed instance, so the code may need to check that resources were actually acquired, before releasing them.
- On the execution of `)CLEAR`, `)LOAD`, `⎕LOAD` or `⎕OFF`.

Note that an Instance of a Class only disappears when the *last reference* to it disappears. For example, the sequence:

```
      I1←⎕NEW MyClass
      I2←I1
      )ERASE I1
```

will not cause the Instance of `MyClass` to disappear because it is still referenced by `I2`.

A Destructor is identified by the statement `:Implements Destructor` which must appear immediately after the function header in the Class script.

```
:Class Parrot
    ...
    ∇ kill
      :Implements Destructor
      'This Parrot is dead'
    ∇
    ...
:EndClass ⍝ Parrot

      pol←⎕NEW Parrot 'Scarlet Macaw'
      )ERASE pol
This Parrot is dead
```

Note that reassignment to `pol` causes the Instance referenced by `pol` to be destroyed and the Destructor invoked:

```
      pol←⎕NEW Parrot 'Scarlet Macaw'
      pol←⎕NEW Parrot 'Scarlet Macaw'
This Parrot is dead
```

If a Class inherits from another Class, the Destructor in its Base Class is automatically called after the Destructor in the Class itself.

So, if we have a Class structure:

```
      DomesticParrot => Parrot => Bird
```

containing the following Destructors:

```
:Class DomesticParrot: Parrot
    ...
    ∇ kill
      :Implements Destructor
      'This ',(⍕⎕THIS),' is dead'
    ∇
     ...
:EndClass ⍝ DomesticParrot

:Class Parrot: Bird
    ...
    ∇ kill
      :Implements Destructor
      'This Parrot is dead'
    ∇
    ...
:EndClass ⍝ Parrot

:Class Bird
    ...
    ∇ kill
      :Implements Destructor
      'This Bird is dead'
    ∇
    ...
:EndClass ⍝ Bird
```

Destroying an Instance of `DomesticParrot` will run the Destructors in `DomesticParrot`, `Parrot` and `Bird` and in that order.

```
      pol←⎕NEW DomesticParrot
      )CLEAR
This Polly is dead
This Parrot is dead
This Bird is dead
clear ws
```

Class Members

A Class may contain *Methods*, *Fields* and *Properties* (commonly referred to together as *Members*) which are defined within the body of the Class script or are inherited from other Classes.

Methods are regular APL defined functions, but with some special characteristics that control how they are called and where they are executed. D-fns may not be used as Methods.

Fields are just like APL variables. To get the Field value, you reference its name; to set the Field value, you assign to its name, and the Field value is stored *in* the Field. However, Fields differ from variables in that they possess characteristics that control their accessibility.

Properties are similar to APL variables. To get the Property value, you reference its name; to set the Property value, you assign to its name. However, Property values are actually accessed via *PropertyGet* and *PropertySet* functions that may perform all sorts of operations. In particular, the value of a Property is not stored *in* the Property and may be entirely dynamic.

All three types of member may be declared as *Public* or *Private* and as *Instance* or *Shared*.

Public members are visible from outside the Class and Instances of the Class, whereas Private members are only accessible from within.

Instance Members are unique to every Instance of the Class, whereas Shared Members are common to all Instances and Shared Members may be referenced directly on the Class itself.

Fields

A Field behaves just like an APL variable.

To get the value of a Field, you reference its name; to set the value of a Field, you assign to its name. Conceptually, the Field value is stored *in* the Field. However, Fields differ from variables in that they possess characteristics that control their accessibility.

A Field may be declared anywhere in a Class script by a `:Field` statement. This specifies:

- the name of the Field
- whether the Field is Public or Private
- whether the Field is Instance or Shared
- whether or not the Field is ReadOnly
- optionally, an initial value for the Field.

Note that Triggers may be associated with Fields. See Trigger Fields for details.

Public Fields

A *Public* Field may be accessed from outside an Instance or a Class. Note that the default is *Private*.

Class `DomesticParrot` has a `Name` Field which is defined to be Public and Instance (by default).

```
:Class DomesticParrot: Parrot
    :Field Public Name

    ∇ egg nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇
    ...
:EndClass ⍝ DomesticParrot
```

The Name field is initialised by the Class constructor.

```
      pet←⎕NEW DomesticParrot'Polly'
      pet.Name
Polly
```

The Name field may also be modified directly:

```
      pet.Name←⌽pet.Name
      pet.Name
ylloP
```

Initialising Fields

A Field may be assigned an initial value. This can be specified by an arbitrary expression that is executed when the Class is fixed by the Editor or by `⎕FIX`.

```
:Class DomesticParrot: Parrot
    :Field Public Name←'Dicky'
        :Field Public Talks←1

    ∇ egg nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇
    ...
:EndClass ⍝ DomesticParrot
```

Field `Talks` will be initialised to `1` in every instance of the Class.

```
      pet←⎕NEW DomesticParrot 'Dicky'

      pet.Talks
1
      pet.Name
Dicky
```

Note that if a Field is ReadOnly, this is the only way that it may be assigned a value.

See also: Shared Fields

Private Fields

A Private Field may only be referenced by code running inside the Class or an Instance of the Class. Furthermore, Private Fields are not inherited.

The ComponentFile Class (see page 157) has a Private Instance Field named `tie` that is used to store the file tie number in each Instance of the Class.

```
:Class ComponentFile
    :Field Private Instance tie

    ∇ Open filename
      :Implements Constructor
      :Access Public Instance
      :Trap 0
          tie←filename ⎕FTIE 0
      :Else
          tie←filename ⎕FCREATE 0
      :EndTrap
      ⎕DF filename,'(Component File)'
    ∇
```

As the field is declared to be Private, it is not accessible from outside an Instance of the Class, but is only visible to code running inside.

```
      F1←⎕NEW ComponentFile 'test1'
      F1.tie
VALUE ERROR
      F1.tie
     ^
```

Shared Fields

If a Field is declared to be *Shared*, it has the same value for every Instance of the Class. Moreover, the Field may be accessed from the Class itself; an Instance is not required.

The following example establishes a Shared Field called `Months` that contains abbreviated month names which are appropriate for the user's current International settings. It also shows that an arbitrarily complex statement may be used to initialise a Field.

```
:Class Example
    :Using System.Globalization
    :Field Public Shared ReadOnly Months←12↑(⎕NEW
DateTimeFormatInfo).AbbreviatedMonthNames
:EndClass ⍝ Example
```

A Shared Field is not only accessible from an instance →

```
      EG←⎕NEW Example
      EG.Months
 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov...
```

→ but also, directly from the Class itself.

```
      Example.Months
Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov...
```

Notice that in this case it is necessary to insert a `:Using` statement (or the equivalent assignment to `⎕USING`) in order to specify the .Net search path for the DateTimeFormatInfo type. Without this, the Class would fail to fix.

You can see how the assignment works by executing the same statements in the Session:

```
      ⎕USING←'System.Globalization'
    12↑(⎕NEW DateTimeFormatInfo).AbbreviatedMonthNames
 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov...
```

Trigger Fields

A Fields may act as a Trigger so that a function may be invoked whenever the value of the Field is changed.

As an example, it is often useful for the Display Form of an Instance to reflect the value of a certain Field. Naturally, when the Field changes, it is desirable to change the Display Form. This can be achieved by making the Field a Trigger as illustrated by the following example.

Notice that the Trigger function is invoked both by assignments made within the Class (as in the assignment in `ctor`) and those made from outside the Instance.

```
:Class MyClass
    :Field Public Name
    :Field Public Country←'England'
    ∇ ctor nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇
    ∇ format
      :Implements Trigger Name,Country
      ⎕DF'My name is ',Name,' and I live in ',Country
    ∇
:EndClass ⍝ MyClass

      me←⎕NEW MyClass 'Pete'
      me
My name is Pete and I live in England

      me.Country←'Greece'
      me
My name is Pete and I live in Greece

      me.Name←'Kostas'
      me
My name is Kostas and I live in Greece
```

Methods

Methods are implemented as regular defined functions, but with some special attributes that control how they are called and where they are executed.

A Method is defined by a contiguous block of statements in a Class Script. A Method begins with a line that contains a `∇`, followed by a valid APL defined function header. The method definition is terminated by a closing `∇`.

The behaviour of a Method is defined by an `:Access` control statement.

Public or Private

Methods may be defined to be Private (the default) or Public.

A Private method may only be invoked by another function that is running inside the Class namespace or inside an Instance namespace. The name of a Private method is not visible from outside the Class or an Instance of the Class.

A Public method may be called from outside the Class or an Instance of the Class.

Instance or Shared

Methods may be defined to be Instance (the default) or Shared.

An Instance method runs in the Instance namespace and may only be called via the instance itself. An Instance method has direct access to Fields and Properties, both Private and Public, in the Instance in which it runs.

A Shared method runs in the Class namespace and may be called via an Instance or via the Class. However, a Shared method that is called via an Instance does not have direct access to the Fields and Properties of that Instance.

Shared methods are typically used to manipulate Shared Properties and Fields or to provide general services for all Instances that are not Instance specific.

Overridable Methods

Instance Methods may be declared with `:Access Overridable`.

A Method declared as being Overridable is replaced in situ (i.e.within its own Class) by a Method of the same name that is defined in a higher Class which itself is declared with the Override keyword. See *Superseding Base Class Methods*.

Shared Methods

A Shared method runs in the Class namespace and may be called via an Instance or via the Class. However, a Shared method that is called via an Instance does not have direct access to the Fields and Properties of that Instance.

Class `Parrot` has a `Speak` method that does not require any information about the current Instance, so may be declared as Shared.

```
:Class Parrot:Bird

    ∇ R←Speak times
      :Access Public Shared
      R←⍕timesρ⊂'Squark!'
    ∇

:EndClass ⍝ Parrot

      wild←⎕NEW Parrot
      wild.Speak 2
 Squark!  Squark!
```

Note that `Parrot.Speak` may be executed directly from the Class and does not in fact require an Instance.

```
      Parrot.Speak 3
 Squark!  Squark!  Squark!
```

Instance Methods

An Instance method runs in the Instance namespace and may only be called via the instance itself. An Instance method has direct access to Fields and Properties, both Private and Public, in the Instance in which it runs.

Class `DomesticParrot` has a `Speak` method defined to be Public and Instance. Where `Speak` refers to `Name`, it obtains the value of `Name` in the current Instance.

Note too that `DomesticParrot.Speak` supersedes the inherited `Parrot.Speak`.

```
:Class DomesticParrot: Parrot
    :Field Public Name

    ∇ egg nm
      :Access Public
      :Implements Constructor
      Name←nm
    ∇

    ∇ R←Speak times
      :Access Public Instance
      R←⊂Name,', ',Name
      R←↑R,timesρ⊂' Who's a pretty boy,then!'
    ∇

:EndClass ⍝ DomesticParrot

      pet←⎕NEW DomesticParrot'Polly'
      pet.Speak  3
Polly, Polly
 Who's a pretty boy,then!
 Who's a pretty boy,then!
 Who's a pretty boy,then!

      bil←⎕NEW  DomesticParrot'Billy'
      bil.Speak  1
Billy, Billy
 Who's a pretty boy,then!
```

Superseding Base Class Methods

Normally, a Method defined in a higher Class supersedes the Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*. This behaviour can be altered using the Overridable and Override key words in the `:Access` statement but only applies to Instance Methods.

If a Public Instance method in a Class is marked as *Overridable*, this allows a Class which derives from the Class with the Overridable method to supersede the Base Class method **in the Base Class**, by providing a method which is marked *Override*. The typical use of this is to replace code in the Base Class which handles an event, with a method provided by the derived Class.

For example, the base class might have a method which is called if any error occurs in the base class:

```
     ∇ ErrorHandler
[1]    :Access Public Overridable
[2]    ⎕←↑⎕DM
     ∇
```

In your derived class, you might supersede this by a more sophisticated error handler, which logs the error to a file:

```
     ∇ ErrorHandler;TN
[1]    :Access Public Override
[2]    ⎕←↑⎕DM
[3]    TN←'ErrorLog'⎕FSTIE 0
[4]    ⎕DM ⎕FAPPEND TN
[5]    ⎕FUNTIE TN
     ∇
```

If the derived class had a function which was not marked Override, then function in the derived class which called `ErrorHandler` would call the function as defined in the derived class, but if a function in the base class called `ErrorHandler`, it would still see the base class version of this function. With Override specified, the new function supersedes the function as seen by code in the base class. Note that different derived classes can specify different Overrides.

In C#, Java and some other compiled languages, the term *Virtual* is used in place of *Overridable*, which is the term used by Visual Basic and Dyalog APL.

Properties

A Property behaves in a very similar way to an ordinary APL variable. To obtain the value of a Property, you simply reference its name. To change the value of a Property, you assign a new value to the name.

However, *under the covers*, a Property is accessed via a *PropertyGet* function and its value is changed via a *PropertySet* function. Furthermore, Properties may be defined to allow partial (indexed) retrieval and assignment to occur.

There are three types of Property, namely *Simple*, *Numbered* and *Keyed*.

A *Simple Property* is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety.

A *Numbered Property* behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices. The Numbered Property is designed to allow APL to perform selections and structural operations on the Property.

A *Keyed Property* is similar to a Numbered Property except that its elements are accessed via arbitrary keys instead of indices.

The following cases illustrate the difference between Simple and Numbered Properties.

If Instance `MyInst` has a Simple Property `Sprop` and a Numbered Property `Nprop`, the expressions

```
      X←MyInst.SProp
      X←MyInst.SProp[2]
```

both cause APL to call the PropertyGet function to retrieve the entire value of `Sprop`. The second statement subsequently uses indexing to extract just the second element of the value.

Whereas, the expression:

```
      X←MyInst.NProp[2]
```

causes APL to call the PropertyGet function with an additional argument which specifies that only the second element of the Property is required. Moreover, the expression:

```
      X←MyInst.NProp
```

causes APL to call the PropertyGet function successively, for every element of the Property.

A Property is defined by a `:Property ... :EndProperty` section in a Class Script.

Within the body of a Property Section there may be:

- one or more `:Access` statements
- a single PropertyGet function.
- a single PropertySet function
- a single PropertyShape function

Simple Instance Properties

A Simple Instance Property is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety. The following examples are taken from the ComponentFile Class Class (see page 157).

The Simple Property `Count` returns the number of components on a file.

```
     :Property Count
     :Access Public Instance
         ∇ r←get
           r←¯1+2⊃⎕FSIZE tie
         ∇
     :EndProperty ⍝ Count

       F1←⎕NEW ComponentFile 'test1'
       F1.Append'Hello World'
1
       F1.Count
1
       F1.Append 42
2
       F1.Count
2
```

Because there is no `set` function defined, the Property is read-only and attempting to change it causes `SYNTAX ERROR`.

```
       F1.Count←99
SYNTAX ERROR
       F1.Count←99
      ^
```

The `Access` Property has both `get` and `set` functions which are used, in this simple example, to get and set the component file access matrix.

```
    :Property Access
    :Access Public Instance
        ∇ r←get
          r←⎕FRDAC tie
        ∇
        ∇ set am;mat;OK
          mat←am.NewValue
          :Trap 0
              OK←(2=⍴⍴mat)∧(3=2⊃⍴mat)∧∧/,mat=⌊mat
          :Else
              OK←0
          :EndTrap
          'bad arg'⎕SIGNAL(~OK)/11
          mat ⎕FSTAC tie
        ∇
    :EndProperty ⍝ Access
```

Note that the `set` function **must** be monadic. Its argument, supplied by APL, will be an Instance of `PropertyArguments`. This is an internal Class whose `NewValue` field contains the value that was assigned to the Property.

Note that the set function does not have to accept the new value that has been assigned. The function may validate the value reject or accept it (as in this example), or perform whatever processing is appropriate.

```
      F1←⎕NEW ComponentFile 'test1'
      ⍴F1.Access
0 3
        F1.Access←3 3⍴28 2105 16385 0 2073 16385 31 ¯1 0
      F1.Access
28 2105 16385
 0 2073 16385
31   ¯1     0

      F1.Access←'junk'
bad arg
      F1.Access←'junk'
     ^

      F1.Access←1 2⍴10
bad arg
      F1.Access←1 2⍴10
     ^
```

Simple Shared Properties

The ComponentFile Class (see page 157) specifies a Simple Shared Property named `Files` which returns the names of all the Component Files in the current directory.

The previous examples have illustrated the use of Instance Properties. It is also possible to define *Shared* properties.

A Shared property may be used to handle information that is relevant to the Class as a whole, and which is not specific to any a particular Instance.

```
:Property Files
:Access Public Shared
    ∇ r←get
      r←⎕FLIB''
    ∇
:EndProperty
```

Note that `⎕FLIB` (invoked by the `Files get` function) does not report the names of *tied* files.

```
      F1←⎕NEW ComponentFile 'test1'
      ⎕EX'F1'
      F2←⎕NEW ComponentFile 'test2'
      F2.Files ⍝ NB ⎕FLIB does not report tied files
test1
      ⎕EX'F2'
```

Note that a Shared Property may be accessed from the Class itself. It is not necessary to create an Instance first.

```
      ComponentFile.Files
test1
test2
```

Numbered Properties

A Numbered Property behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices.

To implement a Numbered Property, you **must** specify a PropertyShape function and either or both a PropertyGet and PropertySet function.

When an expression references or makes an assignment to a Numbered Property, APL first calls its PropertyShape function which returns the dimensions of the Property. Note that the shape of the result of this function determines the *rank* of the Property.

If the expression uses indexing, APL checks that the index or indices are within the bounds of these dimensions, and then calls the PropertyGet or PropertySet function. If the expression specifies a single index, APL calls the PropertyGet or PropertySet function once. If the expression specifies multiple indices, APL calls the function successively.

If the expression references or assigns the entire Property (without indexing) APL generates a set of indices for every element of the Property and calls the PropertyGet or PropertySet function successively for every element in the Property.

Note that APL generates a `RANK ERROR` if an index contains the wrong number of elements or an `INDEX ERROR` if an index is out of bounds.

When APL calls a monadic PropertyGet or PropertySet function, it supplies an argument of type PropertyArguments.

Example

The ComponentFile Class Class (see page 157) specifies a Numbered Property named `Component` which represents the contents of a specified component on the file.

```
:Property Numbered Component
:Access Public Instance
    ∇ r←shape
      r←¯1+2⊃⎕FSIZE tie
    ∇
    ∇ r←get arg
      r←⎕FREAD tie arg.Indexers
    ∇
    ∇ set arg
      arg.NewValue ⎕FREPLACE tie,arg.Indexers
    ∇
:EndProperty
```

```
      F1←⎕NEW ComponentFile 'test1'

      F1.Append¨(⍳5)×⊂⍳4
1 2 3 4 5

      F1.Count
5

      F1.Component[4]
 4 8 12 16

      4⊃F1.Component
4 8 12 16

      (⊂4 3)⌷F1.Component
 4 8 12 16  3 6 9 12
```

Referencing a Numbered Property in its entirety causes APL to call the `get` function successively for every element.

```
      F1.Component
 1 2 3 4  2 4 6 8  3 6 9 12  4 8 12 16  5 10 15 20

      ((⊂4 3)⌷F1.Component)←'Hello' 'World'

      F1.Component[3]
 World
```

Attempting to access a Numbered Property with inappropriate indices generates an error:

```
      F1.Component[6]
INDEX ERROR
      F1.Component[6]
     ^
      F1.Component[1;2]
RANK ERROR
      F1.Component[1;2]
     ^
```

The Default Property

A single Numbered Property may be identified as the *Default* Property for the Class. If a Class has a Default Property, indexing with the ⌷ primitive function and `[...]` indexing may be applied to the Property directly via a reference to the Class or Instance.

The Numbered Property example of the ComponentFile Class Class (see page 157) can be extended by adding the control word `Default` to the `:Property` statement for the `Component` Property.

Indexing may now be applied directly to the Instance `F1`. In essence, `F1[n]` is simply shorthand for `F1.Component[n]` and `n⌷F1` is shorthand for `n⌷F1.Component`

```
    :Property Numbered Default Component
    :Access Public Instance
        ∇ r←shape
          r←¯1+2⊃⎕FSIZE tie
        ∇
        ∇ r←get arg
          r←⎕FREAD tie arg.Indexers
        ∇
        ∇ set arg
          arg.NewValue ⎕FREPLACE tie,arg.Indexers
        ∇
    :EndProperty

      F1←⎕NEW ComponentFile 'test1'
      F1.Append¨(⍳5)×⊂⍳4
1 2 3 4 5
      F1.Count
5

      F1[4]
 4 8 12 16
      (⊂4 3)⌷F1
 4 8 12 16  3 6 9 12
      ((⊂4 3)⌷F1)←'Hello' 'World'
      F1[3]
 World
```

Note however that this feature applies only to indexing.

```
      4⊃F1
DOMAIN ERROR
      4⊃F1
     ^
```

ComponentFile Class Example

```
:Class ComponentFile
    :Field Private Instance tie

    ∇ Open filename
      :Implements Constructor
      :Access Public Instance
      :Trap 0
          tie←filename ⎕FTIE 0
      :Else
          tie←filename ⎕FCREATE 0
      :EndTrap
      ⎕DF filename,'(Component File)'
    ∇

    ∇ Close
      :Access Public Instance
      ⎕FUNTIE tie
    ∇

    ∇ r←Append data
      :Access Public Instance
      r←data ⎕FAPPEND tie
    ∇

    ∇ Replace(comp data)
      :Access Public Instance
      data ⎕FREPLACE tie,comp
    ∇

:Property Count
    :Access Public Instance
        ∇ r←get
          r←¯1+2⊃⎕FSIZE tie
        ∇
    :EndProperty ⍝ Count
```

Component File Class Example (continued)

```
:Property Access
    :Access Public Instance
        ∇ r←get arg
          r←⎕FRDAC tie
        ∇
        ∇ set am;mat;OK
          mat←am.NewValue
          :Trap 0
              OK←(2=⍴⍴mat)^(3=2⊃⍴mat)^^/,mat=⌊mat
          :Else
              OK←0
          :EndTrap
          'bad arg'⎕SIGNAL(~OK)/11
          mat ⎕FSTAC tie
        ∇
    :EndProperty ⍝ Access

    :Property Files
    :Access Public Shared
        ∇ r←get
          r←⎕FLIB''
        ∇
    :EndProperty

    :Property Numbered Default Component
    :Access Public Instance
        ∇ r←shape args
          r←¯1+2⊃⎕FSIZE tie
        ∇
        ∇ r←get arg
          r←⊂⎕FREAD tie,arg.Indexers
        ∇
        ∇ set arg
          (⊃arg.NewValue)⎕FREPLACE tie,arg.Indexers
        ∇
    :EndProperty

    ∇ Delete file;tie
      :Access Public Shared
      tie←file ⎕FTIE 0
      file ⎕FERASE tie
    ∇
:EndClass ⍝ Class ComponentFile
```

Keyed Properties

A Keyed Property is similar to a Numbered Property except that it may **only** be accessed by indexing (so-called square-bracket indexing) and indices are not restricted to integers but may be arbitrary arrays.

To implement a Keyed Property, only a `get` and/or a `set` function are required. APL does not attempt to validate or resolve the specified indices in any way, so does not require the presence of a `shape` function for the Property.

However, APL **does** check that the rank and lengths of the indices correspond to the rank and lengths of the array to the right of the assignment (for an indexed assignment) and the array returned by the get function (for an indexed reference). If the rank or shape of these arrays fails to conform to the rank or shape of the indices, APL will issue a `RANK ERROR` or `LENGTH ERROR`.

Note too that indices **may not be elided**. If `KProp` is a Keyed Property of Instance `I1`, the following expressions would all generate `NONCE ERROR`.

```
      I1.KProp
      I1.KProp[]←10
      I1.KProp[;]←10
      I1.KProp['One' 'Two';]←10
      I1.KProp[;'One' 'Two']←10
```

When APL calls a monadic `get` or a `set` function, it supplies an argument of type PropertyArguments.

The Sparse2 Class illustrates the implementation and use of a Keyed Property.

`Sparse2` represents a 2-dimensional sparse array each of whose dimensions are indexed by arbitrary character keys. The sparse array is implemented as a Keyed Property named `Values`. The following expressions show how it might be used.

```
      SA1←⎕NEW Sparse2
      SA1.Values[⊂'Widgets';⊂'Jan']←100
      SA1.Values[⊂'Widgets';⊂'Jan']
100
      SA1.Values['Widgets' 'Grommets';'Jan' 'Mar'
'Oct']←10×2 3⍴⍳6
      SA1.Values['Widgets' 'Grommets';'Jan' 'Mar' 'Oct']
10 20 30
40 50 60
      SA1.Values[⊂'Widgets';'Jan' 'Oct']
10 30
      SA1.Values['Grommets' 'Widgets';⊂'Oct']
60
30
```

Sparse2 Class Example

```
:Class Sparse2  ⍝ 2D Sparse Array
    :Field Private keys
    :Field Private values
    ∇ make
      :Access Public
      :Implements Constructor
      keys←0⍴⊂'' ''
      values←⍬
    ∇
    :Property Keyed Values
    :Access Public Instance
        ∇ v←get arg;k
          k←arg.Indexers
          ⎕SIGNAL(2≠⍴k)/4
          k←fixkeys k
          v←(values,0)[keys⍳k]
        ∇
        ∇ set arg;new;k;v;n
          v←arg.NewValue
          k←arg.Indexers
          ⎕SIGNAL(2≠⍴k)/4
          k←fixkeys k
          v←(⍴k)(⍴⍣(⊃1=⍴,v))v
          ⎕SIGNAL((⍴k)≠⍴v)/5
          k v←,¨k v
          :If ∨/new←~k∊keys
              values,←new/v
              keys,←new/k
              k v/⍨←⊂~new
          :EndIf
          :If 0<⍴k
              values[keys⍳k]←v
          :EndIf
        ∇
    :EndProperty

    ∇ k←fixkeys k
      k←(2≠≡¨k){,(⊂⍣⍺)⍵}¨k
      k←⊃(∘.{⊃,/⊂¨⍺ ⍵})/k
    ∇
:EndClass ⍝ 2D Sparse Array
```

Internally, Sparse2 maintains a list of keys and a list of values which are initialised to empty arrays by its constructor.

When an indexed assignment is made, the set function receives a list of keys (indices) in arg.Indexer and values in arg.NewValue. The function updates the values of existing keys, and adds new keys and their values to the internal lists.

When an indexed reference is made, the get function receives a list of keys (indices) in arg.Indexer. The function uses these keys to retrieve the corresponding values, inserting 0s for non-existent keys.

Note that in the expression:

```
 SA1.Values['Widgets' 'Grommets';'Jan' 'Mar' 'Oct']
```

the structure of arg.Indexer is:

```
.→----------------------------------------------.
| .→--------------------. .→-----------------. |
| | .→------. .→-------. | | .→--. .→--. .→--. | |
| | |Widgets| |Grommets| | | |Jan| |Mar| |Oct| | |
| | '-------' '--------' | | '---' '---' '---' | |
| '∊--------------------' '∊-----------------' |
'∊----------------------------------------------'
```

Example

A second example of a Keyed Property is provided by the `KeyedFile` Class which is based upon the ComponentFile Class Class (see page 168) used previously.

```
:Class KeyedFile: ComponentFile
    :Field Public Keys
    ⎕ML←0

    ∇ Open filename
      :Implements Constructor :Base filename
      :Access Public Instance
      :If Count>0
          Keys←{⊃⍵⊃⎕BASE.Component}¨⍳Count
      :Else
          Keys←0⍴⊂''
      :EndIf
    ∇

    :Property Keyed Component
    :Access Public Instance
        ∇ r←get arg;keys;sink
          keys←⊃arg.Indexers
          ⎕SIGNAL(~∧/keys∊Keys)/3
          r←{2⊃⍵⊃⎕BASE.Component}¨Keys⍳keys
        ∇
        ∇ set arg;new;keys;vals
          vals←arg.NewValue
          keys←⊃arg.Indexers
          ⎕SIGNAL((⍴,keys)≠⍴,vals)/5
          :If ∨/new←~keys∊Keys
              sink←Append¨↓⍉↑(⊂new)/¨keys vals
              Keys,←new/keys
              keys vals/⍨←⊂~new
          :EndIf
          :If 0<⍴,keys
              Replace¨↓⍉↑(Keys⍳keys)(↓⍉↑keys vals)
          :EndIf
        ∇
    :EndProperty

:EndClass ⍝ Class KeyedFile

      K1←⎕NEW KeyedFile 'ktest'
      K1.Count
0
      K1.Component[⊂'Pete']←42
      K1.Count
1
      K1.Component['John' 'Geoff']←(⍳10)(3 4⍴⍳12)
      K1.Count
3
```

```
    K1.Component['Geoff' 'Pete']
 1  2  3  4  42
 5  6  7  8
 9 10 11 12
      K1.Component['Pete' 'Morten']←(3 4⍴'∘')(⍳⍳3)
      K1.Count
4
      K1.Component['Morten' 'Pete' 'John']
  1 1 1  1 1 2  1 1 3   ∘∘∘∘  1 2 3 4 5 6 7 8 9 10
  1 2 1  1 2 2  1 2 3   ∘∘∘∘
                        ∘∘∘∘
```

Interfaces

An Interface is defined by a Script that contains skeleton declarations of Properties and/or Methods. These members are only *place-holders*; they have no specific implementation; this is provided by each of the the Classes that support the Interface.

An Interface contains a collection of methods and properties that together represents a *protocol* that an application must follow in order to manipulate a Class in a particular way.

An example might be an Interface called Icompare that provides a single method (Compare) which compares two Instances of a Class, returning a value to indicate which of the two is greater than the other. A Class that implements Icompare must provide an appropriate Compare method, but every Class will have its own individual version of Compare. An application can then be written that sorts Instances of any Class that supports the ICompare Interface.

An Interface is implemented by a Class if it includes the name of the Interface in its :Class statement, and defines a corresponding set of the Methods and Properties that are declared in the Interface.

To implement a Method, a function defined in the Class must include a `:Implements Method` statement that maps it to the corresponding Method defined in the Interface:

```
      :Implements Method <InterfaceName.MethodName>
```

Furthermore, the syntax of the function (whether it be result returning, monadic or niladic) must exactly match that of the method described in the Interface. The function name, however, need not be the same as that described in the Interface.

Similarly, to implement a Property the type (Simple, Numbered or Keyed) and syntax (defined by the presence or absence of a PropertyGet and PropertySet functions) must exactly match that of the property described in the Interface. The Property name, however, need not be the same as that described in the Interface.

Example

The Penguin Class Class example illustrates the use of Interfaces to implement *multiple inheritance*.

```
:Interface FishBehaviour
∇ R←Swim ⍝ Returns description of swimming capability
∇
:EndInterface ⍝ FishBehaviour

:Interface BirdBehaviour
∇ R←Fly ⍝ Returns description of flying capability
∇
∇ R←Lay ⍝ Returns description of egg-laying behaviour
∇
∇ R←Sing ⍝ Returns description of bird-song
∇
:EndInterface ⍝ BirdBehaviour

:Class Penguin: Animal,BirdBehaviour,FishBehaviour
    ∇ R←NoCanFly
      :Implements Method BirdBehaviour.Fly
      R←'Although I am a bird, I cannot fly'
    ∇
    ∇ R←LayOneEgg
      :Implements Method BirdBehaviour.Lay
      R←'I lay one egg every year'
    ∇
    ∇ R←Croak
      :Implements Method BirdBehaviour.Sing
      R←'Croak, Croak!'
    ∇
    ∇ R←Dive
      :Implements Method FishBehaviour.Swim
      R←'I can dive and swim like a fish'
    ∇
:EndClass ⍝ Penguin
```

In this case, the `Penguin` Class derives from `Animal` but additionally supports the `BirdBehaviour` and `FishBehaviour` Interfaces, thereby inheriting members from both.

```
      Pingo←⎕NEW Penguin
      ⎕CLASS Pingo
  #.Penguin  #.FishBehaviour  #.BirdBehaviour    #.Animal

      (FishBehaviour ⎕CLASS Pingo).Swim
I can dive and swim like a fish
      (BirdBehaviour ⎕CLASS Pingo).Fly
Although I am a bird, I cannot fly
      (BirdBehaviour ⎕CLASS Pingo).Lay
I lay one egg every year
      (BirdBehaviour ⎕CLASS Pingo).Sing
Croak, Croak!
```

Including Namespaces

A Class may import methods from one or more plain Namespaces. This allows several Classes to share a common set of methods, and provides a degree of multiple inheritance.

To import methods from a Namespace `NS`, the Class Script must include a statement:

```
:Include NS
```

When the Class is fixed by the editor or by `⎕FIX,` all the defined functions and operators in Namespace `NS` are included as methods in the Class. The functions and operators which are brought in as methods from the namespace `NS` are treated exactly as if the source of each function/operator had been included in the class script at the point of the `:Include` statement. For example, if a function contains `:Signature` or `:Access` statements, these will be taken into account. Note that such declarations have no effect on a function/operator which is in an ordinary namespace.

D-fns and D-ops in `NS` are also included in the Class but as *Private members*, because D-fns and D-ops may not contain `:Signature` or `:Access` statements. Variables and Sub-namespaces in `NS` are **not** included.

Note that objects imported in this way are not actually *copied*, so there is no penalty incurred in using this feature. Additions, deletions and changes to the functions in `NS` are immediately reflected in the Class.

If there is a member in the Class with the same name as a function in `NS`, the Class member takes precedence and supersedes the function in `NS`.

Conversely, functions in `NS` will supersede members of the same name that are inherited from the Base Class, so the precedence is:

Class supersedes
    Included Namespace, supersedes
        Base Class

Any number of Namespaces may be included in a Class and the `:Include` statements may occur anywhere in the Class script. However, for the sake of readability, it is recommended that you have `:Include` statements at the top, given that any definitions in the script will supersede included functions and operators.

Example

In this example, Class `Penguin` inherits from `Animal` and includes functions from the plain Namespaces `BirdStuff` and `FishStuff`.

```
:Class Penguin: Animal
    :Include BirdStuff
    :Include FishStuff
:EndClass ⍝ Penguin
```

Namespace `BirdStuff` contains 2 functions, both declared as Public methods.

```
:Namespace BirdStuff
    ∇ R←Fly
      :Access Public Instance
      R←'Fly, Fly ...'
    ∇
    ∇ R←Lay
      :Access Public Instance
      R←'Lay, Lay ...'
    ∇
:EndNamespace ⍝ BirdStuff
```

Namespace `FishStuff` contain a single function, also declared as a Public method.

```
:Namespace FishStuff
    ∇ R←Swim
      :Access Public Instance
      R←'Swim, Swim ...'
    ∇
:EndNamespace ⍝ FishStuff

      Pingo←⎕NEW Penguin
      Pingo.Swim
Swim, Swim ...
      Pingo.Lay
Lay, Lay ...
      Pingo.Fly
Fly, Fly ...
```

This is getting silly - we all know that Penguin's can't fly. This problem is simply resolved by overriding the `BirdStuff.Fly` method with `Penguin.Fly`. We can hide `BirdStuff.Fly` with a Private method in `Penguin` that does nothing. For example:

```
:Class Penguin: Animal
    :Include BirdStuff
    :Include FishStuff
    ∇ Fly ⍝ Override BirdStuff.Fly
    ∇
:EndClass ⍝ Penguin

      Pingo←⎕NEW Penguin
      Pingo.Fly
VALUE ERROR
      Pingo.Fly
     ^
```

or we can supersede it with a different Public method, as follows:

```
:Class Penguin: Animal
    :Include BirdStuff
    :Include FishStuff
    ∇ R←Fly ⍝ Override BirdStuff.Fly
      :Access Public Instance
      R←'Sadly, I cannot fly'
    ∇
:EndClass ⍝ Penguin

      Pingo←⎕NEW Penguin
      Pingo.Fly
Sadly, I cannot fly
```

Nested Classes

It is possible to define *Classes within Classes* (Nested Classes).

A Nested Class may be either `Private` or `Public`. This is specified by a :Access Statement, which must precede the definition of any Class contents. The default is `Private`.

A `Public` Nested Class is visible from outside its containing Class and may be used directly in its own right, whereas a `Private` Nested Class is not and may only be used by code inside the containing Class.

However, methods in the containing Class may return instances of Private Nested Classes and in that way expose them to the calling environment.

The GolfService Example Class illustrates the use of nested classes. GolfService was originally developed as a Web Service for Dyalog.Net and is one of the samples distributed in samples\asp.net\webservices. This version has been reconstructed as a stand-alone APL Class.

GolfService contains the following nested classes, all of which are `Private`.

GolfCourse	A Class that represents a Golf Course, having Fields `Code` and `Name`.
Slot	A Class that represents a tee-time or match, having Fields `Time` and `Players`. Up to 4 players may play together in a match.
Booking	A Class that represents a reservation for a particular tee-time at a particular golf course. This has Fields `OK`, `Course`, `TeeTime` and `Message`. The value of `TeeTime` is an Instance of a Slot Class.
StartingSheet	A Class that represents a day's starting-sheet at a particular golf course. It has Fields `OK`, `Course`, `Date`, `Slots`, `Message`. `Slots` is an array of Instances of Slot Class.

GolfService Example Class

```
:Class GolfService
:Using System

    :Field Private GOLFILE←'' ⍝ Name of Golf data file
    :Field Private GOLFID←0 ⍝ Tie number Golf data file

    :Class GolfCourse
        :Field Public Code←¯1
        :Field Public Name←''

        ∇ ctor args
          :Implements Constructor
          :Access Public Instance
          Code Name←args
          ⎕DF Name,'(',(⍕Code),')'
        ∇

    :EndClass

    :Class Slot
        :Field Public Time
        :Field Public Players

        ∇ ctor1 t
          :Implements Constructor
          :Access Public Instance
          Time←t
          Players←0⍴⊂''
        ∇
        ∇ ctor2 (t pl)
          :Implements Constructor
          :Access Public Instance
          Time Players←t pl
        ∇
        ∇ format
          :Implements Trigger Players
          ⎕DF⍕Time Players
        ∇
    :EndClass
```

```
:Class Booking
    :Field Public OK
    :Field Public Course
    :Field Public TeeTime
    :Field Public Message

    ∇ ctor args
      :Implements Constructor
      :Access Public Instance
      OK Course TeeTime Message←args
    ∇
    ∇ format
      :Implements Trigger OK,Message
      ⎕DF⍕Course TeeTime(⊃OK⌽Message'OK')
    ∇
:EndClass

:Class StartingSheet
    :Field Public OK
    :Field Public Course
    :Field Public Date
    :Field Public Slots←⎕NULL
    :Field Public Message

    ∇ ctor args
      :Implements Constructor
      :Access Public Instance
      OK Course Date←args
    ∇
    ∇ format
      :Implements Trigger OK,Message
      ⎕DF⍕2 1⍴(⍕Course Date)(↑⍕¨Slots)
    ∇
:EndClass

∇ ctor file
  :Implements Constructor
  :Access Public Instance
  GOLFILE←file
  ⎕FUNTIE(((↓⎕FNAMES)~' ')⍳⊂GOLFILE)⊃⎕FNUMS,0
  :Trap 22
      GOLFID←GOLFILE ⎕FTIE 0
  :Else
      InitFile
  :EndTrap
∇
```

```
∇ dtor
  :Implements Destructor
  ⎕FUNTIE GOLFID
∇

∇ InitFile;COURSECODES;COURSES;INDEX;I
  :Access Public
  :If GOLFID≠0
      GOLFILE ⎕FERASE GOLFID
  :EndIf
  GOLFID←GOLFILE ⎕FCREATE 0
  COURSECODES←1 2 3
  COURSES←'St Andrews' 'Hindhead' 'Basingstoke'
  INDEX←(⍴COURSES)⍴0
  COURSECODES COURSES INDEX ⎕FAPPEND GOLFID
  :For I :In ⍳⍴COURSES
      INDEX[I]←⍬ ⍬ ⎕FAPPEND 1
  :EndFor
  COURSECODES COURSES INDEX ⎕FREPLACE GOLFID 1
∇

∇ R←GetCourses;COURSECODES;COURSES;INDEX
  :Access Public
  COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
  R←{⎕NEW GolfCourse ⍵}¨↓⍉↑COURSECODES COURSES
∇
```

```
∇ R←GetStartingSheet ARGS;CODE;COURSE;DATE;COURSECODES
                         ;COURSES;INDEX;COURSEI;IDN
                         ;DATES;COMPS;IDATE;TEETIMES
                         ;GOLFERS;I;T
  :Access Public
  CODE DATE←ARGS
  COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
  COURSEI←COURSECODESιCODE
  COURSE←⎕NEW GolfCourse(CODE(COURSEI⊃COURSES,⊂''))
  R←⎕NEW StartingSheet(0 COURSE DATE)
  :If COURSEI>ρCOURSECODES
      R.Message←'Invalid course code'
      :Return
  :EndIf
  IDN←2 ⎕NQ'.' 'DateToIDN',DATE.(Year Month Day)
  DATES COMPS←⎕FREAD GOLFID,COURSEI⊃INDEX
  IDATE←DATESιIDN
  :If IDATE>ρDATES
      R.Message←'No Starting Sheet available'
      :Return
  :EndIf
  TEETIMES GOLFERS←⎕FREAD GOLFID,IDATE⊃COMPS
  T←DateTime.New¨(⊂DATE.(Year Month Day)),¨↓[1]
                             24 60 1⊤TEETIMES
  R.Slots←{⎕NEW Slot ω}¨T,∘⊂¨↓GOLFERS
  R.OK←1
∇
```

```
    ∇ R←MakeBooking ARGS;CODE;COURSE;SLOT;TEETIME
                         ;COURSECODES;COURSES;INDEX
                         ;COURSEI;IDN;DATES;COMPS;IDATE
                         ;TEETIMES;GOLFERS;OLD;COMP;HOURS
                         ;MINUTES;NEAREST;TIME;NAMES;FREE
                         ;FREETIMES;I;J;DIFF
      :Access Public
      ⍝ If GimmeNearest is 0, tries for specified time
⍝ If GimmeNearest is 1, gets nearest time
      CODE TEETIME NEAREST←3↑ARGS
      COURSECODES COURSES INDEX←⎕FREAD GOLFID 1
      COURSEI←COURSECODES⍳CODE
      COURSE←⎕NEW GolfCourse(CODE(COURSEI⊃COURSES,⊂''))
      SLOT←⎕NEW Slot TEETIME
      R←⎕NEW Booking(0 COURSE SLOT'')
      :If COURSEI>⍴COURSECODES
          R.Message←'Invalid course code'
          :Return
      :EndIf
      :If TEETIME.Now>TEETIME
          R.Message←'Requested tee-time is in the past'
          :Return
      :EndIf
      :If TEETIME>TEETIME.Now.AddDays 30
          R.Message←'Requested tee-time is more than 30
                                      days from now'
          :Return
      :EndIf
      IDN←2 ⎕NQ'.' 'DateToIDN',TEETIME.(Year Month Day)
      DATES COMPS←⎕FREAD GOLFID,COURSEI⊃INDEX
      IDATE←DATES⍳IDN
      :If IDATE>⍴DATES
          TEETIMES←(24 60⊥7 0)+10×¯1+⍳1+8×6
          GOLFERS←((⍴TEETIMES),4)⍴⊂''llowed per tee time
          :If 0=OLD←⊃(DATES<2 ⎕NQ'.' 'DateToIDN',3↑⎕TS)/
                                                ⍳⍴DATES
              COMP←(TEETIMES GOLFERS)⎕FAPPEND GOLFID
              DATES,←IDN
              COMPS,←COMP
              (DATES COMPS)⎕FREPLACE GOLFID,COURSEI⊃INDEX
          :Else
              DATES[OLD]←IDN
              (TEETIMES GOLFERS)⎕FREPLACE GOLFID,
                                      COMP←OLD⊃COMPS
              DATES COMPS ⎕FREPLACE GOLFID,COURSEI⊃INDEX
          :EndIf
```

```
             :Else
             COMP←IDATE⊃COMPS
             TEETIMES GOLFERS←⎕FREAD GOLFID COMP
         :EndIf
         HOURS MINUTES←TEETIME.(Hour Minute)
         NAMES←(3↓ARGS)~⊖''
         TIME←24 60⊥HOURS MINUTES
         TIME←10×⌊0.5+TIME÷10
         :If ~NEAREST
             I←TEETIMES⍳TIME
             :If I>⍴TEETIMES
             :OrIf (⍴NAMES)>⊃,/+/0=⍴¨GOLFERS[I;]
                 R.Message←'Not available'
                 :Return
             :EndIf
         :Else
             :If ~∨/FREE←(⍴NAMES)≤⊃,/+/0=⍴¨GOLFERS
                 R.Message←'Not available'
                 :Return
             :EndIf
             FREETIMES←(FREE×TEETIMES)+32767×~FREE
             DIFF←|FREETIMES-TIME
             I←DIFF⍳⌊/DIFF
         :EndIf
         J←(⊃,/0=⍴¨GOLFERS[I;])/⍳4
         GOLFERS[I;(⍴NAMES)↑J]←NAMES
         (TEETIMES GOLFERS)⎕FREPLACE GOLFID COMP
         TEETIME←DateTime.New TEETIME.(Year Month Day),
                                  3↑24 60⊤I⊃TEETIMES
         SLOT.Time←TEETIME
         SLOT.Players←(⊃,/0<⍴¨GOLFERS[I;])/GOLFERS[I;]
         R.(OK TeeTime)←1 SLOT
       ∇

   :EndClass
```

The GolfService constructor takes the name of a file in which all the data is stored. This file is initialised by method `InitFile` if it doesn't already exist.

```
      G←⎕NEW GolfService 'F:\HELP11.0\GOLFDATA'
      G
#.[Instance of GolfService]
```

The GetCourses method returns an array of Instances of the internal (nested) Class GolfCourse. Notice how the display form of each Instance is established by the GolfCourse constructor, to obtain the output display shown below.

```
      G.GetCourses
 St Andrews(1)  Hindhead(2)  Basingstoke(3)
```

All of the dates and times employ instances of the .Net type System.DateTime, and the following statements just set up some temporary variables for convenience later.

```
      ⎕←Tomorrow←(⎕NEW DateTime(3↑⎕TS)).AddDays 1
31/03/2006 00:00:00
      ⎕←TomorrowAt7←Tomorrow.AddHours 7
31/03/2006 07:00:00
```

The MakeBooking method takes between 4 and 7 parameters viz.

the code for the golf course at which the reservation is required
the date and time of the reservation
a flag to indicate whether or not the nearest available time will do
and a list of up to 4 players who wish to book that time.

The result is an Instance of the internal Class Booking. Once again, `⎕DF` is used to make the default display of these Instances meaningful. In this case, the reservation is successful.

```
      G.MakeBooking 2 TomorrowAt7 1 'Pete' 'Tiger'
 Hindhead(2)   31/03/2006 07:00:00   Pete  Tiger    OK
```

Bob, Arnie and Jack also ask to play at 7:00 but are given the 7:10 tee-time instead (4-player restriction).

```
      G.MakeBooking 2 TomorrowAt7 1 'Bob' 'Arnie' 'Jack'
 Hindhead(2)   31/03/2006 07:10:00   Bob  Arnie  Jack
OK
```

However, Pete and Tiger are joined at 7:00 by Dave and Al.

```
      G.MakeBooking 2 TomorrowAt7 1 'Dave' 'Al'
 Hindhead(2)   31/03/2006 07:00:00   Pete  Tiger  Dave  Al
OK
```

Up to now, all bookings have been made with the tee-time flexibility flag set to 1. Inflexible Jim is only interested in playing at 7:00 →

```
      G.MakeBooking 2 TomorrowAt7 0 'Jim'
 Hindhead(2)  31/03/2006 07:00:00  Not available
```

→ so his reservation fails (4-player restriction).

Finally the GetStartingSheet method is used to obtain an Instance of the internal Class StartingSheet for the given course and day.

```
      G.GetStartingSheet 2 Tomorrow
  Hindhead(2)  31/03/2006 00:00:00
  31/03/2006 07:00:00   Pete  Tiger  Dave  Al
  31/03/2006 07:10:00   Bob  Arnie  Jack
  31/03/2006 07:20:00
  ....
```

Namespace Scripts

A Namespace Script is a script that begins with a `:Namespace` statement and ends with a `:EndNamespace` statement. When a Namespace Script is fixed, it establishes an entire namespace that may contain other namespaces, functions, variables and classes.

The names of Classes defined within a Namespace Script which are parents, children, or siblings are visible both to one another and to code and expressions defined in the same script, regardless of the namespace hierarchy within it. Names of Classes which are nieces or nephews and their descendants are however not visible.

For example:

```
:Namespace a
    d←⎕NEW a1
    e←⎕NEW bb2

    :Class a1
        ∇ r←foo
          :Access Shared Public
          r←⎕NEW¨b1 b2
        ∇
    :EndClass ⍝ a1

    ∇ r←goo
      r←a1.foo
    ∇

    ∇ r←foo
      r←⎕NEW¨b1 b2
    ∇

    :Namespace b
        :Class b1
        :EndClass ⍝ b1
        :Class b2
            :Class bb2
            :EndClass ⍝ bb2
        :EndClass ⍝ b2
    :EndNamespace ⍝ b
:EndNamespace ⍝ a
```

```
      a.d
#.a.[a1]
      a.e
#.a.[bb2]
      a.foo
 #.a.[b1]  #.a.[b2]
```

Note that the names of Classes `b1` (`a.b.b1`) and `b2` (`a.b.b2`) are not visible from their “uncle” `a1` (`a.a1`).

```
      a.goo
VALUE ERROR
foo[2] r←⎕NEW¨b1 b2
```

Notice that Classes in a Namepsace Script are fixed before other objects (hence the assignments to `d` and `e` are evaluated *after* Classes `a1` and `bb2` are fixed), although the order in which Classes themselves are defined is still important if they reference one another during initialisation.

Warning: If you introduce new objects of any type (functions, variables, or classes) into a namespace defined by a script by any other means than editing the script, then these objects will be lost the next time the script is edited and fixed. Also, if you modify a variable which is defined in a script, the script will not be updated.

Namespace Script Example

The DiaryStuff example illustrates the manner in which classes may be defined and used in a Namespace script.

DiaryStuff defines two Classes named `Diary` and `DiaryEntry`.

`Diary` contains a (private) Field named `entries`, which is simply a vector of instances of `DiaryEntry`. These are 2-element vectors containing a .NET DateTime object and a description.

The `entries` Field is initialised to an empty vector of `DiaryEntry` instances which causes the invocation of the default constructor `DiaryEntry.Make0` when `Diary` is fixed. See Empty Arrays of Instances for further explanation.

The `entries` Field is referenced through the `Entry` Property, which is defined as the Default Property. This allows individual entries to be referenced and changed using indexing on a `Diary` Instance.

Note that `DiaryEntry` is defined in the script first (before `Diary`) because it is referenced by the initialisation of the `Diaries.entries` Field

```
:Namespace DiaryStuff
:Using System

    :Class DiaryEntry
        :Field Public When
        :Field Public What
        ∇ Make(ymdhm wot)
          :Access Public
          :Implements Constructor
          When What←(⎕NEW DateTime(6↑5↑ymdhm))wot
          ⎕DF⍕When What
        ∇
        ∇ Make0
          :Access Public
          :Implements Constructor
          When What←⎕NULL''
        ∇
    :EndClass ⍝ DiaryEntry
    :Class Diary
        :Field Private entries←0⍴⎕NEW DiaryEntry
        ∇ R←Add(ymdhm wot)
          :Access Public
          R←⎕NEW DiaryEntry(ymdhm wot)
          entries,←R
        ∇
        ∇ R←DoingOn ymd;X
          :Access Public
          X←,(↑entries.When.(Year Month Day))∧.=3 1⍴3↑ymd
          R←X/entries
        ∇
        ∇ R←Remove ymdhm;X
          :Access Public
          :If R←∨/X←entries.When=⎕NEW DateTime(6↑5↑ymdhm)
              entries←(~X)/entries
          :EndIf
        ∇
        :Property Numbered Default Entry
            ∇ R←Shape
              R←⍴entries
            ∇
            ∇ R←Get arg
              R←arg.Indexers⊃entries
            ∇
            ∇ Set arg
              entries[arg.Indexers]←arg.NewValue
            ∇
        :EndProperty
    :EndClass ⍝ Diary

:EndNamespace
```

Create a new instance of `Diary`.

```
      D←⎕NEW DiaryStuff.Diary
```

Add a new entry "meeting with John at 09:00 on April 30th"

```
      D.Add(2006 4 30 9 0)'Meeting with John'
 30/04/2006 09:00:00  Meeting with John
```

Add another diary entry "Dentist at 10:00 on April 30th".

```
      D.Add(2006 4 30 10 0)'Dentist'
 30/04/2006 10:00:00  Dentist
```

One of the benefits of the Namespace Script is that Classes defined within it (which are typically *related*) may be used *independently*, so we can create a stand-alone instance of `DiaryEntry`; "Doctor at 11:00"→

```
  Doc←⎕NEW DiaryStuff.DiaryEntry((2006 4 30 11 0)'Doctor')
      Doc
 30/04/2006 11:00:00  Doctor
```

→ and then use it to replace the second Diary entry with indexing:

```
      D[2]←Doc
```

and just to confirm it is there →

```
      D[2]
 30/04/2006 11:00:00  Doctor
```

What am I doing on the 30th ?

```
      D.DoingOn 2006 4 30
  30/04/2006 09:00:00  Meeting with John    ...
  ... 30/04/2006 11:00:00  Doctor
```

Remove the 11:00 appointment...

```
      D.Remove 2006 4 30 11 0
1
```

and the complete Diary is...

```
      ⍝D
  30/04/2006 09:00:00  Meeting with John
```

Class Declaration Statements

This section summarises the various declaration statements that may be included in a Class or Namespace Script. For information on other declaration statements, as they apply to functions and methods, see Function Declaration Statements.

:Interface Statement

```
:Interface <interface name>
...
:EndInterface
```

An Interface is defined by a Script containing skeleton declarations of Properties and/or Methods. The script must begin with a `:Interface Statement` and end with a `:EndInterface Statement`.

An Interface may not contain Fields.

Properties and Methods defined in an Interface, and the Class functions that implement the Interface, **may not** contain :Access Statements.

:Namespace Statement

```
:Namespace <namespace name>
...
:EndNamespace
```

A Namespace Script may be used to define an entire namespace containing other namespaces, functions, variables and Classes.

A Namespace script must begin with a `:Namespace` statement and end with a `:EndNamespace` statement.

Sub-namespaces, which may be nested, are defined by pairs of `:Namespace` and `:EndNamespace` statements within the Namespace script.

Classes are defined by pairs of `:Class` and `:EndClass` statements within the Namespace script, and these too may be nested.

The names of Classes defined within a Namespace Script are visible both to one another and to code and expressions defined in the same script, regardless of the namespace hierarchy within it.

A Namespace script is therefore particularly useful to group together Classes that refer to one another where the use of nested classes is inappropriate.

:Class Statement

```
:Class <class name><:base class name> <,interface name...>
:Include <namespace>
...
:EndClass
```

A class script begins with a `:Class` statement and ends with a `:EndClass` statement. The elements that comprise the `:Class` statement are as follows:

Element	Description
`class name`	Optionally, specifies the name of the Class, which must conform to the rules governing APL names.
`base class name`	Optionally specifies the name of a Class from which this Class is derived and whose members this Class inherits.
`interface name`	The names of one or more Interfaces which this Class supports.

A Class may import methods defined in separate plain Namespaces with one or more `:Include` statements. For further details, see Including Namespaces in Classes

Examples:

The following statements define a Class named `Penguin` that derives from (is based upon) a Class named `Animal` and which supports two Interfaces named `BirdBehaviour` and `FishBehaviour`.

```
:Class Penguin: Animal,BirdBehaviour,FishBehaviour
...
:EndClass
```

The following statements define a Class named `Penguin` that derives from (is based upon) a Class named `Animal` and includes methods defined in two separate Namespaces named `BirdStuff` and `FishStuff`.

```
:Class Penguin: Animal
:Include BirdStuff
:Include FishStuff
...
:EndClass
```

:Using Statement

`:Using <NameSpace[,Assembly]>`

This statement specifies a .NET namespace that is to be searched to resolve unqualified names of .NET types referenced by expressions in the Class.

Element	Description
`NameSpace`	Specifies a .NET namespace.
`Assembly`	Specifies the Assembly in which NameSpace is located. If the Assembly is defined in the *global assembly cache*, you need only specify its name. If not, you must specify a full or relative pathname.

If the Microsoft .Net Framework is installed, the System namespace in mscorlib.dll is automatically loaded when Dyalog APL starts. To access this namespace, it is not necessary to specify the name of the Assembly.

When the class is fixed, `⎕USING` is inherited from the surrounding space. Each `:Using` statement appends an element to `⎕USING`, with the exception of `:Using` with no argument:

If you omit `<Namespace>`, this is equivalent to clearing `⎕USING`, which means that no .NET namespaces will be searched (unless you follow this statement with additional `:Using` statements, each of which will append to `⎕USING`).

To set `⎕USING`, to a single empty character vector, which only allows references to fully qualified names of classes in `mscorlib.dll`, you must write:

```
      :Using ,
```
(note the presence of the comma)

or

```
      :Using ,mscorlib.dll
```

(i.e. specify an empty namespace name followed by no assembly, or followed by the default assembly, which is always loaded.

:Attribute Statement

```
:Attribute <Name> [ConstructorArgs]
```

The :Attribute statement is used to attach .Net Attributes to a Class or a Method.

Attributes are descriptive tags that provide additional information about programming elements. Attributes are not used by Dyalog APL but other applications can refer to the extra information in attributes to determine how these items can be used. Attributes are saved with the *metadata* of Dyalog APL .NET assemblies.

Element	Description
`Name`	The name of a .Net attribute
`ConstructorArgs`	Optional arguments for the Attribute constructor

Example

The following Class has `SerializableAttribute` and `CLSCompliantAttribute` attributes attached to the Class as a whole, and `ObsoleteAttribute` attributes attached to Methods `foo` and `goo` within it.

```
:Class c1
:using System
    :attribute SerializableAttribute
    :attribute CLSCompliantAttribute 1

    ∇ foo(p1 p2)
      :Access public instance
      :Signature foo Object,Object
      :Attribute ObsoleteAttribute
    ∇

    ∇ goo(p1 p2)
      :Access public instance
      :Signature foo Object,Object
      :Attribute ObsoleteAttribute 'Don''t use this' 1

    ∇

:EndClass ⍝ c1
```

When this Class is exported as a .Net Class, the attributes are saved in its metadata. For example, Visual Studio will warn developers if they make use of a member which has the ObsoleteAttribute.

:Access Statement

```
:Access <Private|Public><Instance|Shared><Overridable>
                                         <Override>
:Access <WebMethod>
```

The :Access statement is used to specify characteristics for Classes, Properties and Methods.

Element	Description
`Private|Public`	Specifies whether or not the (nested) Class, Property or Method is accessible from outside the Class or an Instance of the Class. The default is `Private`.
`Instance|Shared`	For a Field, specifies if there is a separate value of the Field in each Instance of the Class, or if there is only a single value that is shared between all Instances. For a Property or Method, specifies whether the code associated with the Property or Method runs in the Class or Instance..
`WebMethod`	Applies only to a Method and specifies that the method is exported as a web method. This applies only to a Class that implements a Web Service.
`Overridable`	Applies only to an Instance Method and specifies that the Method may be overridden by a Method in a higher Class. See below.
`Override`	Applies only to an Instance Method and specifies that the Method overrides the corresponding Overridable Method defined in the Base Class. See below.

Overridable/Override

Normally, a Method defined in a higher Class replaces a Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*.

However, a Method declared as being `Overridable` is replaced in situ (i.e. within its own Class) by a Method of the same name in a higher Class if that Method is itself declared with the `Override` keyword. For further information, see Superseding Base Class Methods.

Nested Classes

The :Access statement is also used to control the visibility of one Class that is defined within another (a nested Class). A Nested Class may be either `Private` or `Public`. Note that the :Access Statement must precede the definition of any Class contents..

A `Public` Nested Class is visible from outside its containing Class and may be used directly in its own right, whereas a `Private` Nested Class is not and may only be used by code inside the containing Class.

However, methods in the containing Class may return instances of Private Nested Classes and in that way expose them to the calling environment.

WebMethod

Note that `:Access WebMethod` is equivalent to:

```
:Access Public
:Attribute System.Web.Services.WebMethodAttribute
```

:Field Statement

```
:Field <Private|Public> <Instance|Shared> <ReadOnly>...
      ... FieldName <← expr>
```

A `:Field` statement is a single statement whose elements are as follows:

Element	Description
`Private|Public`	Specifies whether or not the Field is accessible from outside the Class or an Instance of the Class. The default is `Private`.
`Instance|Shared`	Specifies if there is a separate value of the Field in each Instance of the Class, or if there is only a single value that is shared between all Instances.
`ReadOnly`	If specified, this keyword prevents the value in the Field from being changed after initialisation.
`FieldName`	Specifies the name of the Field (mandatory).
`← expr`	Specifies an initial value for the Field.

Examples:

The following statement defines a Field called `Name`. It is (by default), an Instance Field so every Instance of the Class has a separate value. It is a Public Field and so may be accessed (set or retrieved) from outside an Instance.

```
:Field Public Name
```

The following statement defines a Field called `Months`.

```
:Field Shared ReadOnly Months←12↑(⎕NEW DateTimeFormatInfo)
                                 .AbbreviatedMonthNames
```

`Months` is a Shared Field so there is just a single value that is the same for every Instance of the Class. It is (by default), a Private Field and may only be referenced by code running in an Instance or in the Class itself. Furthermore, it is ReadOnly and may not be altered after initialisation. It's initial value is calculated by an expression that obtains the short month names that are appropriate for the current locale using the .Net Type DateTimeFormatInfo.

Note that Fields are initialised when a Class script is fixed by the editor or by `⎕FIX`. If the evaluation of `expr` causes an error (for example, a `VALUE ERROR`), an appropriate message will be displayed in the Status Window and `⎕FIX` will fail with a `DOMAIN ERROR`. Note that a ReadOnly Field may only be assigned a value by its `:Field` statement.

In the second example above, the expression will only succeed if `⎕USING` is set to the appropriate path, in this case System.Globalization.

:Property Section

A Property is defined by a `:Property ... :EndProperty` section in a Class Script. The syntax of the :Property Statement, and its optional `:Access` statement is as follows:

```
:Property <Simple|Numbered|Keyed> <Default> Name<,Name>...
:Access <Private|Public><Instance|Shared>
...
:EndProperty
```

Element	Description
`Name`	Specifies the name of the Property by which it is accessed. Additional Properties, sharing the same PropertyGet and/or PropertySet functions, and the same access behaviour may be specified by a comma-separated list of names.
`Simple|Numbered|Keyed`	Specifies the type of Property (see below). The default is `Simple`.
`Default`	Specifies that this Property acts as the default property for the Class when indexing is applied directly to an Instance of the Class.
`Private|Public`	Specifies whether or not the Property is accessible from outside the Class or an Instance of the Class. The default is `Private`.
`Instance|Shared`	Specifies if there is a separate value of the Property in each Instance of the Class, or if there is only a single value that is shared between all Instances.

A Simple Property is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety.

A Numbered Property behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices.

A Keyed Property is similar to a Numbered Property except that its elements are accessed via arbitrary keys instead of indices.

Numbered and Keyed Properties are designed to allow APL to perform selections and structural operations on the Property.

Within the body of a Property Section there may be:

- one or more `:Access` statements
- a single PropertyGet function.
- a single PropertySet function
- a single PropertyShape function

The three functions are identified by case-independent names `Get`, `Set` and `Shape`.

When a Class is fixed by the Editor or by `⎕FIX`, APL checks the validity of each Property section and the syntax of PropertyGet, PropertySet and PropertyShape functions within them. If anything is wrong, an error is generated and the Class is not fixed.

PropertyArguments Class

Where appropriate, APL supplies the PropertyGet and PropertySet functions with an argument that is an instance of the internal class `PropertyArguments`.

`PropertyArguments` has just 3 read-only Fields which are as follows:

`Name`	The name of the property. This is useful when one function is handling several properties.
`NewValue`	Array containing the new value for the Property or for selected element(s) of the property as specified by `Indexers`.
`Indexers`	A vector that identifies the elements of the Property that are to be referenced or assigned.

:PropertyGet Function Syntax

```
PropertyGet Syntax:     R←Get
                        R←Get ipa
```

The name of the PropertyGet function must be `Get`, but is case-independent. For example, `get`, `Get`, `gEt` and `GET` are all valid names for the PropertyGet function

The PropertyGet function must be result returning. For a Simple Property, it may be monadic or niladic. For a Numbered or Keyed Property it must be monadic.

The result `R` may be any array. However, for a Keyed Property, `R` must conform to the rank and shape specified by `ipa.Indexers` or be scalar.

If monadic, `ipa` is an instance of the internal class PropertyArguments

In all cases, `ipa.Name` contains the name of the Property being referenced and `NewValue` is undefined (`VALUE ERROR`).

If the Property is *Simple*, `ipa.Indexers` is undefined (`VALUE ERROR`).

If the Property is *Numbered*, `ipa.Indexers` is an integer vector of the same length as the rank of the property (as implied by the result of the `Shape` function) that identifies a single element of the Property whose value is to be obtained. In this case, `R` must be scalar.

If the Property is Keyed, `ipa.Indexers` is a vector containing the arrays that were specified within the square brackets in the reference expression. Specifically, `ipa.Indexers` will contain one more elements than the number of semi-colon (;) separators.

PropertySet Function Syntax

PropertySet Syntax: `Set ipa`

The name of the PropertySet function must be `Set`, but is case-independent. For example, `set`, `Set`, `sEt` and `SET` are all valid names for the PropertySet function.

The PropertySet function must be monadic and may not return a result.

`ipa` is an instance of the internal class PropertyArguments.

In all cases, `ipa.Name` contains the name of the Property being referenced and `NewValue` contains the new value(s) for the element(s) of the Property being assigned.

If the Property is *Simple*, `ipa.Indexers` is undefined (`VALUE ERROR`).

If the Property is *Numbered*, `ipa.Indexers` is an integer vector of the same length as the rank of the property (as implied by the result of the `Shape` function) that identifies a single element of the Property whose value is to be set.

If the Property is Keyed, `ipa.Indexers` is a vector containing the arrays that were specified within the square brackets in the assignment expression. Specifically, `ipa.Indexers` will contain one fewer elements than, the number of semi-colon (;) separators. If any index was elided, the corresponding element of ipa.Indexers is `⎕NULL`. However, If the Keyed Property is being assigned in its entirety, without square-bracket indexing, `ipa.Indexers` is undefined (`VALUE ERROR`).

PropertyShape Function Syntax

```
PropertyShape Syntax:  R←Shape
                       R←Shape ipa
```

The name of the PropertyShape function must be `Shape`, but is case-independent. For example, `shape`, `Shape`, `sHape` and `SHAPE` are all valid names for the PropertyShape function.

A PropertyShape function is only called if the Property is a Numbered Property.

The PropertyShape function must be niladic or monadic and must return a result.

If monadic, `ipa` is an instance of the internal class PropertyArguments. `ipa.Name` contains the name of the Property being referenced and `NewValue` and `Indexers` are undefined (`VALUE ERROR`).

The result `R` must be an integer vector or scalar that specifies the `rank` of the Property. Each element of `R` specifies the length of the corresponding dimension of the Property. Otherwise, the reference or assignment to the Property will fail with `DOMAIN ERROR`.

Note that the result `R` is used by APL to check that the number of indices corresponds to the rank of the Property and that the indices are within the bounds of its dimensions. If not, the reference or assignment to the Property will fail with `RANK ERROR` or `LENGTH ERROR`..

CHAPTER 4

Primitive Functions

Scalar Functions

There is a class of primitive functions termed SCALAR FUNCTIONS. This class is identified in Figure 4(i) below. Scalar functions are **pervasive**, ie. their properties apply at all levels of nesting. Scalar functions have the following properties:

Symbol	Monadic	Dyadic
+	Identity	Plus (Add)
-	Negative	Minus (Subtract)
×	Signum	Times (Multiply)
÷	Reciprocal	Divide
|	Magnitude	Residue
⌊	Floor	Minimum
⌈	Ceiling	Maximum
*	Exponential	Power
⍟	Natural Logarithm	Logarithm
○	Pi Times	Circular
!	Factorial	Binomial
~	Not	$
?	Roll	$
∊	Type (See Enlist)	$
∧		And
∨		Or
⍲		Nand
⍱		Nor
<		Less
≤		Less Or Equal
=		Equal
≥		Greater Or Equal
>		Greater
≠		Not Equal
$ Dyadic form is not scalar		

Figure 4(i) : Scalar primitive functions

Monadic Scalar Functions

a) The function is applied independently to each simple scalar in its argument.

Example

```
      ÷2 (1 4)
0.5  1 0.25
```

b) The function produces a result with a structure identical to its argument.

c) When applied to an empty argument, the function produces an empty result. With the exception of + and ∊, the type of this result depends on the function, not on the type of the argument. By definition + and ∊ return a result of the same type as their arguments.

Dyadic Scalar Functions

a) The function is applied independently to corresponding pairs of simple scalars in its arguments.

Examples

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

      2 (3 4) + 1 (2 3)
3  5 7

      (1 2) 3 + 4 (5 6)
 5 6  8 9
```

b) A simple scalar will be replicated to conform to the structure of the other argument. If a simple scalar in the structure of an argument corresponds to a non-simple scalar in the other argument, then the function is applied between the simple scalar and the items of the non-simple scalar. Replication of simple scalars is called SCALAR EXTENSION.

Examples

```
      10 × 2 (3 4)
20  30 40

      2 4 = 2 (4 6)
1  1 0
```

c) A simple unit is treated as a scalar for scalar extension purposes. A UNIT is a single element array of any rank. If both arguments are simple units, the argument with lower rank is extended.

Example

```
      (1 1⍴5) - 1 (2 3)
4  3 2
```

d) The function produces a result with a structure identical to that of its arguments (after scalar extensions).

e) If applied between empty arguments, the function produces a composite structure resulting from any scalar extensions, with type appropriate to the particular function. (All scalar dyadic functions return a result of numeric type.)

Examples

```
      1↑''+⍳0
0

      1↑(0⍴⊂' ' (0 0))×''
0  0 0
```

Note: The Axis operator applies to all scalar dyadic functions.

Mixed Functions

Mixed rank functions are summarised in Figure 4(ii). For convenience, they are sub-divided into five classes:

Structural	These functions change the structure of the arguments in some way.
Selection	These functions select elements from an argument.
Selector	These functions identify specific elements by a boolean map or by an ordered set of indices.
Miscellaneous	These functions transform arguments in some way, or provide information about the arguments.
Special	These functions have special properties.

In general, the structure of the result of a mixed primitive function is different from that of its arguments.

Scalar extension may apply to some, but not all, dyadic mixed functions.

Mixed primitive functions are not pervasive. The function is applied to elements of the arguments, not necessarily independently.

Examples

```
      'CAT' 'DOG' 'MOUSE'⍳⊂'DOG'
2

      3↑ 1 'TWO' 3 'FOUR'
1  TWO  3
```

Class	Symbol	Monadic	Dyadic
Structural	⍴	$	Reshape
	,	Ravel []	Catenate [] Laminate []
	⍪		Catenate First[] Laminate []
	⌽	Reverse []	Rotate []
	⊖	Reverse First[]	Rotate First[]
	⍉	Transpose	Transpose
	↑	Mix/Disclose (First) []	$
	↓	Split []	$
	⊂	Enclose []	Partitioned Enclose []
	∊	Enlist (See Type)	$
Selection	⊃	Disclose/Mix	Pick
	↑	$	Take []
	↓	$	Drop []
	/		Replicate []
	⌿		Replicate First[]
	\		Expand []
	⍀		Expand First []
	~	$	Without (Excluding)
	∩		Intersection
	∪	Unique	Union
Selector	⍳	Index Generator	Index Of
	∊	$	Membership
	⍋	Grade Up	Grade Up
	⍒	Grade Down	Grade Down
	?	$	Deal
	⍷		Find

[] Implies axis specification is optional

$ This function is in another class

Figure 4(ii) : Mixed Primitive Functions

Class	**Symbol**	**Monadic**	**Dyadic**
Miscellaneous	`ρ`	`Shape`	`$`
	`≡`	`Depth`	`Match`
	`≢`		`Not Match`
	`⍎`	`Execute`	`Execute`
	`⍕`	`Format`	`Format`
	`⊥`		`Decode (Base)`
	`⊤`		`Encode (Representation)`
	`⌹`	`Matrix Divide`	`Matrix Inverse`
Special	`→`	`Abort (Niladic)`	
	`→`	`Branch`	
	`←`	`$`	`Assignment`
	`[I]←`	`$`	`Assignment (Indexed)`
	`(I)←`		`Assignment (Selective)`
	`[]`		`Indexing`

[] Implies axis specification is optional

$ This function is in another class

Figure 4(ii) : Mixed Primitive Functions (Continued)

Conformability

The arguments of a dyadic function are said to be CONFORMABLE if the shape of each argument meets the requirements of the function, possibly after scalar extension.

Fill Elements

Some primitive functions may include fill elements in their result. The fill element for an array is the enclosed type of the disclose of the array (`⊂∊⊃Y` for array `Y`). The Type function (`∊`) replaces a numeric value with zero and a character value with `' '`.

The Disclose function (`⊃`) returns the first item of an array. If the array is empty, `⊃Y` is the PROTOTYPE of `Y`. The prototype is the type of the first element of the original array.

Primitive functions which may return an array including fill elements are Expand (`\` or `⍀`), Replicate (`/` or `⌿`), Reshape (`⍴`) and Take (`↑`).

Examples

```
      ∊⍳5
0 0 0 0 0

      ∊⊃(⍳3)('ABC')
0 0 0

      ⊂∊⊃(⍳3)('ABC')
 0 0 0

      ⊂∊⊃⊂(⍳3)('ABC')
  0 0 0

      A←'ABC' (1 2 3)
      A←0⍴A
      ⊂∊⊃A

      '   '=⊂∊⊃A
 1 1 1
```

Axis Specification

The axis operator may be applied to all scalar dyadic primitive functions and certain mixed primitive functions. An integer axis identifies a specific axis along which the function is to be applied to one or both of its arguments. If the primitive function is to be applied without an axis specification, a default axis is implied, either the first or last.

Example

```
      1 0 1/[1] 3 2⍴⍳6
1 2
5 6

      1 2 3+[2]2 3⍴10 20 30
11 22 33
11 22 33
```

Sometimes the axis value is fractional, indicating that a new axis or axes are to be created between the axes identified by the lower and upper integer bounds of the value (either of which might not exist).

Example

```
      'NAMES',[0.5]'='
NAMES
=====
```

`⎕IO` is an implicit argument of an axis specification.

Function Presentation

Scalar and mixed primitive functions are presented in alphabetical order of their descriptive names as shown in Figures 3(i) and 3(ii) respectively. Scalar functions are described in terms of single element arguments. The rules for extension are defined at the beginning of this chapter.

The class of the function is identified in the heading block. The valence of the function is implied by its syntax in the heading block.

Abort: →

This is a special case of the Branch function used in the niladic sense. If it occurs in a statement it must be the only symbol in an expression or the only symbol forming an expression in a text string to be executed by ⍎. It clears the most recently suspended statement and all of its pendent statements from the state indicator.

The Abort function has no explicit result. The function is not in the function domain of operators.

Examples

```
      ∇ F
[1]    'F[1]'
[2]    G
[3]    'F[3]'
     ∇

     ∇ G
[1]    'G[1]'
[2]    →
[3]    'G[3]'
     ∇
      F
F[1]
G[1]

      ⎕VR'VALIDATE'
     ∇ VALIDATE
 [1]   →(12=1↑⎕AI)⍴0 ⋄ 'ACCOUNT NOT AUTHORISED' ⋄ →
     ∇

      VALIDATE
ACCOUNT NOT AUTHORISED

      1↑⎕AI
52
```

Add: R←X+Y

Y must be numeric. X must be numeric. R is the arithmetic sum of X and Y. R is numeric. This function is also known as Plus.

Examples

```
      1 2 + 3 4
4 6

      1 2 + 3,⊂4 5
4  6 7
```

And, Lowest Common Multiple: R←X^Y

Case 1: X and Y are boolean

R is boolean is determined as follows:

X	Y	R
0	0	0
0	1	0
1	0	0
1	1	1

Note that the ASCII caret (^) will also be interpreted as an APL And (∧).

Example

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

Case 2: X and Y are numeric (non-boolean)

R is the lowest common multiple of X and Y.

Example

```
      15 1 2 7 ^ 35 1 4 0
105 1 4 0
```

⎕CT is an implicit argument in case 2.

Assignment: X←Y

Assignment allocates the result of the expression Y to the ***name*** or ***names*** in X.

If Y is an array expression, X must contain one or more names which are variables, system variables, or are undefined. Following assignment, the name(s) in X become variable(s) with value(s) taken from the result of the expression Y.

If X contains a single name, the variable assumes the value of Y.

The assignment arrow (or specification arrow) is often read as 'Is' or 'Gets'.

Examples

```
      A←2.3
      A
2.3

      A←⍳3
      A
1 2 3
```

More than one name may be specified in X by using vector notation. If so, Y must be a vector or a scalar. If Y is a scalar, its value is assigned to all names in X. If Y is a vector, each element of Y is assigned to the corresponding name in X.

Examples

```
      A B←2
      A
2

      B
2

      P ⎕IO Q←'TEXT' 1 (1 2 3)
      P
TEXT
      ⎕IO
1
      Q
1 2 3
```

For compatibility with IBM's APL2, the list of names specified in X may be enclosed in parentheses.

Examples

```
      (A B C)←1 2 3
      (D E)←'Hello' 'World'
```

Multiple assignments are permitted. The value of Y is carried through each assignment:

```
      I←J←K←0

      I,J,K
0 0 0
```

Function Assignment

If Y is a function expression, X must be a single name which is either undefined, or is the name of an existing function or defined operator. X may not be the name of a system function, or a primitive symbol.

Examples

```
      PLUS←+
      PLUS
+

      SUM←+/
      SUM
+/

      MEAN←{(+/⍵)÷⍴⍵}
```

Namespace Reference Assignment

If an expression evaluates to a namespace reference, or *ref,* you may assign it to a name. A name assigned to a simple scalar *ref,* has name class 9, whereas one assigned to an *array containing refs* has name class 2.

```
      'f1'⎕WC'Form'
      'ns1' ⎕NS ''

      N←ns1
      ⎕NC'N'           ⍝ name class of a scalar ref
9
      F←f1
      ⎕NC'F'           ⍝ name class of a scalar ref
9
      refs←N F         ⍝ vector of refs.
      ⎕NC'refs'        ⍝ nameclass of vector.
2
      F2←2⊃refs
      ⎕NC 'F2'
9
```

Re-Assignment

A name that already exists may be assigned a new value if the assigment will not alter its name class, or will change it from 2 to 9 or vice versa. The table of permitted re-assignments is as follow.

	Ref	Variable	Function	Operator
Ref	Yes	Yes		
Variable	Yes	Yes		
Function			Yes	Yes
Operator			Yes	Yes

Assignment (Indexed): {R}←X[I]←Y

Indexed Assignment is the Assignment function modified by the Indexing function. The phrase [I]← is treated as the function for descriptive purposes.

Y may be any array. X may be the ***name*** of any array. I must be a valid index specification. The shape of Y must conform with the shape (implied) of the indexed structure defined by I. If Y is a scalar or a unit vector it will be extended to conform. A side effect of Indexed Assignment is to change the value of the indexed elements of X.

R is the value of Y. If the result is not explicitly assigned or used it is suppressed.

⎕IO is an implicit argument of Indexed Assignment.

Three forms of indexing are permitted.

Simple Indexed Assignment

For vector X, I is a simple integer array whose items are from the set ⍳⍴R. Elements of X identified by index positions I are replaced by corresponding elements of Y.

Examples

```
      +A←⍳5
1 2 3 4 5

      A[2 3]←10 ⋄ A
1 10 10 4 5
```

The last-most element of Y is assigned when an index is repeated in I:

```
      A[2 2]←100 101 ⋄ A
1 101 10 4 5
```

For matrix X, I is composed of two simple integer arrays separated by the semicolon character (;). The arrays select indices from the rows and columns of X respectively.

Examples

```
      +B←2 3⍴'REDSUN'
RED
SUN

      B[2;2]←'O' ⋄ B
RED
SON
```

For higher-order array X, I is a series of simple integer arrays with adjacent arrays separated by a single semicolon character (;). Each array selects indices from an axis of X taken in row-major order.

Examples

```
      C
11 12 13
14 15 16

21 22 23
24 25 26

      C[1;1;3]←103 ⋄ C
11 12 103
14 15  16

21 22  23
24 25  26
```

An indexing array may be ELIDED. That is, if an indexing array is omitted from the Kth axis, the indexing vector ⍳(⍴X)[K] is implied:

```
      C[;1;2 3]←2 2⍴112 113 122 123 ⋄ C
11 112 113
14  15  16

21 122 123
24  25  26
```

```
      C[;;]←0 ⋄ C
0 0 0
0 0 0

0 0 0
0 0 0
```

Choose Indexed Assignment

The index specification `I` is a non-simple integer array. Each item identifies a single element of `X` by a set of indices with one element per axis of `X` in row-major order.

Examples

```
      C
11 12 13 14
21 22 23 24

      C[⊂1 1]←101 ⋄ C
101 12 13 14
 21 22 23 24

      C[(1 2) (2 3)]←102 203 ⋄ C
101 102  13 14
 21  22 203 24

      C[2 2⍴(1 3)(2 4)(2 1)(1 4)]←2 2⍴103 204 201 104⋄C
101 102 103 104
201  22 203 204
```

A scalar may be indexed by the enclosed empty vector:

```
      S
10

      S[⊂⍳0]←⊂'VECTOR' ⋄ S
 VECTOR

      S[⊂⍳0]←5 ⋄ S
5
```

Choose Indexed Assignment may be used very effectively in conjunction with Index Generator (`⍳`) and Structural functions in order to assign into an array:

```
      C
11 12 13 14
21 22 23 24

      ⍳⍴C
1 1  1 2  1 3  1 4
2 1  2 2  2 3  2 4
```

```
      C[1 1⍉⍳⍴C]←1 2 ⋄ C
 1 12 13 14
21  2 23 24

      C[2 ¯1↑⍳⍴C]←99 ⋄ C
 1 12 13 99
21  2 23 99
```

Reach Indexed Assignment

The index specification `I` is a non-simple integer array, each of whose items reach down to a nested element of `X`. The items of an item of `I` are simple vectors (or scalars) forming sets of indices that index arrays at successive levels of `X` starting at the top-most level. A set of indices has one element per axis at the respective level of nesting of `X` in row-major order.

Examples

```
      D←(2 3⍴⍳6)(2 2⍴'SMITH' 'JONES' 'SAM' 'BILL')

      D
 1 2 3   SMITH  JONES
 4 5 6   SAM    BILL

      ≡J←⊂2 (1 2)
¯3

      D[J]←⊂'WILLIAMS' ⋄ D
 1 2 3   SMITH  WILLIAMS
 4 5 6   SAM    BILL

      D[(1 (1 1))(2 (2 2) 1)]←10 'W' ⋄ D
 10 2 3   SMITH  WILLIAMS
  4 5 6   SAM    WILL

      E
 GREEN  YELLOW  RED

      E[⊂2 1]←'M' ⋄ E
 GREEN  MELLOW  RED
```

The context of indexing is important. In the last example, the indexing method is determined to be Reach rather than Choose since `E` is a vector, not a matrix as would be required for Choose. Observe that:

```
      ⊂2 1 ←→ ⊂(⊂2),(⊂1)
```

Assignment (Selective): (EXP X)←Y

X is the ***name*** of a variable in the workspace. EXP is an expression that **selects** elements of X. Y is an array expression. The result of the expression Y is allocated to the elements of X selected by EXP.

The following functions may appear in the selection expression. Where appropriate these functions may be used with axis [].

↑	Take
↓	Drop
,	Ravel
⌽	Reverse, Rotate
⍴	Reshape
⊃	Disclose, Pick
⍉	Transpose (Monadic and Dyadic)
/	Replicate
\	Expand
⌷	Index

Note that Mix and Split (monadic ↑ and ↓) may not be used in the selection expression.

Examples

```
      A←'HELLO'
      ((A∊'AEIOU')/A)←'*'

      A
H*LL*

      Z←3 4⍴⍳12
      (5↑,Z)←0

      Z
0  0  0  0
0  6  7  8
9 10 11 12

      MAT←3 3⍴⍳9
      (1 1⍉MAT)←0

      MAT
0 2 3
4 0 6
7 8 0
```

Binomial: R←X!Y

Y may be any number other than a negative integer. X may be any number other than a negative integer. R is numeric. An element of R is integer if corresponding elements of X and Y are integers. Binomial is defined in terms of the function Factorial for positive integer arguments:

```
      X!Y ←→ (!Y)÷(!X)×!Y-X
```

For other arguments, results are derived smoothly from the Beta function:

```
      Beta(X,Y) ←→ ÷Y×(X-1)!X+Y-1
```

For positive integer arguments, R is the number of selections of X things from Y things.

Example

```
      1 1.2 1.4 1.6 1.8 2!5
5 6.105689248 7.219424686 8.281104786 9.227916704 10
```

Branch: →Y

Y may be a scalar or vector which, if not empty, has a simple numeric scalar as its first element. The function has no explicit result. It is used to modify the normal sequence of execution of expressions or to resume execution after a statement has been interrupted. Branch is not in the function domain of operators.

The following distinct usages of the branch function occur:

	Entered in a Statement in a Defined Function	**Entered in Immediate Execution Mode**
→LINE	Continue with the specific line	Restart execution at the specific line of the most recently suspended function
→⍳0	Continue with the next expression	No effect

In a defined function, if Y is non-empty then the first element in Y specifies a statement line in the defined function to be executed next. If the line does not exist, then execution of the function is terminated. For this purpose, line 0 does not exist. (Note that statement line numbers are independent of the index origin ⎕IO).

If Y is empty, the branch function has no effect. The next expression is executed on the same line, if any, or on the next line if not. If there is no following line, the function is terminated.

The :GoTo statement may be used in place of Branch in a defined function.

Example

```
     ∇ TEST
[1]    1
[2]    →4
[3]    3
[4]    4
     ∇

      TEST
1
4
```

In general it is better to branch to a LABEL than to a line number. A label occurs in a statement followed by a colon and is assigned the value of the statement line number when the function is defined.

Example

```
     ∇ TEST
[1]    1
[2]    →FOUR
[3]    3
[4]   FOUR:4
     ∇

      TEST
1
4
```

The previous examples illustrate unconditional branching. There are numerous APL idioms which result in conditional branching. Some popular idioms are identified:

Branch Expression	**Comment**
`→TEST/L1`	`Branches to label L1 if TEST results in 1 but not if TEST results in 0.`
`→TESTρL1`	`Similar to above.`
`→TEST↑L1`	`Similar to above.`
`→L1ρ⍨TEST`	`Similar to above.`
`→L1⌈⍳TEST`	`Similar to above but only if ⎕IO←→1`
`→L1×⍳TEST`	`Similar to above but only if ⎕IO←→1`
`→(L1,L2,L3)[N]`	`Unconditional branch to a selected label.`
`→(T1,T2,T3)/L1,L2,L3`	`Branches to the first selected label dependent on tests T1,T2,T3.  If all tests result in 0, there is no branch.`
`→N⌽L1,L2,L3`	`Unconditional branch to the first label after rotation.`

A branch expression may occur within a statement including ⋄ separators:

```
[5]   →NEXTρ⍨TEST ⋄ A←A+1 ⋄ →END
[6]  NEXT:
```

In this example, the expressions `'A←A+1'` and `'→END'` are executed only if `TEST` returns the value 1. Otherwise control branches to label `NEXT`.

In immediate execution mode, the branch function permits execution to be continued within the most recently suspended function, if any, in the state indicator. If the state indicator is empty, or if the argument `Y` is the empty vector, the branch expression has no effect. If a statement line is specified which does not exist, the function is terminated. Otherwise, execution is restarted from the beginning of the specified statement line in the most recently suspended function.

Example

```
      ∇ F
[1]   1
[2]   2
[3]   3
[4]   ∇

      2 ⎕STOP'F'
      F
1

F[2]
      )SI
F[2]*
      →2
2
3
```

The system constant `⎕LC` returns a vector of the line numbers of statement lines in the state indicator, starting with that in the most recently suspended function. It is convenient to restart execution in a suspended state by the expression:

```
      →⎕LC
```

Catenate/Laminate: R←X,[K]Y

Y may be any array. X may be any array. The axis specification is optional. If specified, K must be a numeric scalar or unit vector which may have a fractional value. If not specified, the last axis is implied.

The form R←X⍪Y may be used to imply catenation along the first axis.

Two cases of the function catenate are permitted:

1. With an integer axis specification, or implied axis specification.

2. With a fractional axis specification, also called **laminate**.

Catenation with Integer or Implied Axis Specification

The arrays X and Y are joined along the required axis to form array R. A scalar or unit vector is extended to the shape of the other argument except that the required axis is restricted to a unit dimension. X and Y must have the same shape (after extension) except along the required axis, or one of the arguments may have rank one less than the other, provided that their shapes conform to the prior rule after augmenting the array of lower rank to have a unit dimension along the required axis.

The rank of R is the greater of the ranks of the arguments, but not less than 1.

Examples

```
      'FUR','LONG'
FURLONG

      1,2
1 2

      (2 4⍴'THISWEEK')⍪'='
THIS
WEEK
====

      S,[1]+⌿S←2 3⍴⍳6
1 2 3
4 5 6
5 7 9
```

If, after extension, exactly one of X and Y have a length of zero along the joined axis, then the data type of R will be that of the argument with a non-zero length. Otherwise, the data type of R will be that of X.

Lamination with Fractional Axis Specification

The arrays X and Y are joined along a new axis created before the ⌈Kth axis. The new axis has a length of 2. K must exceed ⎕IO (the index origin) minus 1, and K must be less than ⎕IO plus the greater of the ranks of X and Y. A scalar or unit vector argument is extended to the shape of the other argument. Otherwise X and Y must have the same shape.

The rank of R is one plus the greater of the ranks of X and Y.

Examples

```
      'HEADING',[0.5]'-'
HEADING
-------

      'NIGHT',[1.5]'*'
N*
I*
G*
H*
T*

      ⎕IO←0

      'HEADING,[¯0.5]'-'
HEADING
-------
```

Catenate First: R←X⍪[K]Y

The form R←X⍪Y implies catenation along the first axis whereas the form R←X,Y implies catenation along the last axis (columns). See Catenate/Laminate above.

Ceiling: R←⌈Y

Y must be numeric. R is the least integer greater than or equal to Y.

Example

```
      ⌈¯2.3  0.1  100  3.3
¯2 1 100 4
```

⎕CT is an implied argument of Ceiling.

Circular: R←X○Y

Y must be numeric. X must be an integer in the range ¯7 ≤ X ≤ 7. R is numeric.

X determines which of a family of trigonometric functions to apply to Y, from the following table:

Range	Domain	(-X) ○ Y	X	X ○ Y	Range
0≤R≤1	(|Y)≤1	(1-Y*2)*.5	0	(1-Y*2)*.5	0≤R≤1
(-○.5)<R≤○.5	(|Y)≤1	Arcsin Y	1	Sine Y	(|R)≤1
0≤R≤○1	(|Y)≤1	Arccos Y	2	Cosine Y	(|R)≤1
(_R)≤○.5		Arctan Y	3	Tangent Y	
R≥0	(|Y)≥1	(¯1+Y*2)*.5	4	(1+Y*2)*.5	
		Arcsinh Y	5	Sinh Y	
R≥0	Y≥1	Arccosh Y	6	Cosh Y	R>1
	(|Y)<1	Arctanh Y	7	Tanh Y	(|R)<1

Examples

```
      0 ¯1 ○ 1
0 1.570796327

      1○(PI←○1)÷2 3 4
1 0.8660254038  0.7071067812

      2○PI÷3
0.5
```

Deal: R←X?Y

Y must be a simple scalar or unit vector containing a non-negative integer. X must be a simple scalar or unit vector containing a non-negative integer and X≤Y.

R is an integer unit vector obtained by making X random selections from ⍳Y without repetition.

Examples

```
      13?52
7 40 24 28 12 3 36 49 20 44 2 35 1

      13?52
20 4 22 36 31 49 45 28 5 35 37 48 40
```

⎕IO and ⎕RL are implicit arguments of Deal. A side effect of Deal is to change the value of ⎕RL.

Decode: R←X⊥Y

Y must be a simple numeric array. X must be a simple numeric array. R is the numeric array which results from the evaluation of Y in the number system with radix X.

X and Y are conformable if the length of the last axis of X is the same as the length of the first axis of Y. A scalar or unit vector is extended to a vector of the required length. If the last axis of X or the first axis of Y has a length of 1, the array is extended along that axis to conform with the other argument.

The shape of R is the catenation of the shape of X less the last dimension with the shape of Y less the first dimension. That is:

```
      ⍴R ←→ (¯1↓⍴X),1↓⍴Y
```

For vector arguments, each element of X defines the ratio between the units for corresponding pairs of elements in Y. The first element of X has no effect on the result.

This function is also known as Base Value.

Examples

```
      60 60⊥3 13
193

      0 60⊥3 13
193

      60⊥3 13
193

      2⊥1 0 1 0
10
```

For higher order array arguments, each of the vectors along the last axis of X is taken as the radix vector for each of the vectors along the first axis of Y.

Examples

```
      M
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1

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

      A⊥M
0 1 1 2  1  2  2  3
0 1 2 3  4  5  6  7
0 1 3 4  9 10 12 13
0 1 4 5 16 17 20 21
```

Scalar extension may be applied:

```
      2⊥M
0 1 2 3 4 5 6 7
```

Extension along a unit axis may be applied:

```
      +A←2 1⍴2 10
 2
10
      A⊥M
0 1  2  3   4   5   6   7
0 1 10 11 100 101 110 111
```

Depth: (⎕ML) R←≡Y

Y may be any array. R is a simple integer scalar which indicates the number of levels of nesting for the simple scalar item in the array with the greatest number of levels of nesting. A simple scalar has a depth of 0. A simple array has a depth of 1. An array containing non-simple elements has a depth whose magnitude is greater than 1. The array has uniform depth if all simple scalars in the array have the same depth.

If ⎕ML<2, a negative value of R indicates non-uniform depth.

Examples

```
      ≡1
0
      ≡'A'
0
      ≡'ABC'?
1
      ≡1 'A'
1

      ⎕ML←0

      ≡A←(1 2)(3 (4 5))
¯3
      ≡¨A
1 ¯2
      ≡¨¨A
 0 0  0 1

      ⎕ML←2

      ≡A
3
      ≡¨A
1 2
      ≡¨¨A
 0 0  0 1
```

Disclose: (⎕ML) R←⊃Y or R←↑Y

The symbol chosen to represent Disclose depends on the current Migration Level.

If ⎕ML<2, Disclose is represented by the symbol: ⊃.
If ⎕ML≥2, Disclose is represented by the symbol: ↑.

Y may be any array. R is an array. If Y is non-empty, R is the value of the first item of Y taken in ravel order. If Y is empty, R is the prototype of Y.

Disclose is the inverse of enclose. The identity R←→⊃⊂R holds for all R. Disclose is also referred to as First.

Examples

```
      ⊃1
1

      ⊃2 4 6
2

      ⊃'MONDAY' 'TUESDAY'
MONDAY

      ⊃(1 (2 3))(4 (5 6))
1  2 3

      ⊃⍳0
0

      ' '=⊃''
1

      ⊃1↓⊂1,⊂2 3
0  0 0
```

Divide: R←X÷Y

Y must be a numeric array. X must be a numeric array. R is the numeric array resulting from X divided by Y. System variable ⎕DIV is an implicit argument of Divide.

If ⎕DIV=0 and Y=0 then if X=0, the result of X÷Y is 1; if X≠0 then X÷Y is a DOMAIN ERROR.

If ⎕DIV=1 and Y=0, the result of X÷Y is 0 for all values of X.

Examples

```
      2 0 5÷4 0 2
0.5 1 2.5

      ⎕DIV←1
      2 0 5÷4 0 0
0.5 0 0
```

Drop: R←X↓Y

Y may be any array. X must be a simple scalar or vector of integers. If X is a scalar, it is treated as a one-element vector. If Y is a scalar, it is treated as an array whose shape is (⍴X)⍴1 . After any scalar extensions, ⍴X must equal ⍴⍴Y (the shape of X must equal the rank of Y).

R is an array with the same rank as Y but with elements removed from the vectors along each of the axes of Y. For the Ith axis:

1. if X[I] is positive, all but the first X[I] elements of the vectors result.
2. if X[I] is negative, all but the last X[I] elements of the vectors result.

If the magnitude of X[I] exceeds the length of the Ith axis, the result is an empty array with zero length along that axis.

Examples

```
      4↓'OVERBOARD'
BOARD

      ¯5↓'OVERBOARD'
OVER

      ⍴10↓'OVERBOARD'
0

      M
ONE
FAT
FLY

      0 ¯2↓M
O
F
F

      ¯2 ¯1↓M
ON
```

Drop with Axes: R←X↓[K]Y

Y may be any non scalar array. X must be a simple integer scalar or vector. K is a vector of zero or more axes of Y.

R is an array of the elements of Y with the first or last X[i] elements removed. Elements are removed from the beginning or end of Y according to the sign of X[i].

The rank of R is the same as the rank of Y:

```
⍴⍴R ←→ ⍴⍴Y
```

The size of each axis of R is determined by the corresponding element of X:

```
      (⍴R)[,K] ←→ 0⌈(⍴Y)[,K]-|,X
```

Examples

```
⎕←M←2 3 4⍴⍳24
 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]M
 5  6  7  8
 9 10 11 12

17 18 19 20
21 22 23 24

      2↓[3]M
 3  4
 7  8
11 12

15 16
19 20
23 24

      2 1↓[3 2]M
 7  8
11 12

19 20
23 24
```

Enclose: R←⊂Y

Y may be any array. R is a scalar array whose item is the array Y. If Y is a simple scalar, R is the simple scalar unchanged. Otherwise, R has a depth whose magnitude is one greater than the magnitude of the depth of Y.

Examples

```
      ⊂1
1

      ⊂'A'
A

      ⊂1 2 3
 1 2 3

      ⊂1,⊂'CAT'
 1  CAT

      ⊂2 4⍴⍳8
 1 2 3 4
 5 6 7 8

      ⊂⍳0

      ⊂⊂⍳0

      ⊂⊂10
10
```

Enclose with Axes: R←⊂[K]Y

Y may be any array. K is a vector of zero or more axes of Y. R is an array of the elements of Y enclosed along the axes K. The shape of R is the shape of Y with the K axes removed:

```
      ⍴R ←→ (⍴Y)[(⍳⍴⍴R)~K]
```

The shape of each element of R is the shape of the K'th axes of Y:

```
      ⍴⊃R ←→ (⍴Y)[,K]
```

Examples

```
      DISPLAY A←2 3 4⍴'DUCKSWANBIRDWORMCAKESEED'
┌┌→───┐
↓↓DUCK│
││SWAN│
││BIRD│
││    │
││WORM│
││CAKE│
││SEED│
└└────┘
      DISPLAY ⊂[3]A
┌→─────────────────────┐
↓ ┌→───┐ ┌→───┐ ┌→───┐ │
│ │DUCK│ │SWAN│ │BIRD│ │
│ └────┘ └────┘ └────┘ │
│ ┌→───┐ ┌→───┐ ┌→───┐ │
│ │WORM│ │CAKE│ │SEED│ │
│ └────┘ └────┘ └────┘ │
└∊─────────────────────┘
      DISPLAY⊂[2 3]A
┌→──────────────┐
│ ┌→───┐ ┌→───┐ │
│ ↓DUCK│ ↓WORM│ │
│ │SWAN│ │CAKE│ │
│ │BIRD│ │SEED│ │
│ └────┘ └────┘ │
└∊──────────────┘
      DISPLAY ⊂[1 3]A
┌→─────────────────────┐
│ ┌→───┐ ┌→───┐ ┌→───┐ │
│ ↓DUCK│ ↓SWAN│ ↓BIRD│ │
│ │WORM│ │CAKE│ │SEED│ │
│ └────┘ └────┘ └────┘ │
└∊─────────────────────┘
```

Encode: R←X⊤Y

Y must be a simple numeric array. X must be a simple numeric array. R is the numeric array which results from the representation of Y in the number system defined by X.

The shape of R is (⍴X),⍴Y (the catenation of the shapes of X and Y).

If X is a vector or a scalar, the result for each element of Y is the value of the element expressed in the number system defined by radix X. If Y is greater than can be expressed in the number system, the result is equal to the representation of the residue (×/X)|Y. If the first element of X is 0, the value will be fully represented.

This function is also known as Representation.

Examples

```
      10⊤5 15 125
5 5 5

      0 10⊤5 15 125
0 1 12
5 5  5
```

If X is a higher order array, each of the vectors along the first axis of X is used as the radix vector for each element of Y.

Examples

```
      A
2 0  0
2 0  0
2 0  0
2 0  0
2 8  0
2 8  0
2 8 16
2 8 16

      A⊤75
0 0  0
1 0  0
0 0  0
0 0  0
1 0  0
0 1  0
1 1  4
1 3 11
```

The example shows binary, octal and hexadecimal representations of the decimal number 75.

Examples

```
      0 1⊤1.25 10.5
 1     10
 0.25   0.5

      4 13⊤13?52
 3 1 0  2 3 2 0 1  3 1 2 3 1
12 2 4 12 1 7 6 3 10 1 0 3 8
```

Enlist: (⎕ML≥1) R←∊Y

Migration level must be such that ⎕ML≥1 (otherwise see function Type).

Y may be any array, R is a simple vector created from all the elements of Y in ravel order.

Examples

```
      ⎕ML←1         ⍝  Migration level 1

      MAT←2 2⍴'MISS' 'IS' 'SIP' 'PI' ⋄ MAT
MISS  IS
SIP   PI

      ∊MAT
MISSISSIPPI

      M←1 (2 2⍴2 3 4 5) (6(7 8))
      M
1  2 3  6  7 8
   4 5
      ∊M
1 2 3 4 5 6 7 8
```

Equal: R←X=Y

Y may be any array. X may be any array. R is boolean. ⎕CT is an implicit argument of Equal.

If X and Y are character, then R is 1 if they are the same character. If X is character and Y is numeric, or vice-versa, then R is 0.

If X and Y are numeric, then R is 1 if X and Y are within comparison tolerance of each other. That is, X is considered equal to Y if (|X-Y) is not greater than ⎕CT×(|X)⌈|Y .

Examples

```
      3=3.1 3 ¯2 ¯3
0 1 0 0

      'CAT'='FAT'
0 1 1

      'CAT'=1 2 3
0 0 0

      'CAT'='C' 2 3
1 0 0

      ⎕CT←1E¯10
      1=1.000000000001
1

      1=1.0000001
0
```

Excluding: R←X~Y

`X` must be a scalar or vector. `R` is a vector of the elements of `X` excluding those elements which occur in `Y` taken in the order in which they occur in `X`.

Elements of `X` and `Y` are considered the same if `X≡Y` returns 1 for those elements.

`⎕CT` is an implicit argument of Excluding.

This function is also known as Without.

Examples

```
      'HELLO'~'GOODBYE'
HLL
      'MONDAY' 'TUESDAY' 'WEDNESDAY'~'TUESDAY' 'FRIDAY'
 MONDAY  WEDNESDAY

      5 10 15~⍳10
15
```

For performance information, see *Search Functions and Hash Tables* in Chapter 2.

Execute (Monadic): R←⍎Y

Y must be a simple character scalar or vector. If Y is an empty vector, it is treated as an empty character vector. Y is taken to be an APL statement to be executed. R is the result of the last-executed expression. If the expression has no value, then ⍎Y has no value. If Y is an empty vector or a vector containing only blanks, then ⍎Y has no value.

If Y contains a branch expression which evaluates to a non-empty result, R does not yield a result. Instead, the branch is effected in the environment from which the Execute was invoked.

Examples

```
      ⍎'2+2'
4
      4=⍎'2+2'
1
      A
1 2 3
4 5 6
      ⍎'A'
1 2 3
4 5 6
      ⍎'A←2|¯1↑⎕TS ⋄ →0⍴⍨A ⋄ A'
0
      A
0
```

Execute (Dyadic): R←X⍎Y

Y must be a simple character scalar or vector. If Y is an empty vector, it is treated as an empty character vector. X must be a namespace reference or a simple character scalar or vector representing the name of a namespace. Y is then taken to be an APL statement to be executed in namespace X. R is the result of the last-executed expression. If the expression has no value, then X⍎Y has no value.

Example

```
      ⎕SE ⍎ '⎕NL 9'
```

Expand: R←X\[K]Y

Y may be any array. X is a simple integer scalar or vector. The axis specification is optional. If present, K must be a simple integer scalar or unit vector. The value of K must be an axis of Y. If absent, the last axis of Y is implied. The form R←X⍀Y implies the first axis. If Y is a scalar, it is treated as a one-element vector.

The number of positive elements in X must be the length of Kth (or implied) axis of Y.

R is composed from the sub-arrays along the Kth axis of Y. If X[I] (an element of X) is the Jth positive element in X, then the Jth sub-array along the Kth axis of Y is replicated X[I] times. If X[I] is negative, then a sub-array of fill elements of Y (⊂∊⊃Y) is replicated |X[I] times and inserted in relative order along the Kth axis of the result. If X[I] is zero, it is treated as the value ¯1. The shape of R is the shape of Y except that the length of the Kth axis is +/1⌈|X.

Examples

```
      0\⍳0
0

      1 ¯2 3 ¯4 5\'A'
A  AAA    AAAAA

      M
1 2 3
4 5 6

      1 ¯2 2 0 1\M
1 0 0 2 2 0 3
4 0 0 5 5 0 6

      1 0 1⍀M
1 2 3
0 0 0
4 5 6

      1 0 1\[1]M
1 2 3
0 0 0
4 5 6

      1 ¯2 1\(1 2)(3 4 5)
 1 2  0 0  0 0  3 4 5
```

Expand First: R←X⍀Y

The form R←X⍀Y implies expansion along the first axis whereas the form R←X\Y implies expansion along the last axis (columns). See Expand above.

Exponential: R←*Y

Y must be numeric. R is numeric and is the Yth power of *e*, the base of natural logarithms.

Example

```
      *1 0
2.718281828 1
```

Factorial: R←!Y

Y must be numeric excluding negative integers. R is numeric. R is the product of the first Y integers for positive integer values of Y. For non-integral values of Y, !Y is equivalent to the gamma function of Y+1.

Examples

```
      !1 2 3 4 5
1 2 6 24 120

      !¯1.5 0 1.5 3.3
¯3.544907702 1 1.329340388 8.85534336
```

Find: R←X⍷Y

X and Y may be any arrays. R is a simple boolean array the same shape as Y which identifies occurrences of X within Y

If the rank of X is smaller than the rank of Y, X is treated as if it were the same rank with leading axes of size 1. For example a vector is treated as a 1-row matrix.

If the rank of X is larger than the rank of Y, no occurrences of X are found in Y.

⎕CT and ⎕IO are implicit arguments to Find.

Examples

```
      'AN'⍷'BANANA'
0 1 0 1 0 0

      'ANA'⍷'BANANA'
0 1 0 1 0 0

      'BIRDS' 'NEST'⍷'BIRDS' 'NEST' 'SOUP'
1 0 0

      MAT
IS YOU IS
OR IS YOU
IS'NT

      'IS'⍷MAT
1 0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0

      'IS YOU'⍷MAT
1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

First: (⎕ML) R←⊃Y or R←↑Y

See function Disclose.

Floor: R←⌊Y

Y must be numeric. R is the largest integer value less than or equal to Y.

Examples

```
      ⌊¯2.3 0.1 100 3.3
¯3 0 100 3

      ⌊0.5 + 0.4 0.5 0.6
0 1 1
```

⎕CT is an implicit argument of Floor.

Format (Monadic): R←⍕Y

Y may be any array. R is a simple character array which will display identically to the display produced by Y. The result is independent of ⎕PW. If Y is a simple character array, then R is Y.

Example

```
      +B←⍕A←2 6⍴'HELLO PEOPLE'
HELLO
PEOPLE

      B ≡ A
1
```

If Y is a simple numeric scalar, then R is a vector containing the formatted number without any spaces. A floating point number is formatted according to the system variable ⎕PP. ⎕PP is ignored when formatting integers.

Examples

```
      ⎕PP←5

      ⍴C←⍕⍳0
0

      ⍴C←⍕10
2

      C
10

      ⍴C←⍕12.34
5

      C
12.34

      ⍕123456789
123456789

      ⍕123.456789
123.46
```

Scaled notation is used if the magnitude of the non-integer number is too large to represent with `⎕PP` significant digits or if the number requires more than five leading zeroes after the decimal point.

Examples

```
      ⍕123456.7
1.2346E5

      ⍕0.0000001234
1.234E¯7
```

If Y is a simple numeric vector, then R is a character vector in which each element of Y is independently formatted with a single separating space between formatted elements.

Example

```
      ⍴C←⍕¯123456 1 22.5 ¯0.000000667 5.00001
27

      C
¯1.2346E5 1 22.5 ¯6.67E¯7 5
```

If Y is a simple numeric array rank higher than one, R is a character array with the same shape as Y except that the last dimension of Y is determined by the length of the formatted data. The format width is determined independently for each column of Y, such that:

a) the decimal points for floating point or scaled formats are aligned.

b) the E characters for scaled formats are aligned, with trailing zeros added to the mantissae if necessary.

c) integer formats are aligned to the left of the decimal point column, if any, or right-adjusted in the field otherwise.

d) each formatted column is separated from its neighbours by a single blank column.

e) the exponent values in scaled formats are left-adjusted to remove any blanks.

Examples

```
      C←22 ¯0.000000123 2.34 ¯212 123456 6.00002 0

      ⍴C←⍕2 2 3⍴C
2 2 29

      C
  22     ¯1.2300E¯7  2.3400E0
¯212      1.2346E5   6.0000E0

   0      2.2000E1  ¯1.2300E¯7
   2.34 ¯2.1200E2    1.2346E5
```

If Y is non-simple, and all items of Y at any depth are scalars or vectors, then R is a vector.

Examples

```
      B←⍕A←'ABC' 100 (1 2 (3 4 5)) 10

      ⍴A
4
      ≡A
¯3

      ⍴B
26
      ≡B
1

      A
 ABC  100  1 2  3 4 5   10

      B
 ABC  100  1 2  3 4 5   10
```

By replacing spaces with ^, it is clearer to see how the result of ⍕ is formed:

```
^ABC^^100^^1^2^^3^4^5^^^10
```

If Y is non-simple, and all items of Y at any depth are not scalars, then R is a matrix.

Example

```
      D←⍕C←1 'AB' (2 2⍴1+⍳4) (2 2 3⍴'CDEFGHIJKLMN')

      C
1  AB  2 3  CDE
       4 5  FGH

            IJK
            LMN

      ⍴C
4

      ≡C
¯2

      D
1  AB  2 3  CDE
       4 5  FGH

            IJK
            LMN

      ⍴D
5 16

      ≡D
1
```

By replacing spaces with ^, it is clearer to see how the result of ⍕ is formed:

```
1^^AB^^2^3^^CDE^
^^^^^^^4^5^^FGH^
^^^^^^^^^^^^^^^^
^^^^^^^^^^^^IJK^
^^^^^^^^^^^^LMN^
```

⎕PP is an implicit argument of Monadic Format.

Format (Dyadic): R←X⍕Y

Y must be a simple numeric array. X must be a simple integer scalar or vector. R is a character array displaying the array Y according to the specification X. R has rank 1⌈⍴⍴Y and ¯1↓⍴R is ¯1↓⍴Y.

Conformability requires that if X has more than two elements, then ⍴X must be 2×¯1↑⍴Y. If X contains one element, it is extended to (2×¯1↑⍴Y)⍴0,X. If X contains 2 elements, it is extended to (2×¯1↑⍴Y)⍴X.

X specifies two numbers (possibly after extension) for each column in Y. For this purpose, scalar Y is treated as a one-element vector. Each pair of numbers in X identifies a format width (W) and a format precision (P).

If P is 0, the column is to be formatted as integers.

Examples

```
      5 0 ⍕ 2 3⍴⍳6
    1    2    3
    4    5    6

      4 0⍕1.1 2 ¯4 2.547
   1   2  ¯4   3
```

If P is positive, the format is floating point with P significant digits to be displayed after the decimal point.

Example

```
      4 1⍕1.1 2 ¯4 2.547
 1.1 2.0¯4.0 2.5
```

If P is negative, scaled format is used with |P digits in the mantissa.

Example

```
      7 ¯3⍕5 15 155 1555
5.00E0 1.50E1 1.55E2 1.56E3
```

If W is 0 or absent, then the width of the corresponding columns of R are determined by the maximum width required by any element in the corresponding columns of Y, plus one separating space.

Example

```
      3⍕2 3⍴10 15.2346 ¯17.1 2 3 4
 10.000  15.235 ¯17.100
  2.000   3.000   4.000
```

If a formatted element exceeds its specified field width when W>0, the field width for that element is filled with asterisks.

Example

```
      3 0 6 2 ⍕ 3 2⍴10.1 15 1001 22.357 101 1110.1
 10 15.00
*** 22.36
101******
```

If the format precision exceeds the internal precision, low order digits are replaced by the symbol '_'.

Example

```
      26⍕2*100
1267650600228229_______________.__________________________

      ⍴26⍕2*100
59

      0 20⍕÷3
 0.33333333333333______

      0 ¯20⍕÷3
 3.333333333333333____E¯1
```

The shape of R is the same as the shape of Y except that the last dimension of Y is the sum of the field widths specified in X or deduced by the function. If Y is a scalar, the shape of R is the field width.

```
      ⍴5 2 ⍕ 2 3 4⍴⍳24
2 3 20
```

Grade Down (Monadic): R←⍒Y

Y must be a simple character or simple numeric array of rank greater than 0. R is an integer vector being the permutation of ⍳1↑⍴Y that places the sub-arrays of Y along the first axis in descending order. The indices of any set of identical sub-arrays in Y occur in R in ascending order.

If Y is a numeric array of rank greater than 1, the elements in each of the sub-arrays along the first axis are compared in ravel order with greatest weight being given to the first element and least weight being given to the last element.

Example

```
      M
2 5 3 2
3 4 1 1
2 5 4 5
2 5 3 2
2 5 3 4

      ⍒M
2 3 5 1 4

      M[⍒M;]
3 4 1 1
2 5 4 5
2 5 3 4
2 5 3 2
2 5 3 2
```

If Y is a character array, the implied collating sequence is the numerical order of the corresponding Unicode code points (Unicode Edition) or the ordering of characters in ⎕AV (Classic Edition).

⎕IO is an implicit argument of Grade Down.

Note that character arrays sort differently in the Unicode and Classic Editions.

Example

```
      M
Goldilocks
porridge
Porridge
3 bears
```

Unicode Edition	Classic Edition

`      ⍒M`
`2 3 1 4`
	`      ⍒M`
	`3 1 4 2`

`      M[⍒M;]`
`porridge`
`Porridge`
`Goldilocks`
`3 bears`
	`      M[⍒M;]`
	`Porridge`
	`Goldilocks`
	`3 bears`
	`porridge`

Grade Down (Dyadic): R←X⍒Y

`Y` must be a simple character array of rank greater than 0. `X` must be a simple character array of rank 1 or greater. `R` is a simple integer vector of shape `1↑⍴Y` containing the permutation of `⍳1↑⍴Y` that places the sub-arrays of `Y` along the first axis in descending order according to the collation sequence `X`. The indices of any set of identical sub-arrays in `Y` occur in `R` in ascending order.

If `X` is a vector, the following identity holds:

```
      X⍒Y ←→ ⍒X⍳Y
```

A left argument of rank greater than 1 allows successive resolution of duplicate orderings in the following way.

Starting with the last axis:

- The characters in the right argument are located along the current axis of the left argument. The position of the first occurrence gives the ordering value of the character.
- If a character occurs more than once in the left argument its lowest position along the current axis is used.

- If a character of the right argument does not occur in the left argument, the ordering value is one more than the maximum index of the current axis - as with dyadic iota.

The process is repeated using each axis in turn, from the last to the first, resolving duplicates until either no duplicates result or all axes have been exhausted.

For example, if index origin is 1:

Left argument:	**Right argument:**
`abc`	`ab`
`ABA`	`ac`
	`Aa`
	`Ac`

Along last axis:

```
Character:   Value:   Ordering:
ab           1 2      3
ac           1 3      =1    ⇐   duplicate ordering with
Aa           1 1      4
Ac           1 3      =1    ⇐   respect to last axis.
```

Duplicates exist, so resolve these with respect to the first axis:

```
Character:   Value:   Ordering:
ac           1 1      2
Ac           2 1      1
```

So the final row ordering is:

```
        ab                3
        ac                2
        Aa                4
        Ac                1
```

That is, the order of rows is 4 2 1 3 which corresponds to a descending row sort of:

```
        Ac                1
        ac                2
        ab                3
        Aa                4
```

Examples

```
      ⍴S1
2 27

      S1
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
 ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝÞãìðòõ

      S2
 ABCDEFGHIJKLMNOPQRSTUVWXYZÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝÞãìðòõ

      S3
 AÁBÂCÃDÇEÈFÊGËHÌIÍJÎKÏLÐMÒNÓOÔPÕQÙRÚSÛTÝUÞVãWìXðYòZõ

      S4
 ABCDEFGHIJKLMNOPQRSTUVWXYZÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝÞãìðòõ
 ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝÞãìðòõABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The following results are tabulated for comparison:

```
X        X[S1⍋X;]     X[S2⍋X;]     X[S3⍋X;]     X[S4⍋X;]

FIRÛT      TAPE         ÚAT          TAPE         TAPE
TAP        TAP          ÊIRST        TAP          TAP
RATE       RATE         TAPE         ÚAT          RATE
FÍRST      ÚAT          TAP          RATE         ÚAT
FIRST      RAT          RATE         RAT          RAT
ÚAT        MAT          RAT          MAT          MAT
ÊIRST      ÊIRST        MAT          ÊIRST        FIRÛT
TAPE       FÍRST        FÍRST        FÍRST        FÍRST
MAT        FIRÛT        FIRÛT        FIRÛT        FIRST
RAT        FIRST        FIRST        FIRST        ÊIRST
```

More Examples

```
      (2 2⍴'ABBA') ⍒ 'AB'[?5 2⍴2] ⍝ A AND B ARE EQUIVALENT
1 2 3 4 5

        DISPLAY A←2 14⍴' abcdegiklmnrt ABCDEGIKLMNRT'
.→-------------.
↓ abcdegiklmnrt|
| ABCDEGIKLMNRT|
'--------------'

      V←'Ab' 'AB' 'aba' 'ABA' 'abaca' 'abecedarian'
      V,←'Abelian' 'black' 'blackball' 'black belt'
      V,←'blacking' 'Black Mass'

      DISPLAY M←↑V
.→----------.
↓Ab         |
|AB         |
|aba        |
|ABA        |
|abaca      |
|abecedarian|
|Abelian    |
|black      |
|blackball  |
|black belt |
|blacking   |
|Black Mass |
'-----------'

      DISPLAY¨M (M[(,A)⍒M;]) (M[(,⍉A)⍒M;]) (M[A⍒M;])
 .→----------.  .→----------.  .→----------.  .→----------.
 ↓Ab         |  ↓aba        |  ↓aba        |  ↓Ab         |
 |AB         |  |abaca      |  |abaca      |  |AB         |
 |aba        |  |abecedarian|  |abecedarian|  |aba        |
 |ABA        |  |black      |  |Ab         |  |ABA        |
 |abaca      |  |black belt |  |Abelian    |  |abaca      |
 |abecedarian|  |blackball  |  |AB         |  |abecedarian|
 |Abelian    |  |blacking   |  |ABA        |  |Abelian    |
 |black      |  |Ab         |  |black      |  |black      |
 |blackball  |  |Abelian    |  |black belt |  |black belt |
 |black belt |  |AB         |  |blackball  |  |Black Mass |
 |blacking   |  |ABA        |  |blacking   |  |blackball  |
 |Black Mass |  |Black Mass |  |Black Mass |  |blacking   |
 '-----------'  '-----------'  '-----------'  '-----------'
```

`⎕IO` is an implicit argument of Grade Down.

Grade Up (Monadic): R←⍋Y

Y must be a simple character or simple numeric array of rank greater than 0. R is an integer vector being the permutation of ⍳1↑⍴Y that places the sub-arrays along the first axis in ascending order.

If Y is a numeric array of rank greater than 1, the elements in each of the sub-arrays along the first axis are compared in ravel order with greatest weight being given to the first element and least weight being given to the last element.

Examples

```
      ⍋22.5 1 15 3 ¯4
5 2 4 3 1

      M
2 3 5
1 4 7

2 3 5
1 2 6

2 3 4
5 2 4

      ⍋M
3 2 1
```

If Y is a character array, the implied collating sequence is the numerical order of the corresponding Unicode code points (Unicode Edition) or the ordering of characters in ⎕AV (Classic Edition).

⎕IO is an implicit argument of Grade Up

Note that character arrays sort differently in the Unicode and Classic Editions.

```
      M
Goldilocks
porridge
Porridge
3 bears
```

Unicode Edition	Classic Edition

`      ⍋M`
`4 1 3 2`
	`      ⍋M`
	`2 4 1 3`

`      M[⍋M;]`
`3 bears`
`Goldilocks`
`Porridge`
`porridge`
	`      M[⍋M;]`
	`porridge`
	`3 bears`
	`Goldilocks`
	`Porridge`

Grade Up (Dyadic): R←X⍋Y

`Y` must be a simple character array of rank greater than 0. `X` must be a simple character array of rank 1 or greater. `R` is a simple integer vector being the permutation of `⍳1↑⍴Y` that places the sub-arrays of `Y` along the first axis in ascending order according to the collation sequence `X`.

If `X` is a vector, the following identity holds:

```
      X⍋Y ←→ ⍋X⍳Y
```

If `X` is a higher order array, each axis of `X` represents a grading attribute in decreasing order of importance. If a character is repeated in `X`, it is treated as though it were located at the position in the array determined by the lowest index in each axis for all occurrences of the character. The character has the same weighting as the character located at the derived position in `X`.

Refer to dyadic Grade Down for an illustrative example of these principles.

Greater: R←X>Y

Y must be numeric. X must be numeric. R is boolean. R is 1 if X is greater than Y and X=Y is 0. Otherwise R is 0.

⎕CT is an implicit argument of Greater.

Examples

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

      ⎕CT←1E¯10

      1 1.00000000001 1.000000001 > 1
0 0 1
```

Greater Or Equal: R←X≥Y

Y must be numeric. X must be numeric. R is boolean. R is 1 if X is greater than Y or X=Y. Otherwise R is 0.

⎕CT is an implicit argument of Greater Or Equal.

Examples

```
      1 2 3 4 5 ≥ 3
0 0 1 1 1

      ⎕CT←1E¯10

      1≥1
1

      1≥1.00000000001
1

      1≥1.00000001
0
```

Identity: R←+Y

Y is any array. R is the same array unchanged. Identity may be used to generate a printed result after an assignment or from a function with an otherwise suppressed result.

Examples

```
      +A←⍳5
1 2 3 4 5

      +⎕EX'A'
1
```

Index: R←{X}⌷Y

Dyadic case

X must be a scalar or vector of depth ≤2 of integers each ≥⎕IO. Y may be any array. In general, the result R is similar to that obtained by square-bracket indexing in that:

```
      (I J ... ⌷ Y) ≡ Y[I;J;...]
```

The length of left argument X must be equal to the rank of right argument Y.

Note that in common with square-bracket indexing, items of the left argument X may be of any rank and that the shape of the result is the concatenation of the shapes of the items of the left argument:

```
      (⍴X⌷Y) ≡ ↑,/⍴¨X
```

Index is sometimes referred to as *squad indexing*.

Note that index may be used with selective specification.
⎕IO is an implicit argument of index.

Examples

```
      ⎕IO←1

      VEC←111 222 333 444
      3⌷VEC
333
      (⊂4 3)⌷VEC
444 333
      (⊂2 3⍴3 1 4 1 2 3)⌷VEC
333 111 444
111 222 333

            ⎕←MAT←10⊥¨⍳3 4
11 12 13 14
21 22 23 24
31 32 33 34
      2 1⌷MAT
21
      3(2 1)⌷MAT
32 31
      (2 3)1⌷MAT
21 31
      (2 3)(,1)⌷MAT
21
31
      ⍴(2 1⍴1)(3 4⍴2)⌷MAT
2 1 3 4
      ⍴⍬ ⍬⌷MAT
0 0
      (3(2 1)⌷MAT)←0 ⋄ MAT    ⍝ Selective assignment.
11 12 13 14
21 22 23 24
 0  0 33 34
```

Monadic case

If Y is an array, Y is returned.

If Y is a ref to an instance of a Class with a Default property, all elements of the Default property are returned. For example, if Item is the default property of MyClass, and imc is an Instance of MyClass, then by definition:

```
      imc.Items≡⌷imc
```

Version 11.0 issues a NONCE ERROR if the Default Property is Keyed, because in this case APL has no way to determine the list of all the elements. A future version will probably introduce a way for a class to define an ordered "key set" for a Keyed property, at which point monadic squad will be extended to return the corresponding elements.

Note that the *values* of the index set are obtained or assigned by calls to the corresponding PropertyGet and PropertySet functions. Furthermore, if there is a sequence of primitive functions to the left of the Index function, that operate on the index set itself (functions such as dyadic `⍴`, `↑`, `↓`, `⊃`) as opposed to functions that operate on the *values* of the index set (functions such as `+`, `⌈`, `⌊`, `⍴¨`), calls to the PropertyGet and PropertySet functions are deferred until the required index set has been completely determined. The full set of functions that cause deferral of calls to the PropertyGet and PropertySet functions is the same as the set of functions that applies to selective specification.

If for example, `CompFile` is an Instance of a Class with a Default Numbered Property, the expression:

```
      1↑⌽⌷CompFile
```

would only call the PropertyGet function (for `CompFile`) once, to get the value of the last element.

Note that similarly, the expression

```
     10000⍴⌷CompFile
```

would call the PropertyGet function 10000 times, on repeated indices if `CompFile` has less than 10000 elements. The deferral of access function calls is intended to be an optimisation, but can have the opposite effect. You can avoid unnecessary repetitive calls by assigning the result of `⌷` to a temporary variable.

Index with Axes: R←{X}⌷[K]Y

`X` must be a scalar or vector of depth `≤2`, of integers each `≥⎕IO`. `Y` may be any array. `K` is a simple scalar of vector specifying axes of `Y`. The length of `K` must be the same as the length of `X`:

```
     (⍴,X) ≡ ⍴,K
```

In general, the result `R` is similar to that obtained by square-bracket indexing with elided subscripts. Items of K distribute items of X along the axes of Y. For example:

```
     I J ⌷[1 3] Y  ←→  Y[I;;J]
```

Note that index with axis may be used with selective specification. `⎕IO` is an implicit argument of index with axis..

Examples

```
      ⎕IO←1

      ⎕←CUBE←10⊥¨⍳2 3 4
111 112 113 114
121 122 123 124
131 132 133 134

211 212 213 214
221 222 223 224
231 232 233 234

      2⌷[1]CUBE
211 212 213 214
221 222 223 224
231 232 233 234

      2⌷[3]CUBE
112 122 132
212 222 232

      CUBE[;;2] ≡ 2⌷[3]CUBE
1
      (1 3)4⌷[2 3]CUBE
114 134
214 234

      CUBE[;1 3;4] ≡ (1 3)4⌷[2 3]CUBE
1
      (2(1 3)⌷[1 3]CUBE)←0 ⋄ CUBE  ⍝ Selective assignment.
111 112 113 114
121 122 123 124
131 132 133 134

  0 212   0 214
  0 222   0 224
  0 232   0 234
```

Index Generator: R←⍳Y

Y must be a simple scalar or vector array of non-negative numbers. R is a numeric array composed of the set of all possible coordinates of an array of shape Y. The shape of R is Y and each element of R occurs in its self-indexing position in R. In particular, the following identity holds:

```
      ⍳Y ←→ (⍳Y)[⍳Y]
```

⎕IO is an implicit argument of Index Generator. This function is also known as Interval.

Examples

```
      ⎕IO
1
      ⍴⍳0
0
      ⍳5
1 2 3 4 5

      ⍳2 3
 1 1  1 2  1 3
 2 1  2 2  2 3

      +A←2 4⍴'MAINEXIT'
MAIN
EXIT
      A[⍳⍴A]
MAIN
EXIT


      ⎕IO←0
      ⍳5
0 1 2 3 4

      ⍳2 3
 0 0  0 1  0 2
 1 0  1 1  1 2

      A[⍳⍴A]
MAIN
EXIT
```

Index Of: R←X⍳Y

Y may be any array. X may be any vector. R is a simple integer array with the same shape as Y identifying where elements of Y are first found in X. If an element of Y cannot be found in X, then the corresponding element of R will be ⎕IO+⍴X.

Elements of X and Y are considered the same if X≡Y returns 1 for those elements.

⎕IO and ⎕CT are implicit arguments of Index Of.

Examples

```
      ⎕IO←1

      2 4 3 1 4⍳1 2 3 4 5
4 1 3 2 6

      'CAT' 'DOG' 'MOUSE'⍳'DOG' 'BIRD'
2 4
```

For performance information, see *Search Functions and Hash Tables* in Chapter 2.

Indexing: R←X[Y]

X may be any array. Y must be a valid index specification. R is an array composed of elements indexed from X and the shape of X is determined by the index specification.

Bracket Indexing does not follow the normal syntax of a dyadic function.

⎕IO is an implicit argument of Indexing.

Three forms of indexing are permitted. The form used is determined by context.

Simple Indexing

For vector X, Y is a simple integer array composed of items from the set ⍳⍴X.

R consists of elements selected according to index positions in Y. R has the same shape as Y.

Examples

```
      A←10 20 30 40 50

      A[2 3⍴1 1 1 2 2 2]
10 10 10
20 20 20

      A[3]
30

      'ONE' 'TWO' 'THREE'[2]
 TWO
```

For matrix X, Y is composed of two simple integer arrays separated by the semicolon character (;). The arrays select indices from the rows and columns of X respectively.

Examples

```
      +M←2 4⍴10×⍳8
10 20 30 40
50 60 70 80

      M[2;3]
70
```

For higher order array X, Y is composed of a simple integer array for each axis of X with adjacent arrays separated by a single semicolon character (;). The arrays select indices from the respective axes of X, taken in row-major order.

Examples

```
      +A←2 3 4⍴10×⍳24
 10  20  30  40
 50  60  70  80
 90 100 110 120

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

      A[1;1;1]
10
```

```
      A[2;3 2;4 1]
240 210
200 170
```

If an indexing array is omitted for the `K`th axis, the index vector `⍳(⍴X)[K]` is assumed for that axis.

Examples

```
      A[;2;]
 50  60  70  80
170 180 190 200

      M
10 20 30 40
50 60 70 80

      M[;]
10 20 30 40
50 60 70 80

      M[1;]
10 20 30 40

      M[;1]
10 50
```

Choose Indexing

The index specification `Y` is a non-simple array. Each item identifies a single element of `X` by a set of indices with one element per axis of `X` in row-major order.

Examples

```
      M
10 20 30 40
50 60 70 80

      M[⊂1 2]
20

      M[2 2⍴⊂2 4]
80 80
80 80

      M[(2 1)(1 2)]
50 20
```

A scalar may be indexed by the enclosed empty vector:

```
      S←'Z'
      S[3⍴⊂⍳0]
ZZZ
```

Simple and Choose indexing are indistinguishable for vector X:

```
      V←10 20 30 40

      V[⊂2]
20
      ⊂2
2
      V[2]
20
```

Reach Indexing

The index specification Y is a non-simple integer array, each of whose items reach down to a nested element of X. The items of an item of Y are simple vectors (or scalars) forming sets of indices that index arrays at successive levels of X starting at the top-most level. A set of indices has one element per axis at the respective level of nesting of X in row-major order.

Examples

```
      G←('ABC' 1)('DEF' 2)('GHI' 3)('JKL' 4)
      G←2 3⍴G,('MNO' 5)('PQR' 6)
      G
  ABC  1   DEF  2   GHI  3
  JKL  4   MNO  5   PQR  6

      G[((1 2)1)((2 3)2)]
 DEF  6

      G[2 2⍴⊂(2 2)2]
5 5
5 5
      G[⊂⊂1 1]
  ABC  1

      G[⊂1 1]
  ABC  1

      V←,G

      V[⊂⊂1]
  ABC  1

      V[⊂1]
  ABC  1

      V[1]
  ABC  1
```

Intersection: R←X∩Y

Y must be a scalar or vector. X must be a scalar or vector. A scalar X or Y is treated as a one-element vector. R is a vector composed of items occurring in both X and Y in the order of occurrence in X. If an item is repeated in X and also occurs in Y, the item is also repeated in R.

Items in X and Y are considered the same if X≡Y returns 1 for those items.

⎕CT is an implicit argument of Intersection.

Examples

```
      'ABRA'∩'CAR'
ARA

      1 'PLUS' 2 ∩ ⍳5
1 2
```

For performance information, see *Search Functions and Hash Tables* in Chapter 2.

Less: R←X<Y

Y may be any numeric array. X may be any numeric array. R is boolean. R is 1 if X is less than Y and X=Y is 0. Otherwise R is 0.

⎕CT is an implicit argument of Less.

Examples

```
      (2 4) (6 8 10) < 6
 1 1  0 0 0

      ⎕CT←1E¯10

      1 1.00000000001 1.000000001 < 1
0 0 1
```

Less Or Equal: R←X≤Y

Y may be any numeric array. X may be any numeric array. R is boolean. R is 1 if X is less than Y or X=Y. Otherwise R is 0.

⎕CT is an implicit argument of Less Or Equal.

Examples

```
      2 4 6 8 10 ≤ 6
1 1 1 0 0

      ⎕CT←1E¯10

      1  1.00000000001 1.00000001 ≤ 1
1 1 0
```

Logarithm: R←X⍟Y

Y must be a positive numeric array. X must be a positive numeric array. X cannot be 1 unless Y is also 1. R is the base X logarithm of Y.

Examples

```
      10 ⍟ 100 2
2 0.3010299957

      1 ⍟ 1
1

      2 ⍟ 1
0
```

Magnitude: R←|Y

Y may be any numeric array. R is numeric composed of the absolute (unsigned) values of Y.

Example

```
      |2 ¯3.4 0 ¯2.7
2 3.4 0 2.7
```

Match: R←X≡Y

Y may be any array. X may be any array. R is a simple boolean scalar. If X is identical to Y, then R is 1. Otherwise R is 0.

Non-empty arrays are identical if they have the same structure and the same values in all corresponding locations. Empty arrays are identical if they have the same shape and the same prototype (disclosed nested structure).

⎕CT is an implicit argument of Match.

Examples

```
      ⍬≡⍳0
1
      ''≡⍳0
0
      A
THIS
WORD

      A≡2 4⍴'THISWORD'
1
      A≡⍳10
0
      +B←A A
 THIS  THIS
 WORD  WORD

      A≡⊃B
1

      (0⍴A)≡0⍴B
0

      ' '=⊃0⍴B
1 1 1 1
1 1 1 1

      ' '=⊃0⍴A
1
```

Matrix Divide: R←X⌹Y

Y must be a simple numeric array of rank 2 or less. X must be a simple numeric array of rank 2 or less. Y must be non-singular. A scalar argument is treated as a matrix with one-element. If Y is a vector, it is treated as a single column matrix. If X is a vector, it is treated as a single column matrix. The number of rows in X and Y must be the same. Y must have at least the same number of rows as columns.

R is the result of matrix division of X by Y. That is, the matrix product Y+.×R is X. R is determined such that (X-Y+.×R)*2 is minimised.

The shape of R is (1↓⍴Y),1↓⍴X.

Examples

```
      ⎕PP←5

      B
3 1 4
1 5 9
2 6 5

      35 89 79 ⌹ B
2.1444 8.2111 5.0889

      A
35 36
89 88
79 75

      A ⌹ B
2.1444 2.1889
8.2111 7.1222
5.0889 5.5778
```

If there are more rows than columns in the right argument, the least squares solution results. In the following example, the constants a and b which provide the best fit for the set of equations represented by P = a + bQ are determined:

```
      Q
1 1
1 2
1 3
1 4
1 5
1 6

      P
12.03 8.78 6.01 3.75 ¯0.31 ¯2.79

      P⌹Q
14.941 ¯2.9609
```

Matrix Inverse: R←⌹Y

Y must be a simple array of rank 2 or less. Y must be non-singular. If Y is a scalar, it is treated as a one-element matrix. If Y is a vector, it is treated as a single-column matrix. Y must have at least the same number of rows as columns.

R is the inverse of Y if Y is a square matrix, or the left inverse of Y if Y is not a square matrix. That is, R+.×Y is an identity matrix.

The shape of R is ⌽⍴Y.

Examples

```
      M
4 ¯1
2  1

      +A←⌹M
 0.1666666667 0.1666666667
¯0.3333333333 0.6666666667
```

Within calculation accuracy, A+.×M is the identity matrix.

```
      A+.×M
1 0
0 1
```

Maximum: R←X⌈Y

Y may be any numeric array. X may be any numeric array. R is numeric. R is the larger of the numbers X and Y.

Example

```
      ¯2.01 0.1 15.3 ⌈ ¯3.2 ¯1.1 22.7
¯2.01 0.1 22.7
```

Membership: R←X∊Y

Y may be any array. X may be any array. R is boolean. An element of R is 1 if the corresponding element of X can be found in Y.

An element of X is considered identical to an element in Y if X≡Y returns 1 for those elements.

⎕CT is an implicit argument of Membership.

Examples

```
      'THIS NOUN' ∊ 'THAT WORD'
1 1 0 0 1 0 1 0 0

      'CAT' 'DOG' 'MOUSE' ∊ 'CAT' 'FOX' 'DOG' 'LLAMA'
1 1 0
```

For performance information, see *Search Functions and Hash Tables* in Chapter 2.

Minimum: R←X⌊Y

Y may be any numeric array. X may be any numeric array. R is numeric. R is the smaller of X and Y.

Example

```
      ¯2.1 0.1 15.3 ⌊ ¯3.2 1 22
¯3.2 0.1 15.3
```

Minus: R←X-Y

See function Subtract.

Mix: (⎕ML) R←↑[K]Y or R←⊃[K]Y

The symbol chosen to represent Mix depends on the current Migration Level.

If `⎕ML < 2`, Mix is represented by the symbol: `↑`.
If `⎕ML ≥ 2`, Mix is represented by the symbol: `⊃`.

`Y` may be any array. All of the items of `Y` must be scalars and/or arrays of the same rank. It is not necessary that nonscalar items have the same shape.

`K` is an optional axis specification. If present it must be a scalar or unit vector. The value of `K` must be a fractional number indicating the two axes of `Y` between which new axes are to be inserted. If absent, new ones are added at the beginning.

`R` is an array composed from the items of a `Y` assembled into a higher order array with one less level of nesting. If items of `Y` have different shapes, each is padded with the corresponding prototype to a shape that represents the greatest length along each axis of all items in `Y`. The shape of `R` is the shape of `Y` with the shape of a typical (extended) item of `Y` inserted between the `⌊K`th and the `⌈K`th axes of `Y`.

Examples

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

      ↑[0.5](1) (1 2) (1 2 3)
1 1 1
0 2 2
0 0 3

      A←('andy' 19)('geoff' 37)('pauline' 21)

      ↑A
 andy    19
 geoff   37
 pauline 21

      ↑[0.5]A
 andy  geoff  pauline
   19     37       21
```

Multiply: R←X×Y

Y may be any numeric array. X may be any numeric array. R is the arithmetic product of X and Y.

This function is also known as Times.

Example

```
      3 2 1 0 × 2 4 9 6
6 8 9 0
```

Nand: R←X⍲Y

Y must be a boolean array. X must be a boolean array. R is boolean. The value of R is the truth value of the proposition "not both X and Y", and is determined as follows:

X	Y	R
0	0	1
0	1	1
1	0	1
1	1	0

Example

```
      (0 1)(1 0) ⍲ (0 0)(1 1)
 1 1  0 1
```

Natural Logarithm: R←⍟Y

Y must be a positive numeric array. R is numeric. R is the natural (or Napierian) logarithm of Y whose base is the mathematical constant *e=2.71828....*

Example

```
      ⍟1 2
0 0.6931471806
```

Negative: R←-Y

Y may be any numeric array. R is numeric. R is the negative value of Y.

Example

```
      -4 2 0 ¯3 ¯5
¯4 ¯2 0 3 5
```

Nor: R←X⍱Y

Y must be a boolean array. X must be a boolean array. R is boolean. The value of R is the truth value of the proposition "neither X nor Y", and is determined as follows:

X	Y	R
0	0	1
0	1	0
1	0	0
1	1	0

Example

```
      0 0 1 1 ⍱ 0 1 0 1
1 0 0 0
```

Not: R←~Y

Y must be a boolean array. R is boolean. The value of R is 0 if Y is 1, and R is 1 if Y is 0.

Example

```
      ~0 1
1 0
```

Not Equal: R←X≠Y

Y may be any array. X may be any array. R is boolean. R is 0 if X=Y. Otherwise R is 1.

For boolean X and Y, the value of R is the “exclusive or” result, determined as follows:

X	Y	R
0	0	0
0	1	1
1	0	1
1	1	0

⎕CT is an implicit argument of Not Equal.

Examples

```
      1 2 3 ≠ 1.1 2 3
1 0 0

      ⎕CT←1E¯10

      1≠1 1.00000000001 1.0000001
0 0 1

      1 2 3 ≠'CAT'
1 1 1
```

Not Match: R←X≢Y

Y may be any array. X may be any array. R is a simple boolean scalar. If X is identical to Y, then R is 0. Otherwise R is 1.

Non-empty arrays are identical if they have the same structure and the same values in all corresponding locations. Empty arrays are identical if they have the same shape and the same prototype (disclosed nested structure).

⎕CT is an implicit argument of Not Match.

Examples

```
      θ≢⍳0
0
      ''≢⍳0
1

      +A←⊂(⍳3) 'ABC'
  1 2 3  ABC

      A≢(⍳3)'ABC'
1
      A≢⊂(⍳3) 'ABC'
0
      θ≢0⍴A
1
      (1↑0⍴A)≢⊂(0 0 0) '   '
1
```

Or, Greatest Common Divisor: R←X∨Y

Case 1: X and Y are boolean

R is boolean and is determined as follows:

X	Y	R
0	0	0
0	1	1
1	0	1
1	1	1

Example

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

Case 2: X and Y are numeric (non-boolean)

R is the Greatest Common Divisor of X and Y.

Example

```
      15 1 2 7 ∨ 35 1 4 0
5 1 2 7
```

⎕CT is an implicit argument in case 2.

Partition: (⎕ML≥3) R←X⊂[K]Y

Y may be any non scalar array.
X must be a simple scalar or vector of non-negative integers.

The axis specification is optional. If present, it must be a simple integer scalar or one element array representing an axis of Y. If absent, the last axis is implied.

R is an array of the elements of Y partitioned according to X.

A new partition is started in the result whenever the corresponding element in X is greater than the previous one. Items in Y corresponding to 0s in X are not included in the result.

Examples

```
      ⎕ML←3

      DISPLAY 1 1 1 2 2 3 3 3⊂'NOWISTHE'
.→-----------------.
| .→--. .→-. .→--. |
| |NOW| |IS| |THE| |
| '---' '--' '---' |
'∊-----------------'

      DISPLAY 1 1 1 0 0 3 3 3⊂'NOWISTHE'
.→-----------.
| .→--. .→--. |
| |NOW| |THE| |
| '---' '---' |
'∊-----------'

      TEXT←'   NOW     IS      THE      TIME    '
      DISPLAY(' '≠TEXT)⊂TEXT
.→-----------------------.
| .→--. .→-. .→--. .→---. |
| |NOW| |IS| |THE| |TIME| |
| '---' '--' '---' '----' |
'∊-----------------------'

      DISPLAY CMAT←⎕FMT(' ',ROWS),COLS⍪NMAT
.→-------------------------.
↓           Jan   Feb  Mar |
| Cakes       0   100  150 |
| Biscuits    0     0  350 |
| Buns        0  1000  500 |
'--------------------------'
```

```
      DISPLAY (∨⌿' '≠CMAT)⊂CMAT   ⍝ Split at blank cols.
.→------------------------------.
↓ .→-------. .→--. .→---. .→--. |
| |        | |Jan| | Feb| |Mar| |
| '--------' '---' '----' '---' |
| .→-------. .→--. .→---. .→--. |
| |Cakes   | |  0| | 100| |150| |
| '--------' '---' '----' '---' |
| .→-------. .→--. .→---. .→--. |
| |Biscuits| |  0| |   0| |350| |
| '--------' '---' '----' '---' |
| .→-------. .→--. .→---. .→--. |
| |Buns    | |  0| |1000| |500| |
| '--------' '---' '----' '---' |
'∊------------------------------'

      DISPLAY N←4 4⍴⍳16
.→----------.
↓ 1  2  3  4|
| 5  6  7  8|
| 9 10 11 12|
|13 14 15 16|
'~----------'

      DISPLAY 1 1 0 1⊂N
.→-------------.
↓ .→--.   .→.  |
| |1 2|   |4|  |
| '~--'   '~'  |
| .→--.   .→.  |
| |5 6|   |8|  |
| '~--'   '~'  |
| .→---.  .→-. |
| |9 10|  |12| |
| '~---'  '~-' |
| .→----. .→-. |
| |13 14| |16| |
| '~----' '~-' |
'∊-------------'

      DISPLAY 1 1 0 1⊂[1]N
.→------------------------.
↓ .→--.  .→--.  .→--.  .→--. |
| |1 5|  |2 6|  |3 7|  |4 8| |
| '~--'  '~--'  '~--'  '~--' |
| .→-.   .→-.   .→-.   .→-.  |
| |13|   |14|   |15|   |16|  |
| '~-'   '~-'   '~-'   '~-'  |
'∊------------------------'
```

Partitioned Enclose: (⎕ML<3) R←X⊂[K]Y

Y may be any array. X must be a simple boolean scalar or vector.

The axis specification is optional. If present, it must be a simple integer scalar or one-element vector. The value of K must be an axis of Y. If absent, the last axis of Y is implied.

X must have the same length as the Kth axis of Y. However, if X is a scalar or one-element vector, it will be extended to the length of the Kth axis of Y.

R is a vector of items selected from Y. The sub-arrays identified along the Kth axis of Y at positions corresponding to each 1 in X up to the position before the next 1 in X (or the last element of X) become the successive items of Y. The length of R is +/X (after possible extension).

Examples

```
      0 1 0 0 1 1 0 0 0 ⊂⍳9
 2 3 4  5  6 7 8 9

      1 0 1 ⊂[1] 3 4⍴⍳12
 1 2 3 4  9 10 11 12
 5 6 7 8

      1 0 0 1 ⊂[2]3 4⍴⍳12
 1   2  3   4
 5   6  7   8
 9  10 11  12
```

Pi Times: R←○Y

Y may be any numeric array. R is numeric. The value of R is the product of the mathematical constant $\pi=3.14159...$ (Pi), and Y.

Example

```
      ○0.5 1 2
1.570796327 3.141592654 6.283185307
```

Pick: R←X⊃Y

Y may be any array. X is a scalar or vector of indices of Y, viz. ⍳⍴Y. R is an item selected from the structure of Y according to X.

Elements of X select from successively deeper levels in the structure of Y. The items of X are simple integer scalars or vectors which identify a set of indices, one per axis at the particular level of nesting of Y in row-major order. Simple scalar items in Y may be picked by empty vector items in X to any arbitrary depth.

⎕IO is an implicit argument of Pick.

Examples

```
      G←('ABC' 1)('DEF' 2)('GHI' 3)('JKL' 4)

      G←2 3⍴G,('MNO' 5)('PQR' 6)

      G
  ABC  1   DEF  2   GHI  3
  JKL  4   MNO  5   PQR  6

      ((⊂2 1),1)⊃G
JKL

      (⊂2 1)⊃G
 JKL  4

      ((2 1)1 2)⊃G
K

      (5⍴⊂⍳0)⊃10
10
```

Plus: R←X+Y

See function Add.

Power: R←X*Y

Y must be a numeric array. X must be a numeric array. R is numeric. The value of R is X raised to the power of Y.

If Y is zero, R is defined to be 1.

If X is zero, Y must be non-negative.

If X is negative, and Y can be approximated as a rational number of the form P÷Q where P and Q are relatively prime integers, then:

a) if Q is even, X*Y gives a DOMAIN ERROR

b) if Q is odd and P is even, then X*Y ←→ (|X)*Y

c) if Q and P are both odd, then X*Y ←→ -(|X)*Y

If X is negative, and Y cannot be approximated as a rational number, then X*Y ←→ -(|X)*Y.

Examples

```
      2*2 ¯2
4 0.25

      9 64*0.5
3 8

      ¯27*2 3,(1 2÷3),1.2
729 ¯19683 3 9  52.19591521
```

Ravel: R←,Y

Y may be any array. R is a vector of the elements of Y taken in row-major order.

Examples

```
      M
1 2 3
4 5 6

      ,M
1 2 3 4 5 6

      A
ABC
DEF
GHI
JKL

      ,A
ABCDEFGHIJKL

      ⍴,10
1
```

Ravel with Axes: R←,[K]Y

Y may be any array.

K is either:

- A simple fractional scalar adjacent to an axis of Y, or
- A simple integer scalar or vector of axes of Y, or
- An empty vector.

Ravel with axis can be used with selective specification.

R depends on the case of K above.

If K is a fraction, the result R is an array of the same shape as Y, but with a new axis of length 1 inserted at the K'th position.

```
      ⍴⍴R ←→ 1+⍴⍴Y
      ⍴R  ←→ (1,⍴Y)[⍋K,⍳⍴⍴Y]
```

Examples

```
      ,[0.5]'ABC'
ABC
      ⍴,[0.5]'ABC'
1 3
      ,[1.5]'ABC'
A
B
C
      ⍴,[1.5]'ABC'
3 1

      MAT←3 4⍴⍳12
      ⍴,[0.5]MAT
1 3 4
      ⍴,[1.5]MAT
3 1 4
      ⍴,[2.5]MAT
3 4 1
```

If `K` is an integer scalar or vector of axes of `Y`, then:

`K` must contain contiguous axes of `Y` in ascending order.
`R` contains the elements of `Y` raveled along the indicated axes.

Note that if `K` is a scalar or single element vector, `R ←→ Y`.

```
   ⍴⍴R ←→ 1+(⍴⍴Y)-⍴,K
```

Examples

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

13 14 15 16
17 18 19 20
21 22 23 24
      ⍴M
2 3 4
```

```
      ,[1 2]M
 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]M
6 4

      ,[2 3]M
 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 3]M
2 12
```

If K is an empty vector a new last axis of length 1 is created.

```
    ⍴R ←→ (⍴Y),1
```

Examples

```
      Q1←'January' 'February' 'March'
      DISPLAY Q1
.→----------------------------.
| .→------. .→-------. .→----. |
| |January| |February| |March| |
| '-------' '--------' '-----' |
'∊----------------------------'
      DISPLAY ,[⍳0]Q1
.→-----------.
↓ .→------.  |
| |January|  |
| '-------'  |
| .→-------. |
| |February| |
| '--------' |
| .→----.    |
| |March|    |
| '-----'    |
'∊-----------'
```

Reciprocal: R←÷Y

Y must be a numeric array. R is numeric. R is the reciprocal of Y; that is 1÷Y. If ⎕DIV=0, ÷0 results in a DOMAIN ERROR. If ⎕DIV=1, ÷0 returns 0.

⎕DIV is an implicit argument of Reciprocal.

Examples

```
      ÷4 2 5
0.25 0.5 0.2

      ⎕DIV ← 1

      ÷0 0.5
0 2
```

Replicate: R←X/[K]Y

Y may be any array. X is a simple integer vector or scalar.

The axis specification is optional. If present, K must be a simple integer scalar or unit vector. The value of K must be an axis of Y. If absent, the last axis of Y is implied. The form R←X⌿Y implies the first axis of Y. If Y is a scalar, it is treated as a one-element vector.

The length of X must be the length of the Kth (or implied) axis of Y. However, if X is a scalar or one-element vector, it will be extended to the length of the Kth axis.

R is composed from sub-arrays along the Kth axis of Y. If X[I] (an element of X) is positive, then the corresponding sub-array is replicated X[I] times. If X[I] is zero, then the corresponding sub-array of Y is excluded. If X[I] is negative, then the fill element of Y (⊂∊⊃Y) is replicated |X[I] times. Each of the (replicated) sub-arrays and fill items are joined along the Kth axis in the order of occurrence. The shape of R is the shape of Y except that the length of the (implied) Kth axis is +/|X (after possible extension).

This function is sometimes called Compress when X is boolean.

Examples

```
      1 0 1 0 1/⍳5
1 3 5

      1 ¯2 3 ¯4 5/⍳5
1 0 0 3 3 3 0 0 0 0 5 5 5 5 5

      M
1 2 3
4 5 6

      2 0 1/M
1 1 3
4 4 6

      0 1⌿M
4 5 6

      0 1/[1]M
4 5 6
```

Replicate First: R←X⌿[K]Y

The form `R←X⌿Y` implies replication along the first axis of `Y`. See Replicate above.

Reshape: R←XρY

`Y` may be any array. `X` must be a simple scalar or vector of non-negative integers. `R` is an array of shape `X` whose elements are taken from `Y` in row-major sequence and repeated cyclically if required. If `Y` is empty, `R` is composed of fill elements of `Y` (`⊂∊⊃Y`). If `X` contains at least one zero, then `R` is empty. If `X` is an empty vector, then `R` is scalar.

Examples

```
      2 3ρ⍳8
1 2 3
4 5 6

      2 3ρ⍳4
1 2 3
4 1 2

      2 3ρ⍳0
0 0 0
0 0 0
```

Residue: R←X|Y

Y may be any numeric array. X may be any numeric array.

For positive arguments, R is the remainder when Y is divided by X. If X=0, R is Y. For other argument values, R is Y-N×X where N is some integer such that R lies between 0 and X, but is not equal to X.

⎕CT is an implicit argument of Residue.

Examples

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

      0.5|3.12 ¯1 ¯0.6
0.12 0 0.4

      ¯1 0 1|¯5.25 0 2.41
¯0.25 0 0.41
```

Note that the ASCII pipe (□) may also be interpreted as Residue (|).

Reverse: R←⌽[K]Y

Y may be any array. The axis specification is optional. If present, K must be an integer scalar or one-element vector. The value of K must be an axis of Y. If absent, the last axis is implied. The form R←⊖Y implies the first axis.

R is the array Y rotated about the Kth or implied axis.

Examples

```
      ⌽1 2 3 4 5
5 4 3 2 1

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

```
      ⊖M
4 5 6
1 2 3
      ⌽[1]M
4 5 6
1 2 3
```

Reverse First: R←⊖[K]Y

The form `R←⊖Y` implies reversal along the first axis. See Reverse above.

Roll: R←?Y

`Y` may be any positive integer array. `R` is an integer, pseudo-randomly selected from the integers `⍳Y` with each integer in this population having an equal chance of being selected.

`⎕IO` and `⎕RL` are implicit arguments of Roll. A side effect of Roll is to change the value of `⎕RL`.

Examples

```
      ?9 9 9
2 7 5
```

Rotate: R←X⌽[K]Y

`Y` may be any array. `X` must be a simple integer array. The axis specification is optional. If present, `K` must be a simple integer scalar or one-element vector. The value of `K` must be an axis of `Y`. If absent, the last axis of `Y` is implied. The form `R←X⊖Y` implies the first axis.

If `Y` is a scalar, it is treated as a one-element vector. `X` must have the same shape as the rank of `Y` excluding the `K`th dimension. If `X` is a scalar or one-element vector, it will be extended to conform. If `Y` is a vector, then `X` may be a scalar or a one-element vector.

`R` is an array with the same shape as `Y`, with the elements of each of the vectors along the `K`th axis of `Y` rotated by the value of the corresponding element of `X`. If the value is positive, the rotation is in the sense of right to left. If the value is negative, the rotation is in the sense of left to right.

Examples

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

      M
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

      I
0 1¯1 0
0 3 2 1
      I⌽[2]M
 1  6  7  4
 5  2  3  8

 9 14 11 16
13 10 15 12

      J
2 ¯3
3 ¯2
      J⌽M
 3  4  1  2
 6  7  8  5

12  9 10 11
15 16 13 14
```

Rotate First: R←X⊖[K]Y

The form R←X⊖Y implies rotation along the first axis. See Rotate above.

Shape: R←⍴Y

Y may be any array. R is a non-negative integer vector whose elements are the dimensions of Y. If Y is a scalar, then R is an empty vector. The rank of Y is given by ⍴⍴Y.

Examples

```
      ⍴10

      ⍴'CAT'
3

      ⍴3 4⍴⍳12
3 4

      +G←(2 3⍴⍳6)('CAT' 'MOUSE' 'FLEA')
 1 2 3   CAT  MOUSE  FLEA
 4 5 6

      ⍴G
2

      ⍴⍴G
1

      ⍴¨G
 2 3  3

      ⍴¨¨G
    3  5  4
```

Signum: R←×Y

Y may be any numeric array. R is an integer array whose value indicates whether the value of Y is negative (¯1), zero (0) or positive (1).

Example

```
      ×¯15.3 0 101
¯1 0 1
```

Split: R←↓[K]Y

Y may be any array. The axis specification is optional. If present, K must be a simple integer scalar or one-element vector. The value of K must be an axis of Y. If absent, the last axis is implied.

The items of R are the sub-arrays of Y along the Kth axis. R is a scalar if Y is a scalar. Otherwise R is an array whose rank is ¯1+⍴⍴Y and whose shape is (K≠⍳⍴⍴Y)/⍴Y.

Examples

```
      ↓3 4⍴'MINDTHATSTEP'
 MIND  THAT  STEP

      ↓2 5⍴⍳10
 1 2 3 4 5  6 7 8 9 10

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

Subtract: R←X-Y

Y may be any numeric array. X may be any numeric array. R is numeric. The value of R is the difference between X and Y.

This function is also known as Minus.

Example

```
      3 ¯2 4 0 - 2 1 ¯2 4
1 ¯3 6 ¯4
```

Take: R←X↑Y

Y may be any array. X must be a simple integer scalar or vector.

If Y is a scalar, it is treated as a one-element array of shape (⍴,X)⍴1. The length of X must be the same as the rank of Y. However, if X is a scalar or a one-element vector, it will be extended to conform.

R is an array of the same rank as Y (after possible extension), and of shape |X. If X[I] (an element of X) is positive, then X[I] sub-arrays are taken from the beginning of the Ith axis of Y. If X[I] is negative, then X[I] sub-arrays are taken from the end of the Ith axis of Y.

If more elements are taken than exist on axis I, then the extra positions in R are filled with the fill element of Y (⊂∊⊃Y).

Examples

```
      5↑'ABCDEF'
ABCDE

      5↑1 2 3
1 2 3 0 0

      ¯5↑1 2 3
0 0 1 2 3

      5↑(⍳3) (⍳4) (⍳5)
 1 2 3  1 2 3 4  1 2 3 4 5  0 0 0  0 0 0

      M
1 2 3 4
5 6 7 8

      2 3↑M
1 2 3
5 6 7

      ¯1 ¯2↑M
7 8
```

Take with Axes: R←X↑[K]Y

Y may be any non scalar array. X must be a simple integer scalar or vector. K is a vector of zero or more axes of Y.

R is an array of the first or last elements of Y taken along the axes K depending on whether the corresponding element of X is positive or negative respectively.

The rank of R is the same as the rank of Y:

```
      ⍴⍴R ←→ ⍴⍴Y
```

The size of each axis of R is determined by the corresponding element of X:

```
      (⍴R)[,K] ←→ |,X
```

Examples

```
      ⎕←M←2 3 4⍴⍳24
 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↑[2]M
 1  2  3  4
 5  6  7  8

13 14 15 16
17 18 19 20

      2↑[3]M
 1  2
 5  6
 9 10

13 14
17 18
21 22
```

```
          2 ¯2↑[3 2]M
  5  6
  9 10

 17 18
 21 22
```

Times: R←X×Y

See function Multiply.

Transpose (Monadic): R←⍉Y

`Y` may be any array. `R` is an array of shape `⌽⍴Y`, similar to `Y` with the order of the axes reversed.

Examples

```
      M
1 2 3
4 5 6

      ⍉M
1 4
2 5
3 6
```

Transpose (Dyadic): R←X⍉Y

`Y` may be any array. `X` must be a simple scalar or vector whose elements are included in the set `⍳⍴⍴Y`. Integer values in `X` may be repeated but all integers in the set `⍳⌈/X` must be included. Also the number of elements in `X` must not exceed the rank of `Y`.

`R` is an array formed by the transposition of the axes of `Y` as specified by `X`. The `I`th element of `X` gives the new position for the `I`th axis of `Y`. If `X` repositions two or more axes of `Y` to the same axis, the elements used to fill this axis are those whose indices on the relevant axes of `Y` are equal.

`⎕IO` is an implicit argument of Dyadic Transpose.

Examples

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

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

      2 1 3⍉A
 1  2  3  4
13 14 15 16

 5  6  7  8
17 18 19 20

 9 10 11 12
21 22 23 24

      1 1 1⍉A
1 18

      1 1 2⍉A
 1  2  3  4
17 18 19 20
```

Type: (⎕ML<1) R←∊Y

`Migration level must be such that ⎕ML<1 (otherwise ∊ means Enlist).`

`Y` may be any array. `R` is an array with the same shape and structure as `Y` in which a numeric value is replaced by 0 and a character value is replaced by `' '`.

Examples

```
      ∊(2 3⍴⍳6)(1 4⍴'TEXT')
 0 0 0
 0 0 0

      ' '=∊'X'
1
```

Union: R←X∪Y

Y must be a vector. X must be a vector. If either argument is a scalar, it is treated as a one-element vector. R is a vector of the elements of X catenated with the elements of Y which are not found in X.

Items in X and Y are considered the same if X≡Y returns 1 for those items.

⎕CT is an implicit argument of Union.

Examples

```
      'WASH' ∪ 'SHOUT'
WASHOUT

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

For performance information, see *Search Functions and Hash Tables* in Chapter 2.

Unique: R←∪Y

Y must be a vector. R is a vector of the elements of Y omitting non-unique elements after the first.

⎕CT is an implicit argument of Unique.

Examples

```
      ∪ 'CAT' 'DOG' 'CAT' 'MOUSE' 'DOG' 'FOX'
 CAT  DOG  MOUSE  FOX

      ∪ 22 10 22 22 21 10 5 10
22 10 21 5
```

Without: R←X~Y

See function Excluding.

Zilde: R←⍬

The empty vector (`⍳0`) may be represented by the numeric constant `⍬` called ZILDE.

CHAPTER 5

Primitive Operators

Operator Syntax

Operators take one or two operands. An operator with one operand is monadic. The operand of a monadic operator is to the left of the operator. An operator with two operands is dyadic. Both operands are required for a dyadic operator.

Operators have long scope to the left. That is, the left operand is the longest function or array expression to its left (see *Chapter 1*). A dyadic operator has short scope on the right. Right scope may be extended by the use of parentheses.

An operand may be an array, a primitive function, a system function, a defined function or a derived function. An array may be the result of an array expression.

An operator with its operand(s) forms a DERIVED FUNCTION. The derived function may be monadic or dyadic and it may or may not return an explicit result.

Examples

```
      +/⍳5
15
      (*∘2)⍳3
1 4 9

      PLUS ← + ⋄ TIMES ← ×
      1 PLUS.TIMES 2
2

      ⎕NL 2
A
X
      ⎕EX¨↓⎕NL 2
      ⎕NL 2
```

Axis Specification

Some operators may include an axis specification. Axis is itself an operator. However the effect of axis is described for each operator where its specification is permitted. `⎕IO` is an implicit argument of the function derived from the Axis operator.

The description for each operator follows in alphabetical sequence. The valence of the derived function is specifically identified to the right of the heading block.

Class of Operator	**Name**	**Producing Monadic derived function**	**Producing Dyadic derived function**
Monadic	`Assignment`		`Xf←Y`
	`Assignment`		`X[I]f←Y`
	`Assignment`		`(EXP X)f←Y`
	`Commute`		`Xf⍨Y`
	`Each`	`f¨Y`	`Xf¨Y`
	`Reduction`	`f/Y  [ ]`	
		`f⌿Y  [ ]`	
	`Scan`	`f\Y  [ ]`	
		`f⍀Y  [ ]`	
	`Spawn`	`f&Y`	`Xf&Y`
Dyadic	`Axis`	`f[B]Y`	`Xf[B]Y`
	`Composition`	`f∘gY`	`Xf∘gY`
		`A∘gY`	
		`(f∘B)Y`	
	`Inner Product`		`Xf.gY`
	`Outer Product`		`X∘.gY`
`[]` Indicates optional axis specification			

Figure 5(i) : Primitive Operators

Operator Presentation

Monadic and Dyadic primitive operators are presented in alphabetical order of their descriptive names as shown in Figure 5(i).

The valence of the operator and the derived function are implied by the syntax in the heading block.

Assignment (Modified): {R}←Xf←Y

f may be any dyadic function which returns an explicit result. Y may be any array whose items are appropriate to function f. X must be the ***name*** of an existing array whose items are appropriate to function f.

R is the "pass-through" value, that is, the value of Y. If the result of the derived function is not assigned or used, there is no explicit result.

The effect of the derived function is to reset the value of the array named by X to the result of XfY.

Examples

```
      A
1 2 3 4 5

      A+←10

      A
11 12 13 14 15

      ⎕←A×←2
2
      A
22 24 26 28 30

      vec←¯4+9?9 ⋄ vec
3 5 1 ¯1 ¯2 4 0 ¯3 2
      vec/⍨←vec>0 ⋄vec
3 5 1 4 2
```

Assignment (Indexed Modified): {R}←X[I]f←Y

f may be any dyadic function which returns an explicit result. Y may be any array whose items are appropriate to function f. X must be the ***name*** of an existing array. I must be a valid index specification. The items of the indexed portion of X must be appropriate to function f.

R is the “pass-through” value, that is, the value of Y. If the result of the derived function is not assigned or used, there is no explicit result.

The effect of the derived function is to reset the indexed elements of X, that is X[I], to the result of X[I]fY. This result must have the same shape as X[I].

Examples

```
      A
1 2 3 4 5

      +A[2 4]+←1
1

      A
1 3 3 5 5

      A[3]÷←2

      A
1 3 1.5 5 5
```

If an index is repeated, function f will be applied to the successive values of the indexed elements of X, taking the index occurrences in left-to-right order.

Example

```
      B←5⍴0

      B[2 4 1 2 1 4 2 4 1 3]+←1

      B
3 3 1 3 0
```

Assignment (Selective Modified): {R}←(EXP X)f←Y

f may be any dyadic function which returns an explicit result. Y may be any array whose items are appropriate to function f. X must be the ***name*** of an existing array. EXP is an expression that **selects** elements of X. (See Selective Assignment in *Chapter 4* for a list of allowed selection functions.) The selected elements of X must be appropriate to function f.

R is the "pass-through" value, that is, the value of Y. If the result of the derived function is not assigned or used, there is no explicit result.

The effect of the derived function is to reset the selected elements of X to the result of X[I]fY where X[I] defines the elements of X selected by EXP.

Example

```
      A
12 36 23 78 30

      ((A>30)/A) ×← 100
      A
12 3600 23 7800 30
```

Axis (with Monadic Operand): R←f[B]Y

`f` must be a monadic primitive mixed function taken from those shown in Figure 5(ii) below, or a function derived from the operators Reduction (`/`) or Scan (`\`). `B` must be a numeric scalar or vector. `Y` may be any array whose items are appropriate to function `f`. Axis does not follow the normal syntax of an operator.

Function	Name	Range of B
`⌽ or ⊖`	`Reverse`	`B∊⍳⍴⍴Y`
`↑`	`Mix`	`(0≠1|B)^(B>⎕IO-1)^(B<⎕IO+⍴⍴Y)`
`↓`	`Split`	`B∊⍳⍴⍴Y`
`,`	`Ravel`	`fraction, or zero or more axes of Y`
`⊂`	`Enclose`	`(B≡⍳0)∨(^/B∊⍳⍴⍴Y)`

Figure 5(ii) : Primitive monadic mixed functions with optional axis.

In most cases, `B` is required to be an integer which identifies a specific axis of `Y`. An exception occurs when f is the Mix function (`↑`) in which case `B` is a fractional value whose lower and upper integer bounds select an adjacent pair of axes of `Y` or an extreme axis of `Y`. For Ravel (`,`) and Enclose (`⊂`) , `B` can be a **vector** of two or more axes.

`⎕IO` is an implicit argument of the derived function which determines the meaning of `B`.

Examples

```
      ⌽[1]2 3⍴⍳6
4 5 6
1 2 3

      ↑[.1]'ONE' 'TWO'
OT
NW
EO
```

Axis (with Dyadic Operand): R←Xf[B]Y

`f` must be a dyadic primitive scalar function, or a dyadic primitive mixed function taken from Figure 5(iii) below. `B` must be a numeric scalar or vector. `X` and `Y` may be any arrays whose items are appropriate to function `f`. Axis does not follow the normal syntax of an operator.

Function	**Name**	**Range of B**
`/ or ⌿`	`Replicate`	`B∊⍳⍴⍴Y`
`\ or ⍀`	`Expand`	`B∊⍳⍴⍴Y`
`⊂`	`Partitioned Enclose`	`B∊⍳⍴⍴Y`
`⌽ or ⊖`	`Rotate`	`B∊⍳⍴⍴Y`
`, or ⍪`	`Catenate/ Laminate`	`(0≠1|B)^(B>⎕IO-1)^(B<⎕IO+(⍴⍴X)⌈⍴⍴Y)`
`↑`	`Take`	`zero or more axes of Y`
`↓`	`Drop`	`zero or more axes of Y`

Figure 5(iii) : Primitive dyadic mixed functions with optional axis.

In most cases, `B` must be an integer value identifying the axis of `X` and `Y` along which function `f` is to be applied. Exceptionally, `B` must be a fractional value for the Laminate function (`,`) whose upper and lower integer bounds identify a pair of axes or an extreme axis of `X` and `Y`. For Take (`↑`) and Drop (`↓`) , `B` can be a **vector** of two or more axes.

`⎕IO` is an implicit argument of the derived function which determines the meaning of `B`.

Examples

```
      1 4 5 =[1] 3 2⍴⍳6
1 0
0 1
1 0

      2 ¯2 1/[2]2 3⍴'ABCDEF'
AA  C
DD  F

      'ABC',[1.1]'='
A=
B=
C=

      'ABC',[0.1]'='
ABC
===

      ⎕IO←0

      'ABC',[¯0.5]'='
ABC
===
```

Axis with Scalar Dyadic Functions

The axis operator `[X]` can take a scalar dyadic function as operand. This has the effect of 'stretching' a lower rank array to fit a higher rank one. The arguments must be conformable along the specified axis (or axes) with elements of the lower rank array being replicated along the other axes.

For example, if `H` is the higher rank array, `L` the lower rank one, `X` is an axis specification, and `f` a scalar dyadic function, then the expressions `Hf[X]L` and `Lf[X]H` are conformable if `(⍴L)←→(⍴H)[X]`. Each element of L is replicated along the remaining `(⍴H)~X` axes of `H`.

In the special case where both arguments have the same rank, the right one will play the role of the higher rank array. if `R` is the right argument, `L` the left argument, `X` is an axis specification and `f` a scalar dyadic function, then the expression `Lf[X]R` is conformable if `(⍴L)←→(⍴R)[X]`.

Examples

```
      mat
10 20 30
40 50 60

      mat+[1]1 2        ⍝ add along first axis
11 21 31
42 52 62

      mat+[2]1 2 3      ⍝ add along last axis
11 22 33
41 52 63

      cube
 100  200  300
 400  500  600

 700  800  900
1000 1100 1200

      cube+[1]1 2
 101  201  301
 401  501  601

 702  802  902
1002 1102 1202

      cube+[3]1 2 3
 101  202  303
 401  502  603

 701  802  903
1001 1102 1203

      cube+[2 3]mat
 110  220  330
 440  550  660

 710  820  930
1040 1150 1260

      cube+[1 3]mat
 110  220  330
 410  520  630

 740  850  960
1040 1150 1260
```

Commute: {R}←Xf⍨Y

f may be any dyadic function. X and Y may be any arrays whose items are appropriate to function f.

The derived function is equivalent to YfX. The derived function need not return a result.

If left argument X is omitted, the right argument Y is duplicated in its place, i.e.

```
  f⍨Y ←→ Y f⍨Y
```

Examples

```
      N
3 2 5 4 6 1 3

      N/⍨2|N
3 5 1 3

      ρ⍨3
3 3 3

      mean←+/∘(÷∘ρ⍨) ⍝ mean of a vector
      mean ⍳10
5.5
```

The following statements are equivalent:

```
      F/⍨←I
      F←F/⍨I
      F←I/F
```

Commute often eliminates the need for parentheses

Composition (Form I): {R}←f∘gY

f may be any monadic function. g may be any monadic function which returns a result. Y may be any array whose items are appropriate to function g. The items of gY must be appropriate to function f.

The derived function is equivalent to fgY. The derived function need not return a result.

Composition allows functions to be *glued* together to build up more complex functions.

Examples

```
      RANK ← ρ∘ρ
      RANK ¨ 'JOANNE' (2 3ρι6)
 1  2

      +/∘ι¨2 4 6
3 10 21


      ⎕VR'SUM'
     ∇ R←SUM X
[1]    R←+/X
     ∇

      SUM∘ι¨2 4 6
3 10 21
```

Composition (Form II): {R}←A∘gY

g may be any dyadic function. A may be any array whose items are appropriate to function g. Y may be any array whose items are appropriate to function g.

The derived function is equivalent to AgY. The derived function need not return a result.

Examples

```
      2 2∘⍴ ¨ 'AB'
 AA  BB
 AA  BB

      SINE ← 1∘○

      SINE 10 20 30
¯0.5440211109 0.9129452507 ¯0.9880316241
```

The following example uses Composition Forms I and II to list functions in the workspace:

```
      ⎕NL 3
ADD
PLUS

      ⎕∘←∘⎕VR¨↓⎕NL 3
     ∇ ADD X
[1]    →LABρ⍨0≠⎕NC'SUM' ⋄ SUM←0
[2]   LAB:SUM←SUM++/X
     ∇
     ∇ R←A PLUS B
[1]    R←A+B
     ∇
```

Composition (Form III): {R}←(f∘B)Y

f may be any dyadic function. B may be any array whose items are appropriate to function f. Y may be any array whose items are appropriate to function f.

The derived function is equivalent to YfB. The derived function need not return a result.

Examples

```
      (*∘0.5)4 16 25
2 4 5

      SQRT ← *∘.5

      SQRT 4 16 25
2 4 5
```

The parentheses are required in order to distinguish between the operand B and the argument Y.

Composition (Form IV): {R}←Xf∘gY

f may be any dyadic function. g may be any monadic function which returns a result. Y may be any array whose items are appropriate to function g. Also gY must return a result whose items are appropriate as the right argument of function f. X may be any array whose items are appropriate to function f.

The derived function is equivalent to XfgY. The derived function need not return a result.

Examples

```
      +∘÷/40⍴1       ⍝ Golden Ratio! (Bob Smith)
1.618033989

      0,∘⍳¨⍳5
0 1  0 1 2  0 1 2 3  0 1 2 3 4  0 1 2 3 4 5
```

Each (with Monadic Operand): {R}←f¨Y

f may be any monadic function. Y may be any array, each of whose items are separately appropriate to function f.

The derived function applies function f separately to each item of Y. The derived function need not return a result. If a result is returned, R has the same shape as Y, and its elements are the items produced by the application of function f to the corresponding items of Y.

If Y is empty, the derived function is applied once to the prototype of Y, and the shape of R is the shape of Y.

Examples

```
      G←('TOM' (⍳3))('DICK' (⍳4))('HARRY' (⍳5))
      ⍴G
3
      ⍴¨G
 2  2  2

      ⍴¨¨G
  3  3    4  4    5  5

      +⎕FX¨('FOO1' 'A←1')('FOO2' 'A←2')
 FOO1 FOO2
```

Each (with Dyadic Operand): {R}←Xf¨Y

`f` may be any dyadic function. `X` and `Y` may be any arrays whose corresponding items (after scalar extension) are appropriate to function `f` when applied separately.

The derived function is applied separately to each pair of corresponding elements of `X` and `Y`. If `X` or `Y` is a scalar or single-element array, it will be extended to conform with the other argument. The derived function need not produce an explicit result. If a result is returned, `R` has the same shape as `Y` (after possible scalar extension) whose elements are the items produced by the application of the derived function to the corresponding items of `X` and `Y`.

If `X` or `Y` is empty and scalar conformable, the derived function is applied once to the prototypes of `X` and `Y`, and the shape of `R` is determined by the rules for scalar conformability.

Examples

```
      +G←(1 (2 3))(4 (5 6))(8 9)10
 1  2 3   4  5 6   9 8  10

      1⌽¨G
  2 3  1   5 6  4   9 8  10

      1⌽¨¨G
 1  3 2   4  6 5   8 9  10

      1⌽¨¨¨G
 1  2 3   4  5 6   8 9  10

      1 2 3 4↑¨G
 1  4  5 6   8 9 0  10 0 0 0

      'ABC',¨'XYZ'
 AX  BY  CZ
```

Inner Product: R←Xf.gY

f must be a dyadic function. g may be any dyadic function which returns a result. The last axis of X must have the same length as the first axis of Y.

The result of the derived function has shape (¯1↓⍴X),1↓⍴Y. Each item of R is the result of f/xg¨y where x and y are typical vectors taken from all the combinations of vectors along the last axis of X and the first axis of Y respectively.

Function f (and the derived function) need not return a result in the exceptional case when 2=¯1↑⍴X. In all other cases, function f must return a result.

If the result of xg¨y is empty, for any x and y, a DOMAIN ERROR will be reported unless function f is a primitive scalar dyadic function with an identity element shown in Figure 5(iv).

Examples

```
      1 2 3+.×10 12 14
76

      1 2 3 PLUS.TIMES 10 12 14
76

      +/1 2 3×10 12 14
76

      NAMES
HENRY
WILLIAM
JAMES
SEBASTIAN

      NAMES^.='WILLIAM  '
0 1 0 0
```

Outer Product: {R}←X∘.gY

g may be any dyadic function. The left operand of the operator is the symbol ∘. X and Y may be any arrays whose elements are appropriate to the function g.

Function g is applied to all combinations of the elements of X and Y. If function g returns a result, the shape of R is (⍴X),⍴Y. Each element of R is the item returned by function g when applied to the particular combination of elements of X and Y.

Examples

```
      1 2 3∘.×10 20 30 40
10 20 30  40
20 40 60  80
30 60 90 120

      1 2 3∘.⍴'AB'
 A     B
 AA    BB
 AAA   BBB

      1 2∘.,1 2 3
 1 1  1 2  1 3
 2 1  2 2  2 3

      (⍳3)∘.=⍳3
1 0 0
0 1 0
0 0 1
```

Power Operator: {R}←{X}(f⍣g)Y

If right operand `g` is a numeric integer scalar, power applies its left operand function `f` cumulatively `g` times to its argument. In particular, `g` may be boolean 0 or 1 for conditional function application.

If right operand `g` is a scalar-boolean-returning dyadic *function*, then left operand function `f` is applied repeatedly **until** `((f Y) g Y)` or until a strong interrupt occurs. In particular, if `g` is `=` or `≡`, the result is sometimes termed a *fixpoint* of `f`.

If a left argument `X` is present, it is bound as left argument to left operand function `f`:

```
X (f ⍣ g) Y → (X∘f ⍣ g) Y
```

A *negative* right operand `g` applies the *inverse* of the operand function `f`, `(|g)` times. In this case, `f` may be a primitive function or an expression of primitive functions combined with primitive operators:

`∘`	compose
`¨`	each
`∘.`	outer product
`⍨`	commute
`[]`	axis
`\`	scan
`⍣`	power

Defined, dynamic and some primitive functions do not have an inverse. In this case, a negative argument `g` generates `DOMAIN ERROR`.

Examples

```
    (,∘⊂∘,⍣(1=≡,vec))vec    ⍝ ravel-enclose if simple.

    a b c←1 0 1{(⊂⍣⍺)⍵}¨abc ⍝ enclose first and last.

    cap←{(⍺⍺⍣⍺)⍵}           ⍝ conditional application.

    a b c←1 0 1⊂cap¨abc     ⍝ enclose first and last.

    succ←1∘+                ⍝ successor function.

    (succ⍣4)10              ⍝ fourth successor of 10.
14
    (succ⍣¯3)10             ⍝ third predecessor of 10.
7
    1+∘÷⍣=1                 ⍝ fixpoint: golden mean.
1.618033989

    f←(32∘+)∘(×∘1.8)        ⍝ Fahrenheit from Celsius.
    f 0 100
32 212

    c←f⍣¯1                  ⍝ c is Inverse of f.
    c 32 212                ⍝ Celsius from Fahrenheit.
0 100

    invs←{(⍺⍺⍣¯1)⍵}         ⍝ inverse operator.

    +\invs 1 3 6 10         ⍝ scan inverse.
1 2 3 4

    2∘⊥invs 9               ⍝ decode inverse.
1 0 0 1

    dual←{⍵⍵⍣¯1 ⍺⍺ ⍵⍵ ⍵}    ⍝ dual operator.

    mean←{(+/⍵)÷⍴⍵}         ⍝ mean function.

    mean dual⍟ 1 2 3 4 5    ⍝ geometric mean.
2.605171085

    +/dual÷ 1 2 3 4 5       ⍝ parallel resistance.
0.4379562044

    mean dual(×⍨)1 2 3 4 5  ⍝ root-mean-square.
3.31662479

    ⍉dual↑ 'hello' 'world'  ⍝ vector transpose.
 hw  eo  lr  ll  od
```

Reduction: R←f/[K]Y

f must be a dyadic function. Y may be any array whose items in the sub-arrays along the Kth axis are appropriate to function f.

The axis specification is optional. If present, K must identify an axis of Y. If absent, the last axis of Y is implied. The form R←f⌿Y implies the first axis of Y.

R is an array formed by applying function f between items of the vectors along the Kth (or implied) axis of Y.

Function	APL	Identity
Plus (Add)	+	0
Minus (Subtract)	-	0
Times (Multiply)	×	1
Divide	÷	1
Residue	|	0
Minimum	⌊	M
Maximum	⌈	¯M
Power	*	1
Binomial	!	1
And	∧	1
Or	∨	0
Less	<	0
Less Or Equal	≤	1
Equal	=	1
Greater	>	0
Greater Or Equal	≥	1
Not Equal	≠	0
Encode	⊤	0
Union	∪	⍬
Replicate	/⌿	1
Expand	\⍀	1
Rotate	⌽⊖	0
M is the largest number which is representable on the machine.		

Figure 5(iv) : Identity Elements

For a typical vector `Y`, the result is:

```
      ⊂(1⊃Y)f(2⊃Y)f......f(n⊃Y)
```

The shape of `R` is the shape of `Y` excluding the `K`th axis. If `Y` is a scalar then `R` is a scalar. If the length of the `K`th axis is 1, then `R` is the same as `Y`. If the length of the `K`th axis is 0, then `DOMAIN ERROR` is reported unless function `f` occurs in Figure 5(iv), in which case its identity element is returned in each element of the result.

Examples

```
      ∨/0 0 1 0 0 1 0
1

      M
1 2 3
4 5 6

      +/M
6 15

      +⌿M
5 7 9

      +/[1]M
5 7 9

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

      ,/'ONE' 'NESS'
 ONENESS

      +/⍳0
0

      ,/''
DOMAIN ERROR
      ,/''
      ^
```

Reduce First: R←f⌿Y

The form `R←f⌿Y` implies reduction along the first axis of `Y`. See Reduce above.

Reduce N-Wise: R←Xf/[K]Y

f must be a dyadic function. X must be a simple scalar or one-item integer array. Y may be any array whose sub-arrays along the Kth axis are appropriate to function f.

The axis specification is optional. If present, K must identify an axis of Y. If absent, the last axis of Y is implied. The form R←Xf⌿Y implies the first axis of Y.

R is an array formed by applying function f between items of sub-vectors of length X taken from vectors along the Kth (or implied) axis of Y.

X can be thought of as the width of a 'window' which moves along vectors drawn from the Kth axis of Y.

If X is zero, the result is a (⍴Y)+(⍴⍴Y)=⍳⍴⍴Y array of identity elements for the function f. See Figure 5(iv).

If X is negative, each sub-vector is reversed before being reduced.

Examples

```
      ⍳4
1 2 3 4

      3+/⍳4         ⍝ (1+2+3) (2+3+4)
6 9
      2+/⍳4         ⍝ (1+2) (2+3) (3+4)
3 5 7
      1+/⍳4         ⍝ (1) (2) (3) (4)
1 2 3 4

      0+/⍳4         ⍝ Identity element for +
0 0 0 0 0
      0×/⍳4         ⍝ Identity element for ×
1 1 1 1 1

      2,/⍳4         ⍝ (1,2) (2,3) (3,4)
 1 2  2 3  3 4
      ¯2,/⍳4        ⍝ (2,1) (3,2) (4,3)
 2 1  3 2  4 3
```

Scan: R←f\[K]Y

f may be any dyadic function that returns a result. Y may be any array whose items in the sub-arrays along the Kth axis are appropriate to the function f.

The axis specification is optional. If present, K must identify an axis of Y. If absent, the last axis of Y is implied. The form R←f⍀Y implies the first axis of Y.

R is an array formed by successive reductions along the Kth axis of Y. If V is a typical vector taken from the Kth axis of Y, then the Ith element of the result is determined as f/I↑V.

The shape of R is the same as the shape of Y. If Y is an empty array, then R is the same empty array.

Examples

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

      ^\1 1 1 0 1 1 1
1 1 1 0 0 0 0

      +\1 2 3 4 5
1 3 6 10 15

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

```
      M
1 2 3
4 5 6

      +\M
1 3  6
4 9 15

      +⍀M
1 2 3
5 7 9

      +\[1]M
1 2 3
5 7 9

      ,\'ABC'
A AB  ABC

      T←'ONE(TWO) BOOK(S)'

      ≠\T∊'()'
0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 0

      ((T∊'()')⍱≠\T∊'()')/T
ONE BOOK
```

Scan First: R←f⍀Y

The form `R←f⍀Y` implies scan along the first axis of `Y`. See Scan above.

Spawn: {R}←{X}f&Y

& is a monadic operator with an ambivalent derived function. & spawns a new thread in which f is applied to its argument Y (monadic case) or between its arguments X and Y (dyadic case). The shy result of this application is the number of the newly created thread.

When function f terminates, its result (if any), the **thread result**, is returned. If the thread number is the subject of an active ⎕TSYNC, the thread result appears as the result of ⎕TSYNC. If no ⎕TSYNC is in effect, the thread result is displayed in the session in the normal fashion.

Note that & can be used in conjunction with the **each** operator ¨ to launch many threads in parallel.

Examples

```
      ÷&4          ⍝ Reciprocal in background
0.25

      ⎕←÷&4        ⍝ Show thread number
1
0.25

      FOO&88       ⍝ Spawn monadic function.

      2 FOO&3      ⍝ dyadic

      {NIL}&0      ⍝ niladic

      ⍎&'NIL'      ⍝ ..

      X.GOO&99     ⍝ thread in remote space.

      ⍎&'⎕dl 2'    ⍝ Execute async expression.

      'NS'⍎&'FOO'  ⍝ .. remote .. .. ..

      PRT&¨↓⎕nl 9  ⍝ PRT spaces in parallel.
```

CHAPTER 6

System Functions & Variables

System Functions, Variables, Constants and Namespaces provide information and services within the APL environment. Their case-insensitive names begin with ⎕.

⍞	⎕	⎕Á	⎕A	⎕AI
⎕AN	⎕ARBIN	⎕ARBOUT	⎕AT	⎕AV
⎕AVU	⎕BASE	⎕CLASS	⎕CLEAR	⎕CMD
⎕CR	⎕CS	⎕CT	⎕CY	⎕D
⎕DF	⎕DIV	⎕DL	⎕DM	⎕DQ
⎕DR	⎕ED	⎕EM	⎕EN	⎕EX
⎕EXCEPTION	⎕EXPORT	⎕FAPPEND	⎕FAVAIL	⎕FCOPY
⎕FCREATE	⎕FDROP	⎕FERASE	⎕FHOLD	⎕FIX
⎕FLIB	⎕FMT	⎕FNAMES	⎕FNUMS	⎕FPROPS
⎕FRDAC	⎕FRDCI	⎕FREAD	⎕FRENAME	⎕FREPLACE
⎕FRESIZE	⎕FSIZE	⎕FSTAC	⎕FSTIE	⎕FTIE
⎕FUNTIE	⎕FX	⎕INSTANCES	⎕IO	⎕KL
⎕LC	⎕LOAD	⎕LOCK	⎕LX	⎕MAP
⎕ML	⎕MONITOR	⎕NA	⎕NAPPEND	⎕NC
⎕NCREATE	⎕NERASE	⎕NEW	⎕NL	⎕NLOCK
⎕NNAMES	⎕NNUMS	⎕NQ	⎕NR	⎕NREAD
⎕NRENAME	⎕NREPLACE	⎕NRESIZE	⎕NS	⎕NSI
⎕NSIZE	⎕NTIE	⎕NULL	⎕NUNTIE	⎕NXLATE
⎕OFF	⎕OR	⎕PATH	⎕PFKEY	⎕PP
⎕PW	⎕REFS	⎕RL	⎕RTL	⎕SAVE
⎕SD	⎕SE	⎕SH	⎕SHADOW	⎕SI
⎕SIGNAL	⎕SIZE	⎕SM	⎕SR	⎕SRC
⎕STACK	⎕STATE	⎕STOP	⎕SVC	⎕SVO
⎕SVQ	⎕SVR	⎕SVS	⎕TC	⎕TCNUMS

⎕TGET	⎕THIS	⎕TID	⎕TKILL	⎕TNAME
⎕TNUMS	⎕TPOOL	⎕TPUT	⎕TRACE	⎕TRAP
⎕TREQ	⎕TS	⎕TSYNC	⎕UCS	⎕USING
⎕VFI	⎕VR	⎕WA	⎕WC	⎕WG
⎕WN	⎕WS	⎕WSID	⎕WX	⎕XSI
⎕XT				

System Variables

System variables retain information used by the system in some way, usually as implicit arguments to functions.

The characteristics of an array assigned to a system variable must be appropriate, otherwise an error will be reported immediately.

Example

```
      ⎕IO←3
DOMAIN ERROR
      ⎕IO←3
      ^
```

System variables may be localised by inclusion in the header line of a defined function or in the argument list of the system function ⎕SHADOW. When a system variable is localised, it retains its previous value until it is assigned a new one. This feature is known as "pass-through localisation". The exception to this rule is ⎕TRAP.

A system variable can never be undefined. Default values are assigned to all system variables in a clear workspace.

Unless otherwise stated, system variables are associated with namespaces.

Name	Description	Scope
⍞	Character Input/Output	Session
⎕	Evaluated Input/Output	Session
⎕AVU	Atomic Vector - Unicode	Namespace
⎕CT	Comparison Tolerance	Namespace
⎕DIV	Division Method	Namespace
⎕IO	Index Origin	Namespace
⎕LX	Latent Expression	Workspace
⎕ML	Migration Level	Namespace
⎕PATH	Search Path	Session
⎕PP	Print Precision	Namespace
⎕PW	Print Width	Session
⎕RL	Random Link	Namespace
⎕RTL	Response Time Limit	Namespace
⎕SM	Screen Map	Workspace
⎕TRAP	Event Trap	Workspace
⎕USING	Microsoft .Net Search Path	Namespace
⎕WSID	Workspace Identification	Workspace
⎕WX	Window Expose	Namespace

In other words, ⎕, ⍞, ⎕SE, ⎕PATH and ⎕PW relate to the session. ⎕LX, ⎕SM, ⎕TRAP and ⎕WSID relate to the active workspace. All the other system variables relate to the current namespace.

Session	**Workspace**	**Namespace**
⍞	⎕LX	⎕AVU
⎕	⎕SM	⎕CT
⎕PATH	⎕TRAP	⎕DIV
⎕PW	⎕WSID	⎕IO
⎕SE		⎕ML
		⎕PP
		⎕RL
		⎕RTL
		⎕USING
		⎕WX

System Namespaces

⎕SE is currently the only system namespace.

System Constants

System constants, which can be regarded as niladic system functions, return information from the system. They have distinguished names, beginning with the quad symbol, ⎕. A system constant may **not** be assigned a value. System constants may not be localised or erased. System constants are summarised in the following table:

Name	**Description**
⎕Á	Underscored Alphabetic upper case characters
⎕A	Alphabetic upper case characters
⎕AI	Account Information
⎕AN	Account Name
⎕AV	Atomic Vector
⎕D	Digits
⎕EN	Event Number
⎕EXCEPTION	Reports the most recent Microsoft .net Exception
⎕LC	Line Count
⎕NULL	Null Item
⎕SD	Screen (or window) Dimensions
⎕TC	Terminal Control (backspace, linefeed, newline)
⎕TS	Time Stamp
⎕WA	Workspace Available

System Functions

System functions provide various services related to both the APL and the external environment. System functions have distinguished names beginning with the ⎕ symbol. They are implicitly available in a clear workspace.

The following Figure identifies system functions divided into relevant categories. Each function is described in alphabetical order in this chapter

System Commands

These functions closely emulate system commands (see *Chapter 6*)

Name	Description
`⎕CLEAR`	Clear workspace (WS)
`⎕CY`	Copy objects into active WS
`⎕EX`	Expunge objects
`⎕LOAD`	Load a saved WS
`⎕NL`	Name List
`⎕OFF`	End the session
`⎕SAVE`	Save the active WS

External Environment

These functions provide access to the the external environment, such as file systems, Operating System facilities, and input/output devices.

Name	Description
`⎕ARBIN`	Arbitrary Input
`⎕ARBOUT`	Arbitrary Output
`⎕CMD`	Execute the Windows Command Processor or another program
`⎕CMD`	Start a Windows AP
`⎕MAP`	Map a file
`⎕NA`	Declare a DLL function
`⎕SH`	Execute a UNIX command or another program
`⎕SH`	Start a UNIX AP

Defined Functions and Operators

These functions provide services related to defined functions and operators.

Name	Description
⎕AT	Object Attributes
⎕CR	Canonical Representation
⎕CS	Change Space
⎕ED	Edit one or more objects
⎕EXPORT	Export objects
⎕FX	Fix definition
⎕LOCK	Lock a function
⎕MONITOR	Monitor set
⎕MONITOR	Monitor query
⎕NR	Nested Representation
⎕NS	Create Namespace
⎕OR	Object Representation
⎕PATH	Search Path
⎕REFS	Local References
⎕SHADOW	Shadow names
⎕STOP	Set Stop vector
⎕STOP	Query Stop vector
⎕THIS	This Space
⎕TRACE	Set Trace vector
⎕TRACE	Query Trace vector
⎕VR	Vector Representation

Error Trapping

These functions are associated with event trapping and the system variable ⎕TRAP.

Name	Description
⎕EM	Event Messages
⎕SIGNAL	Signal event

Shared Variables

These functions provide the means to communicate between APL tasks and with other applications.

Name	Description
⎕SVC	Set access Control
⎕SVC	Query access Control
⎕SVO	Shared Variable Offer
⎕SVO	Query degree of coupling
⎕SVQ	Shared Variable Query
⎕SVR	Retract offer
⎕SVS	Query Shared Variable State

Object Oriented Programming

These functions provide object oriented programming features.

Name	Description
⎕BASE	Base Class
⎕CLASS	Class
⎕DF	Display Formct
⎕FIX	Fix
⎕INSTANCES	Instances
⎕NEW	New Instance
⎕SRC	Source
⎕THIS	This

Graphical User Interface

These functions provide access to GUI components.

Name	Description
⎕DQ	Await and process events
⎕NQ	Place an event on the Queue
⎕WC	Create GUI object
⎕WG	Get GUI object properties
⎕WN	Query GUI object Names
⎕WS	Set GUI object properties
⎕WX	Expose GUI property names

External Variables

These functions are associated with using external variables.

Name	Description
⎕XT	Associate External variable
⎕XT	Query External variable
⎕FHOLD	External variable Hold

Component Files

The functions provide the means to store and retrieve data on APL Component Files. See *User Guide* for further details.

Name	Description
⎕FAPPEND	Append a component to File
⎕FAVAIL	File system Availability
⎕FCOPY	Copy a File
⎕FCREATE	Create a File
⎕FDROP	Drop a block of components
⎕FERASE	Erase a File
⎕FHOLD	File Hold
⎕FLIB	List File Library
⎕FNAMES	Names of tied Files
⎕FNUMS	Tie Numbers of tied Files
⎕FPROPS	File Properties
⎕FRDAC	Read File Access matrix
⎕FRDCI	Read Component Information
⎕FREAD	Read a component from File
⎕FRENAME	Rename a File
⎕FREPLACE	Replace a component on File
⎕FRESIZE	File Resize
⎕FSIZE	File Size
⎕FSTAC	Set File Access matrix
⎕FSTIE	Share-Tie a File
⎕FTIE	Tie a File exclusively
⎕FUNTIE	Untie Files

Native Files

The functions provide the means to store and retrieve data on native files.

Name	Description
⎕NAPPEND	Append to File
⎕NCREATE	Create a File
⎕NERASE	Erase a File
⎕NLOCK	Lock a region of a file
⎕NNAMES	Names of tied Files
⎕NNUMS	Tie Numbers of tied Files
⎕NREAD	Read from File
⎕NRENAME	Rename a File
⎕NREPLACE	Replace data on File
⎕NRESIZE	File Resize
⎕NSIZE	File Size
⎕NTIE	Tie a File exclusively
⎕NUNTIE	Untie Files
⎕NXLATE	Specify Translation Table

Threads

These functions are associated with threads created using the Spawn operator (&).

Name	Description
⎕TGET	Get Tokens
⎕TID	Current Thread Identity
⎕TCNUMS	Thread Child Numbers
⎕TKILL	Kill Threads
⎕TNAME	Current Thread Name
⎕TNUMS	Thread Numbers
⎕TPOOL	Token Pool
⎕TPUT	Put Tokens
⎕TREQ	Token Requests
⎕TSYNC	Wait for Threads to Terminate

Miscellaneous

These functions provide various miscellaneous services.

Name	Description
⎕AVU	Atomic Vestor - Unicode
⎕DL	Delay execution
⎕DM	Diagnostic Message
⎕FMT	Resolve display
⎕FMT	Format array
⎕KL	Key Labels
⎕NC	Name Classification
⎕NSI	Namespace Indicator
⎕PFKEY	Programmable Function Keys
⎕SI	State Indicator
⎕SIZE	Size of objects
⎕SR	Screen Read
⎕STACK	Report Stack
⎕STATE	Return State of an object
⎕UCS	Unicode Convert
⎕VFI	Verify and Fix numerics
⎕XSI	Extended State Indicator

Programming Reference A-Z

There follows an alphabetical list of system functions and variables which are available in **any** Dyalog APL workspace. Apart from quote-quad (⍞) below, their names all begin with the quad symbol (⎕).

Character Input/Output: ⍞

⍞ is a variable which communicates between the user's terminal and APL. Its behaviour depends on whether it is being assigned or referenced.

When ⍞ is assigned with a vector or a scalar, the array is displayed without the normal ending new-line character. Successive assignments of vectors or scalars to ⍞ without any intervening input or output cause the arrays to be displayed on the same output line.

Example

```
      ⍞←'2+2' ⋄ ⍞←'=' ⋄ ⍞←4
2+2=4
```

Output through ⍞ is independent of the print width in ⎕PW. The way in which lines exceeding the print width of the terminal is treated is dependent on the characteristics of the terminal. Numeric output is formatted in the same manner as direct output (see *Display of Arrays* in *Chapter 1*).

When ⍞ is assigned with a higher-order array, the output is displayed in the same manner as for direct output except that the print width ⎕PW is ignored.

When ⍞ is referenced, terminal input is expected without any specific prompt, and the response is returned as a character vector.

If the ⍞ request was preceded by one or more assignments to ⍞ without any intervening input or output, the last (or only) line of the output characters are returned as part of the response.

Example

```
      mat←↑⌽⍞⍞⍞⍞⍞
```

Examples

```
      ⍞←'OPTION : ' ⋄ R←⍞
OPTION : INPUT

      R
OPTION : INPUT

      ⍴R
14
```

The output of simple arrays of rank greater than 1 through ⍞ includes a new-line character at the end of each line. Input through ⍞ includes the preceding output through ⍞ since the last new-line character. The result from ⍞, including the prior output, is limited to 256 characters.

A soft interrupt causes an `INPUT INTERRUPT` error if entered while ⍞ is awaiting input, and execution is then suspended (unless the interrupt is trapped):

```
      R←⍞
(Interrupt)
INPUT INTERRUPT
```

A time limit is imposed on input through ⍞ if `⎕RTL` is set to a non-zero value:

```
      ⎕RTL←5 ⋄ ⍞←'PASSWORD ? ' ⋄ R←⍞
PASSWORD ?
TIMEOUT
      ⎕RTL←5 ⋄ ⍞←'PASSWORD : ' ⋄ R←⍞
                                   ^
```

The `TIMEOUT` interrupt is a trappable event.

Evaluated Input/Output: ⎕

⎕ is a variable which communicates between the users terminal and APL. Its behaviour depends on whether it is being assigned or referenced.

When ⎕ is assigned an array, the array is displayed at the terminal in exactly the same form as is direct output (see *Display of Arrays* in *Chapter 1*).

Example

```
      ⎕←2+⍳5
3 4 5 6 7

      ⎕←2 4⍴'WINEMART'
WINE
MART
```

When ⎕ is referenced, a prompt (`⎕:`) is displayed at the terminal, and input is requested. The response is evaluated and an array is returned if the result is valid. If an error occurs in the evaluation, the error is reported as normal (unless trapped by a `⎕TRAP` definition) and the prompt (`⎕:`) is again displayed for input. An EOF interrupt reports `INPUT INTERRUPT` and the prompt (`⎕:`) is again displayed for input. A soft interrupt is ignored and a hard interrupt reports `INTERRUPT` and the prompt (`⎕:`) is redisplayed for input.

Examples

```
      10×⎕+2
⎕:
      ⍳3
30 40 50

      2+⎕
⎕:
      X
VALUE ERROR
      X
     ^
⎕:
      2+⍳3
5 6 7
```

A system command may be entered. The system command is effected and the prompt is displayed again (unless the system command changes the environment):

```
      ⍴3,⎕
⎕:
      )WSID
WS/MYWORK
⎕:
      )SI
⎕
⎕:
      )CLEAR
CLEAR WS
```

If the response to a `⎕:` prompt is an abort statement (`→`), the execution will be aborted:

```
      1 2 3 = ⎕
⎕:
      →
```

A trap definition on interrupt events set for the system variable `⎕TRAP` in the range 1000-1006 has no effect whilst awaiting input in response to a `⎕:` prompt.

Example

```
      ⎕TRAP←(11 'C' '''ERROR''')(1000 'C' '''STOP''')

      2+⎕
⎕:
      (Interrupt Signal)
INTERRUPT
⎕:
      'C'+2
ERROR
```

A time limit set in system variable `⎕RTL` has no effect whilst awaiting input in response to a `⎕:` prompt.

Underscored Alphabetic Characters: R←⎕A̲

⎕A̲ is a deprecated feature. Dyalog **strongly** recommends that you move away from the use of ⎕A̲ and of the underscored alphabet itself, as these symbols now constitute the sole remaining non-standard use of characters in Dyalog applications.

In Versions of Dyalog APL prior to Version 11.0, ⎕A̲ was a simple character vector, composed of the letters of the alphabet with underscores. If the Dyalog Alt font was in use, these symbols displayed as additional National Language characters.

Version 10.1 and Earlier

```
      ⎕A̲
A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲
```

For compatibility with previous versions of Dyalog APL, functions that contain references to ⎕A̲ will continue to return characters with the same *index* in ⎕AV as before. However, the display of ⎕A̲ is now ⎕Á, and the old underscored symbols appear as they did in previous Versions when the Dyalog Alt font was in use.

Current Version

```
      ⎕Á
ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝþãìðòõ
```

Alphabetic Characters: R←⎕A

This is a simple character vector, composed of the letters of the alphabet.

Example

```
      ⎕A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Account Information: R←⎕AI

This is a simple integer vector, whose four elements are:

`⎕AI[1]` - user identification.
Under Windows, this is the `aplnid` (network ID from configuration dialog box).Under UNIX and LINUX, this is the UID of the account.

`⎕AI[2]` - compute time for the APL session in milliseconds.

`⎕AI[3]` - connect time for the APL session in milliseconds.

`⎕AI[4]` - keying time for the APL session in milliseconds.

Elements beyond 4 are not defined but reserved.

Example

```
      ⎕AI
52 7396 2924216 2814831
```

Account Name: R←⎕AN

This is a simple character vector containing the user (login) name.

Example

```
      ⎕AN
Pete

      ⍴⎕AN
4
```

Arbitrary Input: R←{X}⎕ARBIN Y

This transmits the prompt Y to an output device specified by X prior to reading from an input device specified by X.

Under Windows, the use of ⎕ARBIN to the screen or in conjunction with RS232 ports is not supported.

Y may be a scalar, a simple vector, or a vector of simple scalars or vectors. The items of the simple arrays of Y must each be a character, or a number in the range 0 to 255. Numbers are sent to the output device without translation. Characters undergo the standard ⎕AV to ASCII translation. If Y is an empty vector, no codes are sent to the output device.

X may take several forms:

```
terminate (input output) ⎕ARBIN  prompt
terminate input          ⎕ARBIN  prompt
terminate                ⎕ARBIN  prompt
                         ⎕ARBIN  prompt
```

Each of these elements is discussed separately.

<terminate>

This defines how the read should be terminated.

If it is omitted, the read terminates on receipt of a Newline character.

If supplied, it must be a simple numeric scalar or vector.

- If it is a numeric scalar, it defines the number of characters to be read.
- If it is a numeric vector, it defines a set of terminating characters.
- If it is the null vector, the read terminates on Newline.

<input>

This defines the input device.

If this is omitted, input is taken from standard input (usually the keyboard).

If supplied, it must be a simple numeric scalar or a simple text vector.

- If it is a numeric scalar, it must correspond to a DOS handle or UNIX stream number.
- If it is a text vector, it must correspond to a valid device or file name.

You must have permission to read from the chosen device.

<output>

This defines the output device.

If this is omitted, output is sent to standard output (usually the screen).

If supplied, it must be a simple numeric scalar or a simple text vector.

- If it is a numeric scalar, it must correspond to a DOS handle or UNIX stream number.
- If it is a text vector, it must correspond to a valid device or file name.

You must have permission to write to the chosen device.

The result `R` is a simple numeric vector. Each item of `R` is the numeric representation of an 8-bit code in the range 0 to 255 received from the input device. The meaning of the code is dependent on the characteristics of the input device. If a set of delimiters was defined by <terminate>, the last code returned will belong to that set.

`⎕RTL` (Response Time Limit) is an implicit argument of `⎕ARBIN`. This allows a time limit to be imposed on input. If the time limit is reached, `⎕ARBIN` returns with the codes read up to that point.

Examples

Write `HELLO` on the screen, and read a line of input
from the keyboard:

```
      R ← ⎕ARBIN 'HELLO'
```

Beep three times, send `ARE YOU AWAKE?` to the screen and wait for a 1 character answer from the keyboard:

```
      R ← 1 ⎕ARBIN 7 7 7 'ARE YOU AWAKE (Y/N)'
```

Read a line from `MYFILE`:

```
      R ← '' 'MYFILE' ⎕ARBIN ''
```

Read `MYFILE` until a SPACE (code 32):

```
      R ← (,32) 'MYFILE' ⎕ARBIN ''
```

Read `MYFILE` until a SPACE (code 32) or a TAB (code 9):

```
      R ← (32 9) 'MYFILE' ⎕ARBIN ''
```

Write `HELLO` on /dev/tty1 (a UNIX terminal screen), then read a line from /dev/tty1 (a UNIX terminal keyboard):

```
      R ← '' ('/dev/tty1' '/dev/tty1') ⎕ARBIN 'HELLO'
```

Write `TITLE` to LPT1 (a DOS printer device), then read from COM1 (a DOS serial port) up to the first NEWLINE character:

```
      R ← (,13) ('COM1' 'LPT1') ⎕ARBIN 'TITLE'
```

Read 100 characters from COM1; timeout after 10 secs:

```
      ⎕RTL←10
      R←100 'COM1' ⎕ARBIN ''
      ⍎(100≠⍴R)/'''Read timed out'''
```

Read until DELIM from COM1; timeout after 10 secs:

```
      ⎕RTL←10
      R←(,DELIM) 'COM1' ⎕ARBIN ''
      ⍎(DELIM≠¯1↑R)/'''Read timed out'''
```

Arbitrary Output: {X}⎕ARBOUT Y

This transmits Y to an output device specified by X.

Under Windows, the use of ⎕ARBOUT to the screen or to RS232 ports is not supported.

Y may be a scalar, a simple vector, or a vector of simple scalars or vectors. The items of the simple arrays of Y must each be a character or a number in the range 0 to 255. Numbers are sent to the output device without translation. Characters undergo the standard ⎕AV to ASCII translation. If Y is an empty vector, no codes are sent to the output device.

X defines the output device. If X is omitted, output is sent to standard output (usually the screen). If X is supplied, it must be a simple numeric scalar or a simple text vector.

If it is a numeric scalar, it must correspond to a DOS handle or UNIX stream number.

If it is a text vector, it must correspond to a valid device or file name.

You must have permission to write to the chosen device.

Examples

Write ASCII digits '123' to UNIX stream 9:

```
      9 ⎕ARBOUT 49 50 51
```

Write ASCII characters 'ABC' to MYFILE:

```
      'MYFILE' ⎕ARBOUT 'ABC'
```

Beep 3 times:

```
      ⎕ARBOUT 7 7 7
```

Prompt for input:

```
      ⍞← 'Prompt: ' ⋄ ⎕arbout 12 ⋄ ans←⍞
```

Attributes: R←{X}⎕AT Y

Y can be a simple character scalar, vector or matrix, or a vector of character vectors representing the names of 0 or more defined functions or operators. Used dyadically, this function closely emulates the APL2 implementation. Used monadically, it returns information that is more appropriate for Dyalog APL.

Monadic Use

If X is omitted, R is a 4 column matrix with the same number of rows as functions in Y containing the following attribute information:

R[;1] Each item is a 3-element integer vector representing the function header syntax:

Item[1] result:

0	- No result
1	- Explicit result
¯1	- Shy result

Item[2] Function valence:

0	- Niladic function
1	- Monadic function
2	- Dyadic function
¯2	- Ambivalent function

Item[3] Operator valence

0	- Not an operator
1	- Monadic operator
2	- Dyadic operator

The following values correspond to the syntax shown alongside:

```
 0  0  0      ∇ FOO
 1  0  0      ∇ Z←FOO
¯1  0  0      ∇ {Z}←FOO
 0 ¯2  0      ∇ {A} FOO B
¯1  1  2      ∇ {Z}←(F OP G)B
```

`R[;2]`	Each item is the (`⎕TS` form) timestamp of the time the function was last fixed.
`R[;3]`	Each item is an integer reporting the current `⎕LOCK` state of the function: `0` - Not locked `1` - Cannot display function `2` - Cannot suspend function `3` - Cannot display or suspend.
`R[;4]`	Each item is a character vector - the network ID of the user who last fixed (edited) the function.

Example

```
    ∇ {z}←{l}(fn myop)r
[1]   ...

    ∇ z←foo
[1]   ...

    ∇ z←{larg}util rarg
[1]   ...

      ⎕LOCK'foo'

      util2←util

      DISPLAY ⎕AT 'myop' 'foo' 'util' 'util2'
.→--------------------------------------------.
↓ .→------. .→-----------------.     .→---.   |
| |¯1 ¯2 1| |1996 8 2 2 13 56 0|   0 |john|   |
| '~------' '~-----------------'     '----'   |
| .→----.   .→------------.          .⊖.      |
| |1 0 0|   |0 0 0 0 0 0 0|        3 | |      |
| '~----'   '~------------'          '-'      |
| .→-----.  .→------------------.    .→---.   |
| |1 ¯2 0|  |1996 3 1 14 12 10 0|  0 |pete|   |
| '~-----'  '~------------------'    '----'   |
| .→-----.  .→-------------------.   .→-----. |
| |1 ¯2 0|  |1998 8 26 16 16 42 0| 0 |graeme| |
| '~-----'  '~-------------------'   '------' |
'∊--------------------------------------------'
```

Dyadic Use

The dyadic form of ⎕AT emulates APL2. It returns the same rank and shape result containing information that matches the APL2 implementation as closely as possible.

If Y specifies a single name, the result R is a vector. If Y specifies more than one name, R is a matrix with one row per name in Y. The number of elements (columns) and their meaning depends upon the value of X which may be 1, 2, 3 or 4.

If X is 1, R specifies *valences* and contains 3 elements (or columns) whose meaning is as follows:

1	Explicit result	1 if the object has an explicit result or is a variable; 0 otherwise
2	Function valence	0 if the object is a niladic function or not a function 1 if the object is a monadic function 2 if the object is an ambivalent function
3	Operator valence	0 if the object is not an operator 1 if the object is a monadic operator 2 if the object is a dyadic operator

If X is 2, R specifies *fix times* (the time the object was last updated) for functions and operators named in Y. The time is reported as 7 integer elements (or columns) whose meaning is as follows. The fix time reported for names in Y which are not defined functions or operators is 0.

1	Year
2	Month
3	Day
4	Hour
5	Minute
6	Second
7	Milliseconds (this is always reported as 0)

If `X` is 3, `R` specifies *execution properties* and contains 4 elements (or columns) whose meaning is as follows:

1	Displayable	0 if the object is displayable 1 if the object is not displayable
2	Suspendable	0 if execution will suspend in the object 1 if execution will not suspend in the object
3	Weak Interrupt behaviour	0 if the object responds to interrupt 1 if the object ignores interrupt
4		(always 0)

Note that the execution properties for primitive and system functions are `0 1 1 0`.

If `X` is 4, `R` specifies *object size* and contains 2 elements (or columns) which both report the `⎕SIZE` of the object.

Atomic Vector: R←⎕AV

⎕AV is a deprecated feature and is replaced by ⎕UCS.

This is a simple character vector of all 256 characters in the Classic Dyalog APL character.

In the Classic Edition the contents of ⎕AV are defined by the Output Translate Table.

In the Unicode Edition, the contents of ⎕AV are defined by the system variable ⎕AVU.

Examples

```
      ⎕AV[48+⍳10]
0123456789

      5 52⍴12↓⎕av
%'αω_abcdefghijklmnopqrstuvwxyz__¯.⍬0123456789_¤¥$£¢
∆ABCDEFGHIJKLMNOPQRSTUVWXYZ__ý·⌂⍙ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛ
ÝÞãìðòõ{€}⊣⌷¨ÀÄÅÆ⍨ÉÑÖØÜßàáâäåæçèéêëíîïñ[/⌿\⍀<≤=≥>≠∨∧
-+÷×?∊⍴~↑↓⍳○*⌈⌊∇∘(⊂⊃∩∪⊥⊤|;,⍱⍲⍒⍋⍉⌽⊖⍟⌹!⍕⍎⍫⍪≡≢óôöø"#_&'
___________@ùúû^ü`□¶:⍷¿¡⋄←→⍝)]⌂⌂§⎕⍞⍣%'αω_abcdefghijk
```

Atomic Vector - Unicode: ⎕AVU

⎕AVU specifies the contents of the atomic vector, ⎕AV, and is used to translate data between Unicode and non-Unicode character formats when required, for example when:

- Unicode Edition loads or copies a Classic Edition workspace or a workspace saved by a Version prior to Version 12.0.
- Unicode Edition reads character data from a non-Unicode component file, or receives data type 82 from a TCP socket.
- Unicode Edition writes data to a non-Unicode component file
- Unicode Edition reads or writes data from or to a Native File using conversion code 82.

- Classic Edition loads or copies a Unicode Edition workspace
- Classic Edition reads character data from a Unicode component file, or receives data type 80, 160, or 320 from a TCP socket.
- Classic Edition writes data to a Unicode component file.

⎕AVU is an integer vector with 256 elements, containing the Unicode code points which define the characters in ⎕AV.

Note

In Versions of Dyalog prior to Version 12.0 and in the Classic Edition, a character is stored internally as an index into the atomic vector, ⎕AV. When a character is displayed or printed, the index in ⎕AV is translated to a number in the range 0-255 which represents the index of the character in an Extended ASCII font. This mapping is done by the Output Translate Table which is user-configurable. Note that although ASCII fonts typically all contain the same symbols in the range 0-127, there are a number of different Extended ASCII font layouts, including proprietary APL fonts, which provide different symbols in positions 128-255. The actual symbol that appears on the screen or on the printed page is therefore a function of the Output Translate Table and the font in use. Classic Edition provides two different fonts (and thus two different ⎕AV layouts) for use with the Development Environment, named *Dyalog Std* (with APL underscores) and *Dyalog Alt* (without APL underscores

The default value of ⎕AVU corresponds to the use of the **Dyalog Alt** Output Translate Table and font in the Classic Edition or in earlier versions of Dyalog APL.

```
      2 13⍴⎕AVU[97+⍳26]
193 194 195 199 200 202 203 204 205 206 207 208 210
211 212 213 217 218 219 221 254 227 236 240 242 245
      ⎕UCS 2 13⍴⎕AVU[97+⍳26]
ÁÂÃÇÈÊËÌÍÎÏÐÒ
ÓÔÕÙÚÛÝþãìðòõ
```

⎕AVU has namespace scope and can be localised, in order to make it straightforward to write access functions which receive or read data from systems with varying atomic vectors. If you have been using Dyalog Alt for most things but have some older code which uses underscores, you can bring this code together in the same workspace and have it all look “as it should” by using the Alt and Std definitions for ⎕AVU as you copy each part of the code into the same Unicode Edition workspace.

```
      )COPY avu.dws Std.⎕AVU
C:\Program Files\Dyalog\Dyalog APL 12.0 Unicode\ws\avu
saved Thu Dec 06 11:24:32 2007

      2 13⍴⎕AVU[97+⍳26]
9398 9399 9400 9401 9402 9403 9404 9405 9406 9407 9408
9409 9410
9411 9412 9413 9414 9415 9416 9417 9418 9419 9420 9421
9422 9423
       ⎕UCS 2 13⍴⎕AVU[97+⍳26]
ⒶⒷⒸⒹⒺⒻⒼⒽⒾⒿⓀⓁⓂ
ⓃⓄⓅⓆⓇⓈⓉⓊⓋⓌⓍⓎⓏ
```

Rules for Conversion on Import

When the Unicode Edition imports APL objects from a non-Unicode source, function comments and character data of type 82 are converted to Unicode. When the Classic Edition imports APL objects from a Unicode source, this translation is performed in reverse.

If the objects are imported from a Version 12.0 (or later) workspace (i.e. from a workspace that contains its own value of `⎕AVU`) the value of `#.⎕AVU` (the value of `⎕AVU` in the root) in the *source* workspace is used. Otherwise, such as when APL objects are imported from a pre-Version 12 workspace, from a component file, or from a TCP socket, the local value of `⎕AVU` in the *target* workspace is used.

Rules for Conversion on Export

When the Unicode Edition exports APL objects to a non-Unicode destination, such as a non-Unicode Component File or non-Unicode TCPSocket Object, function comments (in `⎕OR`s) and character data of type 82 are converted to `⎕AV` indices using the local value of `⎕AVU`.

When the Classic Edition exports APL objects to a Unicode destination, such as a Unicode Component File or Unicode TCPSocket Object, function comments (in `⎕OR`s) and character data of type 82 are converted to Unicode using the local value of `⎕AVU`.

In all cases, if a character to be translated is not defined in `⎕AVU`, a TRANSLATION ERROR (event number 92) will be signalled.

Base Class: R←⎕BASE.Y

`⎕BASE` is used to access the base class implementation of the name specified by `Y`.

`Y` must be the name of a Public member (Method, Field or Property) that is provided by the Base Class of the current Class or Instance.

`⎕BASE` is typically used to call a method in the Base Class which has been *superseded* by a Method in the current Class.

Note that `⎕BASE.Y` is *special syntax* and any direct reference to `⎕BASE` on its own or in any other context, is meaningless and causes `SYNTAX ERROR`.

In the following example, Class `DomesticParrot` derives from Class `Parrot` and supersedes its `Speak` method. `DomesticParrot.Speak` calls the `Speak` method in its Base Class `Parrot`, via `⎕BASE`.

```
:Class Parrot: Bird
    ∇ R←Speak
      :Access Public
      R←'Squark!'
    ∇
:EndClass ⍝ Parrot

:Class DomesticParrot: Parrot
    ∇ R←Speak
      :Access Public
      R←⎕BASE.Speak,' Who's a pretty boy,then!'
    ∇
:EndClass ⍝ DomesticParrot

      Maccaw←⎕NEW Parrot
      Maccaw.Speak
Squark!

      Polly←⎕NEW DomesticParrot
      Polly.Speak
Squark! Who's a pretty boy,then!
```

Class: R←{X}⎕CLASS Y

Monadic Case

Monadic `⎕CLASS` returns a list of references to Classes and Interfaces that specifies the class hierarchy for the Class or Instance specified by `Y`.

`Y` must be a reference to a Class or to an Instance of a Class.

`R` is a vector or vectors whose items represent nodes in the Class hierarchy of `Y`. Each item of `R` is a vector whose first item is a Class reference and whose subsequent items (if any) are references to the Interfaces supported by that Class.

Example 1

This example illustrates a simple inheritance tree or Class hierarchy. There are 3 Classes, namely:

```
Animal
      Bird (derived from Animal)
            Parrot (derived from Bird)
```

```
:Class Animal
...
:EndClass ⍝ Animal

:Class Bird: Animal
...
:EndClass ⍝ Bird

:Class Parrot: Bird
...
:EndClass ⍝ Parrot

      ⎕CLASS Eeyore←⎕NEW Animal
 #.Animal
      ⎕CLASS Robin←⎕NEW Bird
 #.Bird   #.Animal
      ⎕CLASS Polly←⎕NEW Parrot
 #.Parrot   #.Bird   #.Animal

      ⎕CLASS¨ Parrot Animal
  #.Parrot   #.Bird   #.Animal      #.Animal
```

Example 2

The Penguin Class Class example (see page 164) illustrates the use of Interfaces. In this case, the `Penguin` Class derives from `Animal` (as above) but additionally supports the `BirdBehaviour` and `FishBehaviour` Interfaces, thereby inheriting members from both.

```
      Pingo←⎕NEW Penguin
      ⎕CLASS Pingo
 #.Penguin  #.FishBehaviour  #.BirdBehaviour   #.Animal
```

Dyadic Case

If `X` is specified, `Y` must be a reference to an Instance of a Class and `X` is a reference to an Interface that is supported by Instance `Y` or to a Class upon which Instance `Y` is based.

In this case, `R` is a reference to the implementation of Interface `X` by Instance `Y`, or to the implementation of (Base) Class `X` by Instance `Y`,and is used as a *cast* in order to access members of `Y` that correspond to members of Interface of (Base) Class `X`.

Example 1:

Once again, the Penguin Class example (see page 164) is used to illustrate the use of Interfaces.

```
      Pingo←⎕NEW Penguin
      ⎕CLASS Pingo
  #.Penguin  #.FishBehaviour  #.BirdBehaviour    #.Animal

      (FishBehaviour ⎕CLASS Pingo).Swim
I can dive and swim like a fish
      (BirdBehaviour ⎕CLASS Pingo).Fly
Although I am a bird, I cannot fly
      (BirdBehaviour ⎕CLASS Pingo).Lay
I lay one egg every year
      (BirdBehaviour ⎕CLASS Pingo).Sing
Croak, Croak!
```

Example 2:

This example illustrates the use of dyadic `⎕CLASS` to cast an Instance to a lower Class and thereby access a member in the lower Class that has been superseded by another Class higher in the tree.

```
      Polly←⎕NEW DomesticParrot
      Polly.Speak
Squark! Who's a pretty boy,then!
```

Note that the `Speak` method invoked above is the `Speak` method defined by Class `DomesticParrot`, which supersedes the `Speak` methods of sub-classes `Parrot` and `Bird`.

You may use a cast to access the (superseded) `Speak` method in the sub-classes `Parrot` and `Bird`.

```
      (Parrot ⎕CLASS Polly).Speak
Squark!
      (Bird ⎕CLASS Polly).Speak
Tweet, tweet!
```

Clear Workspace: ⎕CLEAR

A clear workspace is activated, having the name `CLEAR WS`. The active workspace is lost. All system variables assume their default values. The maximum size of workspace is available.

The contents of the session namespace `⎕SE` are not affected.

Example

```
      ⎕CLEAR
      ⎕WSID
CLEAR WS
```

Execute Windows Command: R←⎕CMD Y

`⎕CMD` executes a Windows Command Processor or UNIX shell or starts another Windows application program. `⎕CMD` is a synonym of `⎕SH`. Either system function may be used in either environment (Windows or UNIX) with exactly the same effect. `⎕CMD` is probably more natural for the Windows user. This section describes the behaviour of `⎕CMD` and `⎕SH` under Windows. See `⎕SH` for a discussion of the behaviour of these system functions under UNIX.

The system commands `)CMD` and `)SH` provide similar facilities but may only be executed from the APL Session.

Executing a Windows Command

If `Y` is a simple character vector, `⎕CMD` invokes the Windows Command Processor (normally `cmd.exe`) and passes `Y` to it for execution. `R` is a vector of character vectors containing the result of the command. Each element in `R` corresponds to a line of output produced by the command.

Example

```
      Z←⎕CMD'DIR'
      ⍴Z
8
```

```
      ↑Z

 Volume in drive C has no label
 Directory of C:\DYALOG

.            <DIR>     5-07-89  3.02p
..           <DIR>     5-07-89  3.02p
SALES    DWS    110092 5-07-89  3.29p
EXPENSES DWS    154207 5-07-89  3.29p
```

If the command specified in `Y` already contains the redirection symbol (`>`) the capture of output through a pipe is avoided and the result `R` is empty. If the command specified by `Y` issues prompts and expects user input, it is **ESSENTIAL** to explicitly redirect input and output to the console. If this is done, APL detects the presence of a "`>`" in the command line, runs the command processor in a **visible** window, and does not direct output to the pipe. If you fail to do this your system will appear to hang because there is no mechanism for you to receive or respond to the prompt.

Example

```
      ⎕CMD 'DATE <CON >CON'
```
(Command Prompt window appears)
```
Current date is Wed 19-07-1995
Enter new date (dd-mm-yy): 20-07-95
```
(COMMAND PROMPT window disappears)

Implementation Notes

The right argument of `⎕CMD` is simply passed to the appropriate command processor for execution and its output is received using an *unnamed pipe*.

By default, `⎕CMD` will execute the string `('cmd.exe /c',Y)`; where `Y` is the argument given to `⎕CMD`. However, the implementation permits the use of alternative command processors as follows.

Before execution, the argument is prefixed and postfixed with strings defined by the APL parameters CMD_PREFIX and CMD_POSTFIX. The former specifies the name of your command processor and any parameters that it requires. The latter specifies a string which may be required. If CMD_PREFIX is not defined, it defaults to the name defined by the environment variable COMSPEC followed by "/c". If COMSPEC is not defined, it defaults to `cmd.exe`. If CMD_POSTFIX is not defined, it defaults to an empty vector.

⎕CMD treats certain characters as having special meaning as follows:

#	marks the start of a trailing comment,
;	divides the command into sub-commands,
>	if found within the last sub-command, causes ⎕CMD to use a visible window.

If you simply wish to open a Command Prompt window, you may execute the command as a Windows Program (see below). For example:

```
      ⎕CMD 'cmd.exe' ''
```

Executing a Windows Program

If Y is a 2-element vector of character vectors, ⎕CMD starts the executable program named by Y[1] with the initial window parameter specified by Y[2]. The shy result is an integer scalar containing the window handle allocated by the window manager.

Y[1] must specify the name or complete pathname of an executable program. If the name alone is specified, Windows will search the following directories :

1. the current directory,
2. the Windows directory,
3. the Windows system directory,
4. the directories specified by the PATH variable,
5. the list of directories mapped in a network.

Note that Y[1] may contain the complete command line, including any suitable parameters for starting the program. If Windows fails to find the executable program, ⎕CMD will fail and report FILE ERROR 2.

Y[2] specifies the window parameter and may be one of the following. If not, a DOMAIN ERROR is reported.

'Normal' ''	Application is started in a normal window, which is given the input focus
'Unfocused'	Application is started in a normal window, which is NOT given the input focus
'Hidden'	Application is run in an invisible window
'Minimized' 'Minimised'	Application is started as an icon which is NOT given the input focu
'Maximized' 'Maximised'	Application is started maximized (full screen) and is given the input focus

An application started by ⎕CMD may ONLY be terminated by itself or by the user. There is no way to close it from APL. Furthermore, if the window parameter is HIDDEN, the user is unaware of the application (unless it makes itself visible) and has no means to close it.

Examples

```
      Path←'c:\Program Files\Microsoft Office\Office\'
      ⎕←⎕CMD (Path,'excel.exe') ''
33
      ⎕CMD (Path,'winword /mMyMacro') 'Minimized'
```

Start Windows Auxiliary Processor: X ⎕CMD Y

Used dyadically, ⎕CMD starts an Auxiliary Processor. The effect, as far as the APL workspace is concerned, is identical under both Windows and UNIX, although the method of implementation differs. ⎕CMD is a synonym of ⎕SH. Either function may be used in either environment (Windows or UNIX) with exactly the same effect. ⎕CMD is probably more natural for the Windows user. This section describes the behaviour of ⎕CMD and ⎕SH under Windows. See ⎕SH for a discussion of the behaviour of these system functions under UNIX.

X must be a simple character vector containing the name (or pathname) of a Dyalog APL Auxiliary Processor (AP). See *User Guide* for details of how to write an AP.

Y may be a simple character scalar or vector, or a vector of character vectors. Under Windows the contents of Y are ignored.

⎕CMD loads the Auxiliary Processor into memory. If no other APs are currently running, ⎕CMD also allocates an area of memory for communication between APL and its APs.

The effect of starting an AP is that one or more **external functions** are defined in the workspace. These appear as locked functions and may be used in exactly the same way as regular defined functions.

When an external function is used in an expression, the argument(s) (if any) are passed to the AP for processing via the communications area described above. APL halts whilst the AP is processing, and waits for a result. Under Windows, unlike under UNIX, it is not possible for external functions to run in parallel with APL.

Canonical Representation: R←⎕CR Y

Y must be a simple character scalar or vector which represents the name of a defined function or operator.

If Y is a name of a defined function or operator, R is a simple character matrix. The first row of R is the function or operator header. Subsequent rows are lines of the function or operator. R contains no unnecessary blanks, except for leading indentation of control structures, trailing blanks that pad each row, and the blanks in comments. If Y is the name of a variable, a locked function or operator, an external function, or is undefined, R is an empty matrix whose shape is 0 0.

Example

```
      ∇R←MEAN X    ⍝ Arithmetic mean
[1]  R←(+/X)÷⍴X
[2]  ∇
      +F←⎕CR'MEAN'
R←MEAN X    ⍝ Arithmetic mean
R←(+/X)÷⍴X

      ⍴F
2 30
```

The definition of ⎕CR has been extended to names assigned to functions by specification (←), and to local names of functions used as operands to defined operators.

If Y is a name assigned to a primitive function, R is a one-element vector containing the corresponding function symbol. If Y is a name assigned to a system function, R is a one element nested array containing the name of the system function.

Examples

```
      PLUS←+
      +F←⎕CR'PLUS'
+
      ⍴F
1
      C←⎕CR
      C'C'
 ⎕CR
      ⍴C'C'
1
```

```
      ∇R←CONDITION (FN1 ELSE FN2) X
[1]    →CONDITION/L1
[2]    R←FN2 X ⋄ →0
[3]  L1:R←FN1 X
[4]    ∇

      2 ⎕STOP 'ELSE'
      (X≥0) ⌊ ELSE ⌈ X←¯2.5

ELSE[2]
      X
¯2.5
      ⎕CR'FN2'
⌈
      →⎕LC
¯2
```

If Y is a name assigned to a derived function, R is a vector whose elements represent the arrays, functions, and operators from which Y was constructed. Constituent functions are represented by their own ⎕CRs, so in this respect the definition of ⎕CR is recursive. Primitive operators are treated like primitive functions, and are represented by their corresponding symbols. Arrays are represented by themselves.

Example

```
      BOX←2 2∘⍴
      +F←⎕CR'BOX'
 2 2 ∘⍴
      ⍴F
3
      DISPLAY F
.→----------.
| .→--.     |
| |2 2| ∘ ⍴ |
| '~--' - - |
'∊----------'
```

If Y is a name assigned to a defined function, R is the ⎕CR of the defined function. In particular, the name that appears in the function header is the name of the original defined function, not the assigned name Y.

Example

```
      AVERAGE←MEAN
      ⎕CR'AVERAGE'
R←MEAN X    ⍝ Arithmetic mean
R←(+/X)÷⍴X
```

Change Space: {R}←{X}⎕CS Y

`Y` must be namespace reference (ref) or a simple character scalar or vector identifying the name of a namespace.

If specified, `X` is a simple character scalar, vector, matrix or a nested vector of character vectors identifying zero or more workspace objects to be *exported* into the namespace `Y`.

The identifiers in `X` and `Y` may be simple names or compound names separated by `'.'` and including the names of the special namespaces `'⎕SE'`, `'#'`, `'##'` and `''`.

The result `R` is the full name (starting "`#.`") of the space in which the function or operator was executing prior to the `⎕CS`.

`⎕CS` changes the space in which the current function or operator is running to the namespace `Y` and returns the original space, in which the function was previously running, as a shy result. **After the** `⎕CS`, references to *global* names (with the exception of those specified in `X`) are taken to be references to *global* names in `Y`. References to local names (i.e. those local to the current function or operator) are unaffected.

When the function or operator terminates, the calling function resumes execution in its original space.

The names listed in `X` are temporarily *exported* to the namespace `Y`. If objects with the same name exist in `Y` , these objects are effectively *shadowed* and are inaccessible.

Note that calling `⎕CS` with an empty argument `Y` obtains the namespace in which a function is currently executing.

Example

This simple example illustrates how `⎕CS` may be used to avoid typing long pathnames when building a tree of GUI objects. Note that the objects `NEW` and `OPEN` are created as children of the `FILE` menu as a result of using `⎕CS` to change into the `F.MB.FILE` namespace.

```
     ∇ MAKE_FORM;F;OLD
[1]    'F'⎕WC'Form'
[2]    'F.MB'⎕WC'MenuBar'
[3]    'F.MB.FILE'⎕WC'Menu' '&File'
[4]
[5]    OLD←⎕CS'F.MB.FILE'
[6]    'NEW'⎕WC'MenuItem' '&New'
[7]    'OPEN'⎕WC'MenuItem' '&Open'
[8]    ⎕CS OLD
[9]
[10]   'F.MB.EDIT'⎕WC'Menu' '&Edit'
[11]
[12]   OLD←⎕CS'F.MB.EDIT'
[13]   'UNDO'⎕WC'MenuItem' '&Undo'
[14]   'REDO'⎕WC'MenuItem' '&Redo'
[15]   ⎕CS OLD
[16]   ...
     ∇
```

Example

Suppose a form `F1` contains buttons `B1` and `B2`. Each button maintains a count of the number of times it has been pressed, and the form maintains a count of the total number of button presses. The single callback function `PRESS` and its subfunction `FMT` can reside in the form itself

```
      )CS F1
#.F1
      ⍝ Note that both instances reference
      ⍝ the same callback function
      'B1'⎕WS'Event' 'Select' 'PRESS'
      'B2'⎕WS'Event' 'Select' 'PRESS'

      ⍝ Initialise total and instance counts.
      TOTAL ← B1.COUNT ← B2.COUNT ← 0

    ∇ PRESS MSG
[1]   'FMT TOTAL'⎕CS⊃MSG ⍝       Switch to instance space
[2]   (TOTAL COUNT)+←1   ⍝    Incr total & instance count
[3]   ⎕WS'Caption'(COUNT FMT TOTAL)⍝ Set instance caption
    ∇

    ∇ CAPT←INST FMT TOTL      ⍝ Format button caption.
[1]   CAPT←(⍕INST),'/',⍕TOTL  ⍝ E.g. 40/100.
    ∇
```

Example

This example uses ⎕CS to explore a namespace tree and display the structure. Note that it must export its own name (tree) each time it changes space, because the name tree is global.

```
      ∇ tabs tree space;subs     ⍝ Display namespace tree
[1]    tabs,space
[2]    'tree'⎕CS space
[3]    →(⍴subs←↓⎕NL 9)↓0
[4]    (tabs,'.   ')∘tree¨subs
     ∇

      )ns x.y
#.x.y
      )ns z
#.z
      ''tree '#'
#
.   x
.   .   y
.   z
```

Comparison Tolerance: ⎕CT

The value of ⎕CT determines the precision with which two numbers are judged to be equal. Two numbers, X and Y, are judged to be equal if:

      (|X-Y)≤⎕CT×(|X)⌈|Y   where ≤ is applied without tolerance.

⎕CT may be assigned any value in the range from 0 to 16*¯8. A value of 0 ensures exact comparison. The value in a clear workspace is 1E¯14.

⎕CT is an implicit argument of the monadic primitive functions Ceiling (⌈), Floor (⌊) and Unique (∪), and of the dyadic functions Equal (=), Excluding (~), Find (⍷), Greater (>), Greater or Equal (≥), Index of (⍳), Intersection (∩), Less (<), Less or Equal (≤), Match (≡), Membership (∊), Not Match (≢), Not Equal (≠), Residue (|) and Union (∪), as well as ⎕FMT O-format.

Examples

```
      ⎕CT←1E¯10
      1.00000000001 1.0000001 = 1
1 0
```

Copy Workspace: {X}⎕CY Y

Y must be a simple character scalar or vector identifying a saved workspace. X is optional. If present, it must be a simple character scalar, vector or matrix. A scalar or vector is treated as a single row matrix. Each (implied) row of X is interpreted as an APL name.

Each (implied) row of X is taken to be the name of an active object in the workspace identified by Y. If X is omitted, the names of all defined active objects in that workspace are implied (defined functions and operators, variables, labels and namespaces).

Each object named in X (or implied) is copied from the workspace identified by Y to become the active object referenced by that name in the active workspace if the object can be copied. A copied label is re-defined to be a variable of numeric type. If the name of the copied object has an active referent in the active workspace, the name is disassociated from its value and the copied object becomes the active referent to that name. In particular, a function in the state indicator which is disassociated may be executed whilst it remains in the state indicator, but it ceases to exist for other purposes, such as editing.

You may copy an object from a namespace by specifying its full pathname. The object will be copied to the current namespace in the active workspace, losing its original parent and gaining a new one in the process. You may only copy a GUI object into a namespace that is a suitable parent for that object. For example, you could only copy a Group object from a saved workspace if the current namespace in the active workspace is itself a Form, SubForm or Group.

See)COPY for further information and, in particular, the manner in which dependant objects are copied.

A DOMAIN ERROR is reported in any of the following cases:

1. Y is ill-formed, or is not the name of a workspace with access authorised for the active user account.
2. Any name in X is ill-formed.
3. An object named in X does not exist as an active object in workspace named in Y.
4. An object being copied has the same name as an active label.

When copying data between Classic and Unicode Editions, ⎕CY will fail and a TRANSLATION ERROR will be reported if *any* object in workspace Y fails conversion between Unicode and ⎕AV indices, whether or not that object is specified by X. See ⎕AVU for further details.

A `WS FULL` is reported if the active workspace becomes full during the copying process.

Example

```
      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇
      'FOO' ⎕CY 'BACKUP'
      ⎕VR'FOO'
     ∇ R←FOO X
[1]    R←10×X
     ∇
```

System variables are copied if explicitly included in the left argument, but not if the left argument is omitted.

Example

```
      ⎕LX

      (2 3ρ'⎕LX  X')⎕CY'WS/CRASH'
      ⎕LX
→RESTART
```

A copied object may have the same name as an object being executed. If so, the name is disassociated from the existing object, but the existing object remains defined in the workspace until its execution is completed.

Example

```
      )SI
FOO[1]*

      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇

      'FOO'⎕CY'WS/MYWORK'

      FOO
1 2 3
      )SI
FOO[1]*
      →⎕LC
10
```

Digits: R←⎕D

This is a simple character vector of the digits from 0 to 9.

Example

```
      ⎕D
0123456789
```

Display Form: R←⎕DF Y

⎕DF sets the *Display Form* of a namespace, a GUI object, a Class, or an Instance of a Class.

Y must be a simple character array that specifies the display form of a namespace. If defined, this array will be returned by the *format* functions and ⎕FMT instead of the default for the object in question. This also applies to the string that is displayed when the name is referenced but not assigned (the *default display*).

The result R is the previous value of the Display Form which initially is ⎕NULL.

```
      'F'⎕WC'Form'
      ⍕F
#.F
      ⍴⍕F
3
      ⎕FMT F
#.F
      ⍴⎕FMT F
1 3
      F ⍝ default display uses ⍕
#.F

       F.⎕DF 'Pete's Form'
       ⍕F
Pete's Form
       ⍴⍕F
11
       ⎕FMT F
Pete's Form
       ⍴⎕FMT F
1 11
```

Notice that ⎕DF will accept any character array, but ⎕FMT always returns a matrix.

```
      F.⎕DF 2 2 5⍴⎕A
      F
ABCDE
FGHIJ

KLMNO
PQRST
      ⍴⍕F
2 2 5
      ⍴⎕←⎕FMT F
ABCDE
FGHIJ

KLMNO
PQRST
5 5
```

Note that ⎕DF defines the Display Form statically, rather than dynamically.

```
      'F'⎕WC'Form' 'This is the Caption'
      F
#.F
      F.(⎕DF Caption)⍝ make current caption the display
form
      F
This is the Caption
      F.Caption←'New Caption' ⍝ changing caption does not
change the display form
      F
This is the Caption
```

You may use the Constructor function to assign the Display Form to an Instance of a Class. For example:

```
:Class MyClass
    ∇ Make arg
      :Access Constructor
      :Implements Constructor
      ⎕DF arg
    ∇
:EndClass ⍝ MyClass

      PD←⎕NEW MyClass 'Pete'
      PD
Pete
```

It is possible to set the Display Form for the Root and for ⎕SE

```
      )CLEAR
clear ws
      #
#
      ⎕DF ⎕WSID
      #
CLEAR WS

      ⎕SE
⎕SE
      ⎕SE.⎕DF 'Session'
      ⎕SE
Session
```

Note that ⎕DF applies directly to the object in question and is not automatically applied in a hierarchical fashion.

```
      'X'⎕NS ''
      X
#.X

      'Y'X.⎕NS ''
      X.Y
#.X.Y
      X.⎕DF 'This is X'
      X
This is X

      X.Y
#.X.Y
```

Division Method: ⎕DIV

The value of ⎕DIV determines how division by zero is to be treated. If ⎕DIV=0, division by 0 produces a DOMAIN ERROR except that the special case of 0÷0 returns 1.

If ⎕DIV=1, division by 0 returns 0.

⎕DIV may be assigned the value 0 or 1. The value in a clear workspace is 0.

⎕DIV is an implicit argument of the monadic function Reciprocal (÷) and the dyadic function Divide (÷).

Examples

```
      ⎕DIV←0

      1 0 2 ÷ 2 0 1
0.5 1 2

      ÷0 1
DOMAIN ERROR
      ÷0 1
      ^

      ⎕DIV←1

      ÷0 2
0 0.5

      1 0 2 ÷ 0 0 4
0 0 0.5
```

Delay: {R}←⎕DL Y

Y must be a simple non-negative numeric scalar or one element vector. A pause of approximately Y seconds is caused.

The shy result R is an integer scalar value indicating the length of the pause in seconds.

The pause may be interrupted by a strong interrupt.

Diagnostic Message: R←⎕DM

This niladic function returns the last reported APL error as a three-element vector, giving error message, line in error and position of caret pointer.

Example

```
      2÷0
DOMAIN ERROR
      2÷0
     ^

      ⎕DM
 DOMAIN ERROR         2÷0       ^
```

Dequeue Events: {R}←⎕DQ Y

⎕DQ awaits and processes events. Y specifies the GUI objects(s) for which events are to be processed. Objects are identified by their names, as character scalars/vectors, or by namespace references. These may be objects of type Root, Form, Locator, Filebox, MsgBox, PropertySheet, TCPSocket, Timer, Clipboard and pop-up Menu. Sub-objects (children) of those named in Y are also included. However, any objects which exist, but are not named in Y, are effectively disabled (do not respond to the user).

If Y is '.', all objects currently owned and subsequently created by the current thread are included in the ⎕DQ. Note that because the Root object is owned by thread 0, events on Root are reported only to thread 0.

If Y is empty it specifies the object associated with the current namespace and is only valid if the current space is one of the objects listed above.

Otherwise, Y contains the name(s) of or reference(s) to the objects for which events are to be processed. Effectively, this is the list of objects with which the user may interact. A DOMAIN ERROR is reported if an element of Y refers to anything other than an existing "top-level" object.

Associated with every object is a set of events. For every event there is defined an "action" which specifies how that event is to be processed by ⎕DQ. The "action" may be a number with the value 0, 1 or ¯1, or a character vector containing the name of a "callback function", or a character vector containing the name of a callback function coupled with an arbitrary array. Actions can be defined in a number of ways, but the following examples will illustrate the different cases.

```
      OBJ ⎕WS 'Event' 'Select' 0

      OBJ ⎕WS 'Event' 'Select' 1

      OBJ ⎕WS 'Event' 'Select' 'FOO'

      OBJ ⎕WS 'Event' 'Select' 'FOO' 10

      OBJ ⎕WS 'Event' 'Select' 'FOO&'
```

These are treated as follows :

`Action = 0` (the default)

`⎕DQ` performs "standard" processing appropriate to the object and type of event. For example, the standard processing for a KeyPress event in an Edit object is to action the keypress, i.e. to echo the character on the screen.

`Action = ¯1`

This disables the event. The "standard" processing appropriate to the object and type of event is **not** performed, or in some cases is reversed. For example, if the "action code" for a KeyPress event (22) is set to `¯1`, `⎕DQ` simply ignores all keystrokes for the object in question.

`Action = 1`

`⎕DQ` terminates and returns information pertaining to the event (the **event message** in `R` as a nested vector whose first two elements are the name of the object (that generated the event) and the event code. `R` may contain additional elements depending upon the type of event that occurred.

`Action = fn {larg}`

`fn` is a character vector containing the name of a *callback* function. This function is automatically invoked by `⎕DQ` whenever the event occurs, and **prior** to the standard processing for the event. The callback is supplied the **event message** (see above) as its right argument, and, if specified, the array `larg` as its left argument. If the callback function fails to return a result, or returns the scalar value 1, `⎕DQ` then performs the standard processing appropriate to the object and type of event. If the callback function returns a scalar 0, the standard processing is not performed or in some cases is reversed.

If the callback function returns its event message with some of the parameters changed, these changes are incorporated into the standard processing. An example would be the processing of a keystroke message where the callback function substitutes upper case for lower case characters. The exact nature of this processing is described in the reference section on each event type.

`Action = ⍎expr`

If `Action` is set to a character vector whose first element is the execute symbol (`⍎`) the remaining string will be executed automatically whenever the event occurs. The default processing for the event is performed first and may not be changed or inhibited in any way.

`Action = fn& {larg}`

`fn` is a character vector containing the name of a *callback* function. The function is executed in a new thread. The default processing for the event is performed first and may not be changed or inhibited in any way.

⎕DQ terminates in one of four instances. Note that its result is shy.

Firstly, ⎕DQ terminates when an event occurs whose "action code" is 1. In this case, its result is a nested vector containing the **event message** associated with the event. The structure of an event message varies according to the event type (see *Object Reference*). However, an event message has at least two elements of which the first is a character vector containing the name of the object, and the second is a numeric code specifying the event type.

⎕DQ also terminates if all of the objects named in Y have been deleted. In this case, the result is an empty character vector. Objects are deleted either using ⎕EX, or on exit from a defined function or operator if the names are localised in the header, or on closing a form using the system menu.

Thirdly, ⎕DQ terminates if the object named in its right argument is a special *modal* object, such as a MsgBox, FileBox or Locator, and the user has finished interacting with the object (e.g. by pressing an "OK" button). The return value of ⎕DQ in this case depends on the action code of the event.

Finally, ⎕DQ terminates with a VALUE ERROR if it attempts to execute a callback function that is undefined.

Data Representation (Monadic): R←⎕DR Y

Monadic ⎕DR returns the type of its argument Y. The result R is an integer scalar containing one of the following values. Note that the internal representation and data types for character data differs between the Unicode and Classic Editions.

Value	Data Type
11	1 bit Boolean
80	**8 bits character**
83	**8 bits signed integer**
160	**16 bits character**
163	**16 bits signed integer**
320	**32 bits character**
323	32 bits signed integer
326	32 bits Pointer
645	64 bits Floating

Unicode Edition

Value	Data Type
11	1 bit Boolean
82	8 bits character
83	**8 bits signed integer**
163	**16 bits signed integer**
323	32 bits signed integer
326	32 bits Pointer
645	64 bits Floating

Classic Edition

Note that types **80, 160** and **320** and **83** and **163** are exclusive to Dyalog APL.

Data Representation (Dyadic): R←X ⎕DR Y

Dyadic `⎕DR` converts the data type of its argument `Y` according to the type specification `X`. See monadic `⎕DR` on the previous page for a list of data types.

Case 1:

`X` is a single integer value. The bits in the right argument are interpreted as elements of an array of type `X`. The shape of the resulting new array will typically be changed along the last axis. For example, a character array seen as Boolean will have 8 times as many elements along the last axis.

Case 2:

`X` is a 2-element integer value. The bits in the right argument are interpreted as type `X[1]`. The system then attempts to convert the elements of the resulting array to type `X[2]` without loss of precision. The result `R` is a two element nested array comprised of:

[1] The converted elements or a fill element (0 or blank) where the conversion failed

[2] A Boolean array of the same shape indicating which elements were successfully converted.

Case 3: Classic Edition Only

`X` is a 3-element integer value and `X[2 3]` is `163 82`. The bits in the right argument are interpreted as elements of an array of type `X[1]`. The system then converts them to the character representation of the corresponding 16 bit integers. This case is provided primarily for compatibility with APL*PLUS. For new applications, the use of the [conv] field with `⎕NAPPEND` and `⎕NREPLACE` is recommended.

Conversion to and from character (data type 82) uses the translate vector given by `⎕NXLATE 0`. By default this is the mapping defined by the current output translate table (usually WIN.DOT).

Note. The internal representation of data may be modified during workspace compaction. For example, numeric arrays and (in the Unicode Edition) character arrays will if possible, be squeezed to occupy the least possible amount of memory. However, the internal representation of the result `R` is guaranteed to remain unmodified until it is re-assigned (or partially re-assigned) with the result of any function.

Edit Object: {R}←{X}⎕ED Y

⎕ED invokes the Editor. Y is a simple character vector, a simple character matrix, or a vector of character vectors, containing the name(s) of objects to be edited. The optional left argument X is a character scalar or character vector with as many elements as there are names in Y. Each element of X specifies the type of the corresponding (new) object named in Y, where :

∇	function/operator
→	simple character vector
∊	vector of character vectors
-	character matrix
⍟	Namespace script
○	Class script
∘	Interface

If an object named in Y already exists, the corresponding type specification in X is ignored.

If ⎕ED is called from the Session, it opens Edit windows for the object(s) named in Y and returns a null result. The cursor is positioned in the first of the Edit windows opened by ⎕ED, but may be moved to the Session or to any other window which is currently open. The effect is almost identical to using)ED.

If ⎕ED is called from a defined function or operator, its behaviour is different. On asynchronous terminals, the Edit windows are automatically displayed in "full-screen" mode (ZOOMED). In all implementations, the user is restricted to those windows named in Y. The user may not skip to the Session even though the Session may be visible

⎕ED terminates and returns a result ONLY when the user explicitly closes all the windows for the named objects. In this case the result contains the names of any objects which have been changed, and has the same structure as Y.

Event Message: R←⎕EM Y

Y must be a simple non-negative integer scalar or vector of event codes. If Y is a scalar, R is a simple character vector containing the associated event message. If Y is a vector, R is a vector of character vectors containing the corresponding event messages.

If Y refers to an undefined error code "n", the event message returned is "ERROR NUMBER n".

Example

```
      ⎕EM 11
DOMAIN ERROR
```

Event Number: R←⎕EN

This simple integer scalar reports the identification number for the most recent event which occurred, caused by an APL action or by an interrupt or by the ⎕SIGNAL system function. Its value in a clear workspace is 0.

Exception: R←⎕EXCEPTION

This is a system object that identifies the most recent *Exception* thrown by a Microsoft .Net object.

⎕EXCEPTION derives from the Microsoft .Net class System.Exception. Among its properties are the following, all of which are strings:

Source	The name of the .Net namespace in which the exception was generated
StackTrace	The calling stack
Message	The error message

```
      ⎕USING←'System'
      DT←DateTime.New 100000 0 0
EXCEPTION
      DT←DateTime.New 100000 0 0

      ⎕EN
90
```

```
      ⎕EXCEPTION.Message
Specified argument was out of the range of valid values.

Parameter name: Year, Month, and Day parameters describe
an unrepresentable DateTime.

      ⎕EXCEPTION.Source
mscorlib

      ⎕EXCEPTION.StackTrace
   at System.DateTime.DateToTicks(Int32 year, Int32 month,
                                  Int32 day)

   at System.DateTime..ctor(Int32 year, Int32 month,
                            Int32 day)
```

Expunge Object: {R}←⎕EX Y

Y must be a simple character scalar, vector or matrix. A scalar or vector is treated as a single row matrix. Each row of Y is interpreted as an APL name. R is a simple logical vector with one element per row of Y.

Each name in Y is disassociated from its value if the active referent for the name is a defined function, operator, variable or namespace.

The value of an element of R is 1 if the corresponding name in Y is now available for use. This does not necessarily mean that the existing value was erased for that name. A value of 0 is returned for an ill-formed name or for a distinguished name in Y. The result is suppressed if not used or assigned.

Examples

```
      ⎕EX'VAR'
      +⎕EX¨'FOO' '⎕IO' 'X' '123'
 1  0  1  0
```

If a named object is being executed the existing value will continue to be used until its execution is completed. However, the name becomes available immediately for other use.

Examples

```
      )SI
FOO[1]*

      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇
      +⎕EX'FOO'
1
      )SI
FOO[1]*


     ∇FOO[⎕]
defn error

      FOO←1 2 3
      →⎕LC
10
      FOO
1 2 3
```

If a named object is an external variable, the external array is disassociated from the name:

```
      ⎕XT'F'
FILES/COSTS
      ⎕EX'F' ⋄ ⎕XT'F'
```

If the named object is a GUI object, the object and all its children are deleted and removed from the screen. The expression `⎕EX'.'` deletes all objects owned by the current thread **except** for the Root object itself. In addition, if this expression is executed by thread 0, it resets all the properties of `'.'` to their default values. Furthermore, any unprocessed events in the event queue are discarded.

If the named object is a shared variable, the variable is retracted.

If the named object is the last remaining external function of an auxiliary process, the AP is terminated.

If the named object is the last reference into a dynamic link library, the DLL is freed.

Export Object: {R}←{X}⎕EXPORT Y

⎕EXPORT is used to set or query the export type of a defined function (or operator) referenced by the ⎕PATH mechanism.

Y is a character matrix or vector-of-vectors representing the names of functions and operators whose export type is to be set or queried.

X is an integer scalar or vector (one per name in the namelist) indicating the export type. X can currently be one of the values:

0 - not exported.
1 - exported (default).

A scalar or 1-element-vector type is replicated to conform with a multi-name list.

The result R is a vector that reports the export type of the functions and operators named in Y. When used dyadically to set export type, the result is shy.

When the path mechanism locates a referenced function (or operator) in the list of namespaces in the ⎕PATH system variable, it examines the function's export type:

0	This instance of the function is ignored and the search is resumed at the next namespace in the ⎕PATH list. Type-0 is typically used for functions residing in a utility namespace which are not themselves utilities, for example the private sub-function of a utility function.
1	This instance of the function is executed in the namespace in which is was found and the search terminated. The effect is exactly as if the function had been referenced by its full path name.

Warning: The left domain of ⎕EXPORT may be extended in future to include extra types 2, 3, ... (for example, to change the behaviour of the function). This means that, while ⎕EXPORT returns a boolean result in the first version, this may not be the case in the future. If you need a boolean result, use 0≠ or an equivalent.

```
      (0≠⎕EXPORT ⎕nl 3 4)⌿⎕nl 3 4  ⍝ list of exported
                                   ⍝ functions and operators.
```

File Append Component: {R}←X ⎕FAPPEND Y

Access code 8

Y must be a simple integer scalar or a 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. X may be any array including, for example, the ⎕OR of a namespace.

The shy result R is the number of the component to which X is written, and is 1 greater than the previously highest component number in the file, or 1 if the file is new.

Examples

```
      (1000?1000) ⎕FAPPEND 1

      ⎕←(2 3⍴⍳6) 'Geoff' (⎕OR'FOO') ⎕FAPPEND 1
12

      ⎕←A B C ⎕FAPPEND¨1
13 14 15

Dump←{
    tie←⍺ ⎕FCREATE 0                 ⍝ create file.
    (⎕FUNTIE tie){}⍵ ⎕FAPPEND tie ⍝ append and untie.
}
```

File System Available: R←⎕FAVAIL

This niladic function returns the boolean value 1 unless the component file system is unavailable for some reason, in which case it returns 0. If ⎕FAVAIL does return 0, most of the component file system functions will generate the error message:

```
FILE SYSTEM NOT AVAILABLE
```

See *User Guide* for further details.

File Copy: R←X ⎕FCOPY Y

Access Code: 4609

Y must be a simple integer scalar or 1 or 2-element vector containing the file tie number and optional passnumber. The file need not be tied exclusively.

X is a character vector containing the name of a new file to be copied to.

⎕FCOPY creates a copy of the tied file specified by Y, named X. The new file X will be a 64-bit file, but will otherwise be identical to the original file. In particular all component level information, including the user number and update time, will be the same. The operating system file creation, modification and access times will be set to the time at which the copy occurred.

The result R is the file tie number associated with the new file X.

Example

```
      told←'oldfile32'⎕FTIE 0
      'S' ⎕FPROPS told
32
      tnew←'newfile64' ⎕FCOPY told

      'S' ⎕FPROPS tnew
64
```

If X specifies the name of an existing file, the operation fails with a FILE NAME ERROR.

Note: This operation is atomic. If an error occurs during the copy operation (such as disk full) or if a strong interrupt is issued, the copy will be aborted and the new file X will not be created.

File Create: {R}←X ⎕FCREATE Y

Y must be a simple integer scalar or a 1 or 2 element vector containing the *file tie number* followed by an optional *address size*. .

The *file tie number* must not be the tie number associated with another tied file.

The *address size* is an integer and may be either 32 or 64. A value of 32 causes the internal component addresses to be represented by 32-bit values which allow a maximum file size of 4GB. A value of 64 (the default) causes the internal component addresses to be represented by 64-bit values which allows file sizes up to operating system limits.

Note:

- a 32-bit component file *may not* contain Unicode character data.
- a 64-bit component file may not be accessed by versions of Dyalog APL prior to Version 10.1.0

X must be either

a) a simple character scalar or vector which specifies the name of the file to be created. See *User Guide* for file naming conventions under UNIX and Windows.
b) a vector of length 1 or 2 whose items are:
  i. a simple character scalar or vector as above.
  ii. an integer scalar specifying the file size limit in bytes.

The newly created file is tied for exclusive use.

The shy result of ⎕FCREATE is the tie number of the new file.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

```
      tie←1+⌈/0,⎕FNUMS     ⍝ With next available number,
      file ⎕FCREATE tie    ⍝ ... create file.
```

to:

```
      tie←file ⎕FCREATE 0 ⍝ Create with first available..
```

Examples

```
      '..\BUDGET\SALES'    ⎕FCREATE 2    ⍝ Windows
      '../budget/SALES.85' ⎕FCREATE 2    ⍝ UNIX

      'COSTS' 200000 ⎕FCREATE 4          ⍝ max size 200000

      'LARGE' ⎕FCREATE 5 64              ⍝ 64-bit file
      'SMALL' ⎕FCREATE 6 32              ⍝ 32-bit file
```

File Drop Component: {R}←⎕FDROP Y

Access code 32

Y must be a simple integer vector of length 2 or 3 whose elements are:

[1] a file tie number

[2] a number specifying the position and number of components to be dropped. A positive value indicates that components are to be removed from the beginning of the file; a negative value indicates that components are to be removed from the end of the file

[3] an optional passnumber which if omitted is assumed to be zero

The shy result of a ⎕FDROP is a vector of the numbers of the dropped components. This is analogous to ⎕FAPPEND in that the result is potentially useful for updating some sort of dictionary:

```
      cnos,←vec ⎕FAPPEND¨tie ⍝ Append index to dictionary

      cnos~←⎕FDROP tie,-⍴vec ⍝ Remove index from dict.
```

Note that the result vector, though potentially large, is generated only on request.

Examples

```
      ⎕FSIZE 1
1 21 5436 4294967295

      ⎕FDROP 1 3 ⋄ ⎕FSIZE 1
4 21 5436 4294967295

      ⎕FDROP 1 ¯2 ⋄ ⎕FSIZE 1
4 19 5436 4294967295
```

File Erase: {R}←X ⎕FERASE Y

Access code 4

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. X must be a character scalar or vector containing the name of the file associated with the tie number Y. This name must be identical with the name used to tie the file, and the file must be exclusively tied. The file named in X is erased and untied. See *User Guide* for file naming conventions under UNIX and Windows.

The shy result of ⎕FERASE is the tie number of the erased file.

Examples

```
      'SALES'⎕FERASE 'SALES' ⎕FTIE 0


      './temp' ⎕FCREATE 1
      'temp' ⎕FERASE 1
FILE NAME ERROR
      'temp'⎕FERASE 1
      ^
```

File Hold: {R}←⎕FHOLD Y

Access code 2048

This function holds component file(s) and/or external variable(s).

If applied to component files, then Y is an integer scalar, vector, or one-row matrix of file tie numbers, or a two-row matrix whose first row contains file tie numbers and whose second row contains passnumbers.

If applied to external variables, then Y is a non-simple scalar or vector of character vectors, each of which is the name of an external variable. (NOT the file names associated with those variables).

If applied to component files **and** external variables, `Y` is a vector whose elements are either integer scalars representing tie numbers, or character vectors containing names of external variables.

The effect is as follows :

1. The user's preceding holds (if any) are released.
2. Execution is suspended until the designated files are free of holds by any other task.
3. When all the designated files are free, execution proceeds. Until the hold is released, other tasks using `⎕FHOLD` on any of the designated files will wait.

If `Y` is empty, the user's preceding hold (if any) is released, and execution continues.

A hold is released by any of the following :

1. Another `⎕FHOLD`
2. Untying or retying all the designated files. If some but not all are untied or retied, they become free for another task but the hold persists for those that remain tied.
3. Termination of APL.
4. Any untrapped error or interrupt.
5. A return to immediate execution.

Note that a hold is not released by a request for input through `⎕` or `⍞`.

Note also that point 5 above implies that `⎕FHOLD` is generally useful only when called from a defined function, as holds set in immediate execution (desk calculator) mode are released immediately.

The shy result of `⎕FHOLD` is a vector of tie numbers of the files held.

Examples :

```
      ⎕FHOLD 1

      ⎕FHOLD ⍬

      ⎕FHOLD ⊂'XTVAR'

      ⎕FHOLD 1 2,[0.5]0 16385

      ⎕FHOLD 1 'XTVAR'
```

Fix Script: {R}←{X}⎕FIX Y

⎕FIX fixes a Class from the script specified by Y.

Y must be a vector of character vectors or character scalars that contains a well-formed Class script. If so, the shy result R is a reference to the new Class fixed by ⎕FIX.

The Class specified by Y may be named or unnamed.

If specified, X must be a numeric scalar. If X is omitted or non-zero, and the Class script Y specifies a name (for the Class), ⎕FIX establishes that Class in the workspace.

If X is 0 or the Class specified by Y is unnamed, the Class is not established *per se*, although it will exist for as long as a reference to it exists.

In the first example, the Class specified by Y is *named* (MyClass) but the result of ⎕FIX is discarded. The end-result is that MyClass is established in the workspace as a Class.

```
      ⎕←⎕FIX ':Class MyClass' ':EndClass'
#.MyClass
```

In the second example, the Class specified by Y is *named* (MyClass) and the result of ⎕FIX is assigned to a different name (MYREF). The end-result is that a Class named MyClass is established in the workspace, and MYREF is a reference to it.

```
      MYREF←⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MyClass MYREF
      ⎕NC'MyClass' 'MYREF'
9.4 9.4
      MYREF
#.MyClass
```

In the third example, the left-argument of 0 causes the named Class MyClass to be visible only via the reference to it (MYREF). It is there, but hidden.

```
      MYREF←0 ⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MYREF
      MYREF
#.MyClass
```

The final example illustrates the use of un-named Classes.

```
      src←':Class' '∇Make n'
      src,←'Access Public' 'Implements Constructor'
      src,←'⎕DF n' '∇' ':EndClass'
      MYREF←⎕FIX src
      )CLASSES
MYREF
      MYINST←⎕NEW MYREF'Pete'
      MYINST
Pete
```

Component File Library: R←⎕FLIB Y

Y must be a simple character scalar or vector which specifies the name of the directory whose APL component files are to be listed. If Y is empty, the current working directory is assumed.

The result R is a character matrix containing the names of the component files in the directory with one row per file. The number of columns is given by the longest file name. Each file name is prefixed by Y followed by a directory delimiter character. The ordering of the rows is not defined.

If there are no APL component files accessible to the user in the directory in question, the result is an empty character matrix with 0 rows and 0 columns.

Note that if a file is exclusively tied (as opposed to share tied) then it is not reported by ⎕FLIB.

Examples

```
      ⎕FLIB ''
SALESFILE
COSTS

      ⎕FLIB '.'
./SALESFILE
./COSTS

      ⎕FLIB '../budget'
../budget/SALES.85
../budget/COSTS.85
```

Format (Monadic): R←⎕FMT Y

Y may be any array. R is a simple character matrix which appears the same as the default display of Y. If Y contains control characters from ⎕TC, they will be resolved.

Examples

```
      A←⎕FMT '∩' ,⎕TC[1],'∘'

      ⍴A
1 1
      A
⍝

      A←⎕VR 'FOO'

      A
     ∇ R←FOO
[1]    R←10
     ∇

      ⍴A
31
      B←⎕FMT A

      B
     ∇ R←FOO
[1]    R←10
     ∇

      ⍴B
3 12
```

Format (Dyadic): R←X ⎕FMT Y

Y must be a simple array of rank not exceeding two, or a non-simple scalar or vector whose items are simple arrays of rank not exceeding two. The simple arrays in Y must be homogeneous, either character or numeric. X must be a simple character vector. R is a simple character matrix.

X is a format specification that defines how columns of the simple arrays in Y are to appear. A simple scalar in Y is treated as a one-element matrix. A simple vector in Y is treated as a one-column matrix. Each column of the simple arrays in Y is formatted in left-to-right order according to the format specification in X taken in left-to-right order and used cyclically if necessary.

R has the same number of columns as the longest column (or implied column) in Y, and the number of rows is determined from the format specification.

The **format specification** consists of a series of control phrases, with adjacent phrases separated by a single comma, selected from the following:

`rAw`	`Alphanumeric format`
`rEw.s`	`Scaled format`
`rqFw.d`	`Decimal format`
`rqG⍞pattern⍞`	`Pattern`
`rqIw`	`Integer format`
`Tn`	`Absolute tabulation`
`Xn`	`Relative tabulation`
`⍞t⍞`	`Text insertion.`

(Alternative surrounding pairs for Pattern or Text insertion are
`< >`, `⊂ ⊃`, `⎕ ⎕` or `¨ ¨`.)

where:

`r`	is an optional repetition factor indicating that the format phrase is to be applied to r columns of Y.
`q`	is an optional usage of qualifiers or affixtures from those described below.
`w`	is an integer value specifying the total field width per column of Y, including any affixtures.

`s`	is an integer value specifying the number of significant digits in Scaled format; `s` must be less than `w-1`.
`d`	is an integer value specifying the number of places of decimal in Decimal format; `d` must be less than `w`.
`n`	is an integer value specifying a tab position relative to the notional left margin (for `T`-format) or relative to the last formatted position (for `X`-format) at which to begin the next format.
`t`	is any arbitrary text excluding the surrounding character pair. Double quotes imply a single quote in the result.
`pattern`	see following section **G format**

Qualifiers `q` are as follows:

`B`	leaves the field blank if the result would otherwise be zero.
`C`	inserts commas between triads of digits starting from the rightmost digit of the integer part of the result.
`Km`	scales numeric values by `1Em` where m is an integer; negation may be indicated by `¯` or - preceding the number.
`L`	left justifies the result in the field width.
`Ov□t□`	replaces specific numeric value `v` with the text `t`.
`S□p□`	substitutes standard characters. p is a string of pairs of symbols enclosed between any of the Text Insertion delimiters. The first of each pair is the standard symbol and the second is the symbol to be substituted. Standard symbols are:

`*`	overflow fill character
`.`	decimal point
`,`	triad separator for `C` qualifier
`0`	fill character for `Z` qualifier
`_`	loss of precision character

`Z`	fills unused leading positions in the result with zeros (and commas if `C` is also specified).
`9`	digit selector

Affixtures are as follows:

`M⎕t⎕`	prefixes negative results with the text t instead of the negative sign.
`N⎕t⎕`	post-fixes negative results with the text `t`.
`P⎕t⎕`	prefixes positive or zero results with the text `t`.
`Q⎕t⎕`	post-fixes positive or zero results with the text `t`.
`R⎕t⎕`	presets the field with the text `t` which is repeated as necessary to fill the field. The text will be replaced in parts of the field filled by the result, including the effects of other qualifiers and affixtures except the `B` qualifier.

The surrounding affixture delimiters may be replaced by the alternative pairs described for Text Insertion.

Examples

A vector is treated as a column:

```
      'I5' ⎕FMT 10 20 30
   10
   20
   30
```

The format specification is used cyclically to format the columns of the right argument:

```
      'I3,F5.2' ⎕FMT 2 4⍴⍳8
  1 2.00  3 4.00
  5 6.00  7 8.00
```

The columns of the separate arrays in the items of a non-simple right argument are formatted in order. Rows in a formatted column beyond the length of the column are left blank:

```
      '2I4,F7.1' ⎕FMT (⍳4)(2 2⍴ 0.1×⍳4)
   1   0    0.2
   1   0    0.4
   3
   4
```

Characters are right justified within the specified field width, unless the `L` qualifier is specified:

```
      'A2' ⎕FMT 1 6⍴'SPACED'
 S P A C E D
```

If the result is too wide to fit within the specified width, the field is filled with asterisks:

```
      'F5.2' ⎕FMT 0.1×5 1000 ¯100
 0.50
*****
*****
```

Relative tabulation (`X`-format) identifies the starting position for the next format phrase relative to the finishing position for the previous format, or the notional left margin if none. Negative values are permitted providing that the starting position is not brought back beyond the left margin. Blanks are inserted in the result, if necessary:

```
      'I2,X3,3A1' ⎕FMT (⍳3)(2 3⍴'TOPCAT')
 1   TOP
 2   CAT
 3
```

Absolute tabulation (`T`-format) specifies the starting position for the next format relative to the notional left margin. If position 0 is specified, the next format starts at the next free position as viewed so far. Blanks are inserted into the result as required. Over-written columns in the result contain the most recently formatted array columns taken in left-to-right order:

```
      X←'6I1,T5,A1,T1,3A1,T7,F5.1'

      X ⎕FMT (1 6⍴⍳6)('*')(1 3⍴'ABC')(22.2)
ABC4*6 22.2
```

If the number of specified significant digits exceeds the internal precision, low order digits are replaced by the symbol _:

```
      'F20.1' ⎕FMT 1E18÷3
333333333333333__._
```

The Text Insertion format phrase inserts the given text repeatedly in all rows of the result:

```
      MEN←3 5⍴'FRED BILL JAMES'
      WOMEN←2 5⍴'MARY JUNE '

      '5A1,<|>' ⎕FMT MEN WOMEN
FRED |MARY |
BILL |JUNE |
JAMES|     |
```

The last example also illustrates that a Text Insertion phrase is used even though the data is exhausted. The following example illustrates effects of the various qualifiers:

```
      X←'F5.1,BF6.1,X1,ZF5.1,X1,LF5.1,K3CS<.,,.>F10.1'

      X ⎕FMT ⍉5 3⍴¯1.5 0 25
 ¯1.5  ¯1.5 ¯01.5 ¯1.5   ¯1.500,0
  0.0       000.0  0.0          0,0
 25.0  25.0 025.0 25.0   25.000,0
```

Affixtures allow text to be included within a field. The field width is not extended by the inclusion of affixtures. N and Q affixtures shift the result to the left by the number of characters in the text specification. Affixtures may be used to enclose negative results in parentheses in accordance with common accounting practice:

```
      'M<(>N<)>Q< >F9.2' ⎕FMT 150.3 ¯50.25 0 1114.9
  150.30
  (50.25)
    0.00
 1114.90
```

One or more format phrases may be surrounded by parentheses and preceded by an optional repetition factor. The format phrases within parentheses will be re-used the given number of times before the next format phrase is used. A Text Insertio388 phrase will not be re-used if the last data format phrase is preceded by a closing parenthesis:

```
      'I2,2(</>,ZI2)' ⎕FMT 1 3⍴⌽100|3↑⎕TS
20/07/89
```

G Format

Only the `B`, `K`, `S` and `O` qualifiers are valid with the `G` option

`⎕pattern⎕` is an arbitrary string of characters, excluding the delimiter characters. Characters '9' and '`Z`' (unless altered with the `S` qualifier) are special and are known as **digit selectors.**

The result of a `G` format will have length equal to the length of the pattern.

The data is rounded to the nearest integer (after possible scaling). Each digit of the rounded data replaces one digit selector in the result. If there are fewer data digits than digit selectors, the data digits are padded with leading zeros. If there are more data digits than digit selectors, the result will be filled with asterisks.

A '9' digit selector causes a data digit to be copied to the result.

A 'Z' digit selector causes a non-zero data digit to be copied to the result. A zero data digit is copied if and only if digits appear on either side of it. Otherwise a blank appears. Similarly text between digit selectors appears only if digits appear on either side of the text. Text appearing before the first digit selector or after the last will always appear in the result.

Examples

```
      'G⊂99/99/99⊃'⎕FMT 0 100 100 ⊥8 7 89
08/07/89

      'G⊂ZZ/ZZ/ZZ⊃'⎕FMT 80789 + 0 1
 8/07/89
 8/07/9

      'G⊂Andy ZZ Pauline ZZ⊃' ⎕FMT 2721.499 2699.5
Andy 27 Pauline 21
Andy 27

      ρ⎕←'K2G⊂DM Z.ZZZ.ZZ9,99⊃' ⎕FMT 1234567.89 1234.56
DM 1.234.567,89
DM     1.234,56
2 15
```

An error will be reported if:

1. Numeric data is matched against an `A` control phrase.
2. Character data is matched against other than an `A` control phrase.
3. The format specification is ill-formed.
4. For an F control phrase, `d>w-2`
5. For an E control phrase, `s>w-2`

O Format Qualifier

The O format qualifier replaces a specific numeric value with a text string and may be used in conjunction with the E, F, I and G format phrases.

An O-qualifier consists of the letter "O" followed by the optional numeric value which is to be substituted (if omitted, the default is 0) and then the text string within pairs of symbols such as "<>". For example:

O - qualifier	Description
`O<nil>`	Replaces the value 0 with the text "nil"
`O42<N/A>`	Replaces the value 42 with the text "N/A"
`O0.001<1/1000>`	Replaces the value 0.001 with the text "1/1000"

The replacement text is inserted into the field in place of the numeric value. The text is normally right-aligned in the field, but will be left-aligned if the L qualifier is also specified.

It is permitted to specify more than one O-qualifier within a single phrase.

The O-qualifier is ⎕CT sensitive.

Examples

```
      'O<NIL>F7.2'⎕FMT 12.3 0 42.5
  12.30
    NIL
  42.50

      'O<NIL>LF7.2'⎕FMT 12.3 0 42.5
12.30
NIL
42.50

      'O<NIL>O42<N/A>I6'⎕FMT 12 0 42 13
    12
   NIL
   N/A
    13

      'O99<replace>F20.2'⎕fmt 99 100 101
             replace
              100.00
              101.00
```

File Names: R←⎕FNAMES

The result is a character matrix containing the names of all tied files, with one file name per row. The number of columns is that required by the longest file name.

A file name is returned precisely as it was specified when the file was tied. If no files are tied, the result is a character matrix with 0 rows and 0 columns. The rows of the result are in the order in which the files were tied.

Examples

```
      '/usr/pete/SALESFILE' ⎕FSTIE 16

      '../budget/COSTFILE' ⎕FSTIE 2

      'PROFIT' ⎕FCREATE 5

       ⎕FNAMES
/usr/pete/SALESFILE
../budget/COSTFILE
PROFIT

      ⍴⎕FNAMES
3 19
      ⎕FNUMS,⎕FNAMES
16 /usr/pete/SALESFILE
 2 ../budget/COSTFILE
 5 PROFIT
```

File Numbers: R←⎕FNUMS

The result is an integer vector of the **tie numbers** of all tied files. If no files are tied, the result is empty. The elements of the result are in the order in which the files were tied.

Examples

```
      '/usr/pete/SALESFILE' ⎕FSTIE 16

      '../budget/COSTFILE' ⎕FSTIE 2

      'PROFIT' ⎕FCREATE 5

      ⎕FNUMS
16 2 5

      ⎕FNUMS,⎕FNAMES
16 /usr/pete/SALESFILE
 2 ../budget/COSTFILE
 5 PROFIT

      ⎕FUNTIE ⎕FNUMS
      ⍴⎕FNUMS
0
```

File Properties: R←X ⎕FPROPS Y

Access Code 1 or 8192

`⎕FPROPS` reports and sets the properties of a component file.

`Y` must be a simple integer scalar or vector containing the file tie number.

`X` must be a simple character scalar or vector containing one or more valid Identifiers listed in the table below, or a vector of 2-element vectors, each of which contains an Identifier and a (new) value for that property.

If the left argument is a simple character array, the result `R` contains the current values for the properties identified by `X`. If the left argument is nested, the result `R` contains the previous values for the properties identified by `X`

Identifier	Property	Description / Legal Values
`S`	File Size (read only)	32 = Small Component Files (<4Gb) 64 = Large Component Files
`E`	Endian-ness (read only)	0 = Little-endian 1 = Big-endian
`U`	Unicode	0 = Characters must be written as type 82 arrays 1 = Characters must be written as Unicode arrays
`J`	Journaling	0 = Disable Journaling 1 = Enable Journaling

The default properties for a newly created file are as follows:

- S = 64
- U = 1 (Unicode Edition and 64-bit file) or 0 (otherwise)
- J = 0.
- E depends upon the computer architecture.

Example

```
      tn←'myfile64' ⎕FCREATE 0
      'SEUJ' ⎕FPROPS tn
64 0 1 0

      tn←'myfile32' ⎕FCREATE 0 32
      'SEUJ' ⎕FPROPS tn
32 0 0 0
```

The following expression disables Unicode and switches Journaling on. The function returns the previous settings:

```
      ('U' 0)('J' 1) ⎕FPROPS tn
1 0
```

The Unicode property applies only to 64-bit component files. 32-bit component files *may not* contain Unicode character data and the value of the Unicode property is always 0. To convert a 32-bit component file to a 64-bit component file, use `⎕FCOPY`.

Properties may be read by a task with `⎕FREAD` permission (access code 1), and set by a task with `⎕FSTAC` access (8192). To set the value of the Journaling property, the file must be exclusively tied.

If Journaling or Unicode properties are set, the file cannot be tied by Versions prior to Version 12.0.

File Read Access: R←⎕FRDAC Y

Access code 4096

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. The result is the access matrix for the designated file.

See "File Access Control" in *User Guide* for further details.

Examples

```
      ⎕FRDAC 1
28 2105 16385
 0 2073 16385
31   ¯1     0
```

File Read Component Information: R←⎕FRDCI Y

Access code 512

Y must be a simple integer vector of length 2 or 3 containing the file tie number, component number and an optional passnumber. If the passnumber is omitted it is assumed to be zero.

The result is a 3 element numeric vector containing the following information:

a) the size of the component in bytes (i.e. how much disk space it occupies).
b) the user number of the user who last updated the component.
c) the time of the last update in 60ths of a second since 1st January 1970.

Example

```
      ⎕FRDCI 1 13
2200 207 3.702094494E10
```

File Read Component: R←⎕FREAD Y

Access code 1

Y must be a simple 2 or 3 element integer vector containing the file tie number, the component number, and an optional passnumber. If the passnumber is omitted it is assumed to be zero.

The result is the value of the array stored on the tied file at the given component number.

Examples

```
      ⍴SALES←⎕FREAD 1 241
3 2 12

GetFile←{⎕io←0                    ⍝ Extract contents.
    tie←⍵ ⎕fstie 0                ⍝ new tie number.
    fm to←2↑⎕fsize tie            ⍝ first and next component.
    cnos←fm+⍳to-fm                ⍝ vector of component nos.
    cvec←{⎕fread tie ⍵}¨cnos      ⍝ vector of components.
    cvec{⍺}⎕funtie tie            ⍝ ... untie and return.
}
```

File Rename: {R}←X ⎕FRENAME Y

Access code 128

Y must be a simple 1 or 2 element integer vector containing a file tie number and an optional passnumber. If the passnumber is omitted it is assumed to be zero.

X must be a simple character scalar or vector containing the new name of the file. This name must be in accordance with the operating system's conventions, and may be specified with a relative or absolute pathname.

The file being renamed must be tied exclusively.

The shy result of ⎕FRENAME is the tie number of the file.

Examples

```
      'SALES' ⎕FTIE 1
      'PROFIT' ⎕FTIE 2

      ⎕FNAMES
SALES
PROFIT

      'SALES.85' ⎕FRENAME 1
      '../profits/PROFIT.85' ⎕FRENAME 2

      ⎕FNAMES
SALES.85
../profits/PROFITS.85

Rename←{
    fm to←⍵
    ⎕FUNTIE to ⎕FRENAME fm ⎕FTIE 0
}
```

File Replace Component: {R}←X ⎕FREPLACE Y

Access code 16

Y must be a simple 2 or 3 element integer vector containing the file tie number, the component number, and an optional passnumber. If the passnumber is omitted it is assumed to be zero. The component number specified must lie within the file's component number limits.

X is any array (including, for example, the ⎕OR of a namespace), and overwrites the value of the specified component. The component information (see ⎕FRDCI) is also updated.

The shy result of ⎕FREPLACE is the file index (component number of replaced record).

Example

```
      SALES←⎕FREAD 1 241

      (SALES×1.1) ⎕FREPLACE 1 241
```

Define a function to replace (index, value) pairs in a component file JMS.DCF:

```
Frep←{
    tie←⍺ ⎕FTIE 0
    _←{⍵ ⎕FREPLACE tie ⍺}/¨⍵
    ⎕FUNTIE tie
}

      'jms'Frep(3 'abc')(29 'xxx')(7 'yyy')
```

File Resize: {R}←{X}⎕FRESIZE Y

Access code 1024

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero.

X is included for compatibility with other APL systems.

A side effect of ⎕FRESIZE is to cause the file to be compacted. Any interrupt entered at the keyboard during the compaction is ignored.

The shy result of ⎕FRESIZE is the tie number of the file.

Example

```
      ⎕FSIZE 1
1 21 65271 4294967295

      200000 ⎕FRESIZE 1 ⋄ ⎕FSIZE 1
1 21 41456 4294967295

      0 ⎕FRESIZE 1      ⍝ Force file compaction.
```

File Size: R←⎕FSIZE Y

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. The result is a 4 element numeric vector containing the following:

Element	Description
1	the number of first component
2	1 + the number of the last component, (i.e. the result of the next ⎕FAPPEND)
3	the current size of the file in bytes
4	the file size limit in bytes

Example

```
      ⎕FSIZE 1
1 21 65271 4294967295
```

File Set Access: {R}←X ⎕FSTAC Y

Access code 8192

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero.

X must be a valid access matrix, i.e. a 3 column integer matrix with any number of rows.

See "File Access Control" in *User Guide* for further details.

The shy result of ⎕FSTAC is the tie number of the file.

Examples

```
      SALES ⎕FCREATE 1
      (3 3⍴28 2105 16385 0 2073 16385 31 ¯1 0) ⎕FSTAC 1
      ((⎕FRDAC 1)⍪21 2105 16385) ⎕FSTAC 1

      (1 3⍴0 ¯1 0)⎕FSTAC 2
```

File Share Tie: {R}←X ⎕FSTIE Y

`Y` must be 0 or a simple 1 or 2 element integer vector containing an available file tie number to be associated with the file for further file operations, and an optional passnumber. If the passnumber is omitted it is assumed to be zero. The tie number must not already be associated with a tied file.

`X` must be a simple character scalar or vector which specifies the name of the file to be tied. The file must be named in accordance with the operating system's conventions, and may be specified with a relative or absolute pathname.

The file must exist and be accessible by the user. If it is already tied by another task, it must not be tied exclusively.

The shy result of `⎕FSTIE` is the tie number of the file.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create, share tie or exclusive tie operation, allocates the first (closest to zero) available tie number and returns it as an explicit result. This allows you to simplify code. For example:

from:

```
      tie←1+⌈/0,⎕FNUMS  ⍝ With next available number,
      file ⎕FSTIE tie   ⍝ ... share tie file.
```

to:

```
      tie←file ⎕FSTIE 0 ⍝ Tie with first available number.
```

Example

```
      'SALES' ⎕FSTIE 1

      '../budget/COSTS' ⎕FSTIE 2
```

Exclusive File Tie: {R}←X ⎕FTIE Y

Access code 2

Y must be 0 or a simple 1 or 2 element integer vector containing an available file tie number to be associated with the file for further file operations, and an optional passnumber. If the passnumber is omitted it is assumed to be zero. The tie number must not already be associated with a share tied or exclusively tied file.

X must be a simple character scalar or vector which specifies the name of the file to be exclusively tied. The file must be named in accordance with the operating system's conventions, and may be a relative or absolute pathname.

The file must exist and be accessible by the user. It may not already be tied by another user.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create, share tie or exclusive tie operation, allocates the first (closest to zero) available tie number, and returns it as an explicit result,. This allows you to simplify code. For example:

from:

```
      tie←1+⌈/0,⎕FNUMS ⍝ With next available number,
      file ⎕FTIE tie   ⍝ ... tie file.
```

to:

```
      tie←file ⎕FTIE 0 ⍝ Tie with first available number.
```

The shy result of ⎕FTIE is the tie number of the file.

Examples

```
      'SALES' ⎕FTIE 1

      '../budget/COSTS' ⎕FTIE  2

      '../budget/expenses' ⎕FTIE 0
3
```

File Untie: {R}←⎕FUNTIE Y

Y must be a simple integer scalar or vector (including Zilde). Files whose tie numbers occur in Y are untied. Other elements of Y have no effect.

If Y is empty, no files are untied, but all internal file buffers are written to disk. Under UNIX this is achieved with `fsync`; under Windows with the Commit File function (Int 21h Function 68h); on Win32 systems it is achieved with `FlushFileBuffers()`. This special facility allows the programmer to add extra security (at the expense of performance) for application data files.

The shy result of ⎕FUNTIE is a vector of tie numbers of the files **actually untied**.

Example

```
      ⎕FUNTIE ⎕FNUMS ⍝ Unties all tied files

      ⎕FUNTIE ⍬      ⍝ Flushes all buffers to disk
```

Fix Definition: {R}←⎕FX Y

Y is the representation form of a function or operator which may be:

1. its canonical representation form similar to that produced by ⎕CR except that redundant blanks are permitted other than within names and constants.

2. its nested representation form similar to that produced by ⎕NR except that redundant blanks are permitted other than within names and constants.

3. its object representation form produced by ⎕OR.

4. its vector representation form similar to that produced by ⎕VR except that additional blanks are permitted other than within names and constants.

R is either a simple character vector or an integer scalar. A result is not returned if a result is not explicitly used or assigned.

A side effect of ⎕FX is to create (fix) a function or operator in the workspace or current namespace from the definition given by Y. ⎕IO is an implicit argument of ⎕FX.

If the function or operator is successfully fixed, R is its name. Otherwise R is the row of the canonical representation form in which the first error preventing its definition is detected. An integer R is dependent on ⎕IO.

Functions and operators which are pendent, that is, in the State Indicator without a suspension mark (*), retain their original definition until they complete, or are cleared from the State Indicator. All other occurrences of the function or operator assume the new definition. The function or operator will fail to fix if it has the same name as an existing variable, or a visible label.

Instances: R←⎕INSTANCES Y

⎕INSTANCES returns a list all the current instances of the Class specified by Y.

Y must be a reference to a Class.
R is a vector of references to all existing Instances of Class Y.

Examples

This example illustrates a simple inheritance tree or Class hierarchy. There are 3 Classes, namely:

```
Animal
       Bird (derived from Animal)
             Parrot (derived from Bird)
```

```
:Class Animal
...
:EndClass ⍝ Animal

:Class Bird: Animal
...
:EndClass ⍝ Bird

:Class Parrot: Bird
...
:EndClass ⍝ Parrot

      Eeyore←⎕NEW Animal
      Robin←⎕NEW Bird
      Polly←⎕NEW Parrot

      ⎕INSTANCES Parrot
 #.[Parrot]
      ⎕INSTANCES Bird
 #.[Bird]  #.[Parrot]
      ⎕INSTANCES Animal
 #.[Animal]  #.[Bird]  #.[Parrot]

      Eeyore.⎕DF 'eeyore'
      Robin.⎕DF 'robin'
      Polly.⎕DF 'polly'
      ⎕INSTANCES Parrot
polly
      ⎕INSTANCES Bird
robin  polly
      ⎕INSTANCES Animal
eeyore  robin  polly
```

Index Origin: ⎕IO

`⎕IO` determines the index of the first element of a non-empty vector.

`⎕IO` may be assigned the value 0 or 1. The value in a clear workspace is 1.

`⎕IO` is an implicit argument of any function derived from the Axis operator (`[K]`), of the monadic functions Fix (`⎕FX`), Grade Down (`⍒`), Grade Up (`⍋`), Index Generator (`⍳`), Roll (`?`), and of the dyadic functions Deal (`?`), Find (`⍷`), Grade Down (`⍒`), Grade Up (`⍋`), Index Of (`⍳`), Indexed Assignment, Indexing, Pick (`⊃`) and Transpose (`⍉`).

Examples

```
      ⎕IO←1
      ⍳5
1 2 3 4 5

      ⎕IO←0
      ⍳5
0 1 2 3 4

      +/[0]2 3⍴⍳6
3 5 7

      'ABC',[¯.5]'='
ABC
===
```

Key Label: R←⎕KL Y

Classic Edition only.

Y is a simple character vector or a vector of character vectors containing Input Codes for Keyboard Shortcuts. In the Classic Edition, keystrokes are associated with Keyboard Shortcuts by the Input Translate Table.

R is a simple character vector or a vector of character vectors containing the labels associated with the codes. If Y specifies codes that are not defined, the corresponding elements of R are the codes in Y.

⎕KL provides the information required to build device-independent help messages into applications, particularly full-screen applications using ⎕SM and ⎕SR.

Examples :

```
      ⎕KL 'RC'
Right

      ⎕KL 'ER' 'EP' 'QT' 'F1' 'F13'
 Enter  Esc  Shift+Esc  F1  Shift+F1
```

Line Count: R←⎕LC

This is a simple vector of line numbers drawn from the state indicator (See *Chapter 2*). The most recently activated line is shown first. If a value corresponds to a defined function in the state indicator, it represents the current line number where the function is either suspended or pendent.

The value of ⎕LC changes immediately upon completion of the most recently activated line, or upon completion of execution within ⍎ or ⎕. If a ⎕STOP control is set, ⎕LC identifies the line on which the stop control is effected. In the case where a stop control is set on line 0 of a defined function, the first entry in ⎕LC is 0 when the control is effected.

The value of ⎕LC in a clear workspace is the null vector.

Examples

```
      )SI
TASK1[5]*
⍎
BEGIN[3]

      ⎕LC
5 3
```

```
      →⎕LC
      ⎕LC

      ⍴⎕LC
0
```

Load Workspace: ⎕LOAD Y

Y must be a simple character scalar or vector containing the identification of a saved workspace.

If Y is ill-formed or does not identify a saved workspace or the user account does not have access permission to the workspace, a DOMAIN ERROR is reported.

Otherwise, the active workspace is replaced by the workspace identified in Y. The active workspace is lost. If the loaded workspace was saved by the)SAVE system command, the latent expression (⎕LX) is immediately executed, unless APL was invoked with the -x option. If the loaded workspace was saved by the ⎕SAVE system function, execution resumes from the point of exit from the ⎕SAVE function, with the result of the ⎕SAVE function being 0.

The workspace identification and time-stamp when saved is not displayed.

If the workspace contains any GUI objects whose Visible property is 1, these objects will be displayed. If the workspace contains a non-empty ⎕SM but does not contain an SM GUI object, the form defined by ⎕SM will be displayed in a window on the screen.

Lock Definition: {X}⎕LOCK Y

Y must be a simple character scalar, or vector which is taken to be the name of a defined function or operator in the active workspace.

The active referent to the name in the workspace is locked. Stop or trace vectors, formerly set by the ⎕STOP and ⎕TRACE functions, are cancelled.

The optional left argument X specifies to what extent the function code is hidden. X may be 1, 2 or 3 (the default) with the following meaning:

1. The object may not be displayed and you may not obtain its character form using ⎕CR, ⎕VR or ⎕NR.
2. Execution cannot be suspended with the locked function or operator in the state indicator. On suspension of execution the state indicator is cut back to the statement containing the call to the locked function or operator.
3. Both 1 and 2 apply. You can neither display the locked object nor suspend execution within it.

Locks are additive, so that the following are equivalent:

```
      1 ⎕LOCK'FOO'
      2 ⎕LOCK'FOO'

      3 ⎕LOCK'FOO'
```

A DOMAIN ERROR is reported if:
1. Y is ill-formed.
2. The name in Y is not the name of a visible defined function or operator which is not locked.

Examples

```
      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇

      ⎕LOCK'FOO'
      ⎕VR'FOO'


      ⎕LOCK'FOO'
DOMAIN ERROR
      ⎕LOCK'FOO'
      ^
```

Latent Expression: ⎕LX

This may be a character vector or scalar representing an APL expression. The expression is executed automatically when the workspace is loaded. If APL is invoked using the `-x` flag, this execution is suppressed.

The value of `⎕LX` in a clear workspace is `''`.

Example

```
      ⎕LX←'''GOOD MORNING PETE'''

      )SAVE GREETING
GREETING saved Tue Sep 8 10:49:29 1998

      )LOAD GREETING
./GREETING saved Tue Sep 8 10:49:29 1998
GOOD MORNING PETE
```

Map File: R←{X}⎕MAP Y

`⎕MAP` function associates a mapped file with an APL array in the workspace.

Two types of mapped files are supported; *APL* and *raw*. An *APL* mapped file contains the binary representation of a Dyalog APL array, including its header. A file of this type must be created using the supplied utility function `∆MPUT`. When you map an APL file, the rank, shape and data type of the array is obtained from the information on the file.

A *raw* mapped file is an arbitrary collection of bytes. When you map a raw file, you must specify the characteristics of the APL array to be associated with this data. In particular, the data type and its shape.

The type of mapping is determined by the presence (raw) or absence (APL) of the left argument to `⎕MAP`.

The right argument `Y` specifies the name of the file to be mapped and, optionally, the access type and a start byte in the file. `Y` may be a simple character vector, or a 2 or 3-element nested vector containing:

1. file name (character scalar/vector)
2. access code (character scalar/vector) : one of : 'R' , 'W', 'r' or 'w'
3. start byte offset (integer scalar/vector). Must be a multiple of 4 (default 0)

If `X` is specified, it defines the type and shape to be associated with *raw* data on file. `X` must be an integer scalar or vector. The first item of `X` specifies the data type and must be one of the following values:

Classic Edition	11, 82, 83, 163, 323 or 645
Unicode Edition	11, 80, 83, 160, 163, 320, 323 or 645

Following items determine the shape of the mapped array. A value of `¯1` on any (but normally the first) axis in the shape is replaced by the system to mean: read as many complete records from the file as possible. Only one axis may be specified in this way.

If no left argument is given, file is assumed to contain a simple APL array, complete with header information (type, rank, shape, etc).

Mapped files may be updated by changing the associated array using indexed assignment: `var[a]←b`.

Note that a *raw* mapped file may be updated *only* if its *file offset* is 0.

Examples

Map raw file as a read-only *vector* of doubles:

```
      vec←645 ¯1 ⎕MAP'c:\myfile'
```

Map raw file as a 20-column read-write *matrix* of 1-byte integers:

```
      mat←83 ¯1 20 ⎕MAP'c:\myfile' 'W'
```

Replace some items in mapped file:

```
      mat[2 3;4 5]←2 2⍴⍳4
```

Map bytes 100-180 in raw file as a 5×2 read-only matrix of doubles:

```
      dat←645 5 2 ⎕MAP'c:\myfile' 'R' 100
```

Put simple 4-byte integer array on disk ready for mapping:

```
      (⊃83 323 ⎕DR 2 3 4⍴⍳24)∆MPUT'c:\myvar'
```

Then, map a read-write variable:

```
      var←⎕MAP'c:\myvar' 'w'
```

Note that a mapped array need not be *named*. In the following example, a ‘raw’ file is mapped, summed and released, all in a single expression:

```
      +/163 ¯1 ⎕MAP'c:\shorts.dat'
42
```

Compatibility between Editions

In the Unicode Edition `⎕MAP` will fail with a `TRANSLATION ERROR` (event number 92) if you attempt to map an APL file which contains character data type 82.

In order for the Unicode Edition to correctly interpret data in a raw file that was written using data type 82, the file may be mapped with data type 83 and the characters extracted by indexing into `⎕AVU`.

Migration Level: ⎕ML

`⎕ML` determines the degree of migration of the Dyalog APL language towards IBM's APL2. Setting this variable to other than its default value of `0` changes the interpretation of certain symbols and language constructs.

`⎕ML←0`		Native Dyalog (Default)
`⎕ML←1`	`Z←∊R`	Monadic `'∊'` is interpreted as 'enlist' rather than 'type'.
`⎕ML←2`	`Z←↑R`	Monadic `'↑'` is interpreted as 'first' rather than 'mix'.
	`Z←⊃R`	Monadic `'⊃'` is interpreted as 'mix' rather than 'first'.
	`Z←≡R`	Monadic `'≡'` returns the absolute value of the depth of its argument, rather than a negative value if the depths of its subarrays are unequal
`⎕ML←3`	`R←X⊂[K]Y`	Dyadic `'⊂'` follows the APL2 (rather than the original Dyalog APL) convention
	`⎕TC`	The order of the elements of `⎕TC` is the same as in APL2

Subsequent versions of Dyalog APL may provide further migration levels.

Examples

```
      X←2(3 4)

      ⎕ML←0
      ∊X
0  0 0
      ↑X
2 0
3 4
      ⊃X
2
      ≡X
¯2
```

```
      ⎕ML←1
      ∊X
2 3 4
      ↑X
2 0
3 4
      ⊃X
2
      ≡X
¯2

      ⎕ML←2
      ∊X
2 3 4
      ↑X
2
      ⊃X
2 0
3 4
      ≡X
2
```

Set Monitor: {R}←X ⎕MONITOR Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. X must be a simple non-negative integer scalar or vector. R is a simple integer vector of non-negative elements.

X identifies the numbers of lines in the function or operator named by Y on which a monitor is to be placed. Numbers outside the range of line numbers in the function or operator (other than 0) are ignored. The number 0 indicates that a monitor is to be placed on the function or operator as a whole. The value of X is independent of ⎕IO.

R is a vector of numbers on which a monitor has been placed in ascending order. The result is suppressed unless it is explicitly used or assigned.

The effect of ⎕MONITOR is to accumulate timing statistics for the lines for which the monitor has been set. See Monitor Query for details.

Examples

```
      +(0,⍳10) ⎕MONITOR 'FOO'
0 1 2 3 4 5
```

Existing monitors are cancelled before new ones are set:

```
      +1 ⎕MONITOR 'FOO'
1
```

All monitors may be cancelled by supplying an empty vector:

```
      ⍬ ⎕MONITOR 'FOO'
```

Monitors may be set on a locked function or operator, but no information will be reported. Monitors are saved with the workspace.

Query Monitor: R←⎕MONITOR Y

`Y` must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. `R` is a simple non-negative integer matrix of 5 columns with one row for each line in the function or operator `Y` which has the monitor set, giving:

Column 1 :	Line number
Column 2 :	Number of times the line was executed
Column 3 :	CPU time in milliseconds
Column 4 :	Elapsed time in milliseconds
Column 5 :	Reserved

The value of `0` in column one indicates that the monitor is set on the function or operator as a whole.

Example

```
      ∇ FOO
[1]   A←?25 25⍴100
[2]   B←⌹A
[3]   C←⌹B
[4]   R1←⌊0.5+A+.×B
[5]   R2←A=C
      ∇

      (0,⍳5) ⎕MONITOR 'FOO' ⍝ Set monitor

      FOO                   ⍝ Run function

      ⎕MONITOR 'FOO'        ⍝ Monitor query
0 1 1418 1000 0
1 1   83    0 0
2 1  400    0 0
3 1  397    0 0
4 1  467 1000 0
5 1  100    0 0
```

Name Association: {R}←{X}⎕NA Y

⎕NA provides access from APL to compiled functions within a **Dynamic Link Library** (**DLL**). A DLL is a collection of functions typically written in C (or C++) each of which may take arguments and return a result.

Instructional examples using ⎕NA can be found in supplied workspace: QUADNA.DWS.

The DLL may be part of the standard operating system software, purchased from a third party supplier, or one that you have written yourself.

The right argument Y is a character vector that identifies the name and syntax of the function to be associated. The left argument X is a character vector that contains the name to be associated with the external function. If the ⎕NA is successful, a function (name class 3) is established in the active workspace with name X. If X is omitted, the name of the external function itself is used for the association.

The shy result R is a character vector containing the name of the external function that was fixed.

For example, `math.dll` might be a library of mathematical functions containing a function `divide`. To associate the APL name `div` with this external function:

```
      'div' ⎕NA 'F8 math|divide I4 I4'
```

where F8 and I4, specify the types of the result and arguments expected by `divide`. The association has the effect of establishing a new function: `div` in the workspace, which when called, passes its arguments to `divide` and returns the result.

```
      )fns
div
      div 10 4
2.5
```

Type Declaration

In a compiled language such as C, the types of arguments and results of functions must be declared explicitly. Typically, these types will be published with the documentation that accompanies the DLL. For example, function `divide` might be declared:

```
double divide(long int, long int);
```

which means that it expects two long (4-byte) integer arguments and returns a double (8-byte) floating point result. Notice the correspondence between the C declaration and the right argument of `⎕NA`:

```
C:             double      divide (long int, long int);

APL: 'div' ⎕NA 'F8   math|divide       I4        I4 '
```

It is imperative that care be taken when coding type declarations. A DLL *cannot* check types of data passed from APL. A wrong type declaration will lead to erroneous results or may even cause the workspace to become corrupted and crash.

The full syntax for the right argument of `⎕NA` is:

[result] library|function [arg1] [arg2] ...

Note that functions associated with DLLs are never dyadic. All arguments are passed as items of a (possibly nested) vector on the right of the function.

Locating the DLL

The DLL may be specified using a full pathname, file extension, and function type.

Pathname: APL uses the `LoadLibrary()` system function under Windows and `dlopen()` under UNIX and LINUX to load the DLL. If a full or relative pathname is omitted, these functions search standard operating system directories in a particular order. For further details, see the operating system documentation about these functions.

Alternatively, a full or relative pathname may be supplied in the usual way:

```
      ⎕NA'... c:\mydir\mydll|foo ...'
```

Errors: If the specified DLL (or a dependent DLL) fails to load it will generate:

```
FILE ERROR 1 No such file or directory
```

If the DLL loads successfully, but the specified library function is not accessible, it will generate:

```
VALUE ERROR
```

File Extension: If the file extension is omitted, **.dll** is assumed. Note that some DLLs are in fact **.exe** files, and in this case the extension must be specified explicitly:

```
      ⎕NA'... mydll.exe|foo ...'
```

Function Type: On a Windows computer, two distinct conventions, namely ‘C’ and ‘Pascal’ are in use for passing of arguments and receipt of results. If the type of the function you are calling differs from the default, for your version of Dyalog APL (see below) you must specify the function type explicitly immediately following the DLL name. Combinations are

`.C32`	32 bit, C calling convention	(the default).
`.P32`	32 bit, Pascal calling convention	

Example

```
      ⎕NA'... mydll.exe.P32|foo ...'⍝ 32 bit Pascal
```

Call by Ordinal Number

A DLL may associate an *ordinal number* with any of its functions. This number may then be used to call the function as an alternative to calling it by name. Using `⎕NA` to call by ordinal number uses the same syntax but with the function name replaced with its ordinal number. For example:

```
      ⎕NA'... mydll|57 ...'
```

Multi-Threading

Appending the ‘`&`’ character to the function name causes the external function to be run in its own system thread. For example:

```
      ⎕NA'... mydll|foo& ...'
```

This means that other APL threads can run concurrently with the one that is calling the `⎕NA` function.

Data Type Coding Scheme

The type coding scheme introduced above is of the form:

[direction] [special] type [width] [array]

The options are summarised in the following table and their functions detailed below.

Description	**Symbol**	**Meaning**
Direction	`<`	Pointer to array *input* to DLL function.
	`>`	Pointer to array *output* from DLL function
	`=`	Pointer to input/output array.
Special	`0`	Null-terminated string.
	`#`	Byte-counted string
Type	`I`	int
	`U`	unsigned int
	`C`	char
	`T`	Classic Edition char: translated to/from ANSI Unicode Edition char
	`F`	float
	`A`	APL array
Width	`1`	1-byte
	`2`	2-byte
	`4`	4-byte
	`8`	8-byte
Array	`[n]`	Array of length *n* elements
	`[]`	Array, length determined at call-time
Structure	`{...}`	Structure.

In the Classic Edition, `C` specifies untranslated character, whereas `T` specifies that the character data will be translated to/from `⎕AV`.

In the Unicode Edition, C and T are identical (no translation of character data is performed) except that for C the default width is 1 and for T the default width is "wide" (2 bytes under Windows).

The use of T with default width is recommended to ensure portability between Editions.

Direction

C functions accept data arguments either by *value* or by *address*. This distinction is indicated by the presence of a '`*`' character in the argument declaration:

```
int  num1;         // value of num1 passed.
int *num2;         // Address of num2 passed.
```

An argument (or result) of an external function of type pointer, must be matched in the `⎕NA` call by a declaration starting with one of the characters: `<`, `>`, or `=`.

In C, when an address is passed, the corresponding value can be used as either an *input* or an *output* variable. An output variable means that the C function overwrites values at the supplied address. Because APL is a call-by-value language, and doesn't have pointer types, we accommodate this mechanism by distinguishing output variables, and having them returned explicitly as part of the result of the call.

This means that where the C function indicates a *pointer type*, we must code this as starting with one of the characters: `<`, `>` or `=`.

`<` indicates that the address of the argument will be used by C as an input variable and values at the address will *not* be over-written.

`>` indicates that C will use the address as an output variable. In this case, APL must allocate an output array over which C can write values. After the call, this array will be included in the nested result of the call to the external function.

`=` indicates that C will use the address for both input and output. In this case, APL duplicates the argument array into an output buffer whose address is passed to the external function. As in the case of an output only array, the newly modified copy will be included in the nested result of the call to the external function.

Examples

`<I2`	Pointer to 2-byte integer - *input* to external function
`>C`	Pointer to character *output* from external function.
`=T`	Pointer to character *input* to and *output* from function.
`=A`	Pointer to APL array *modified* by function.

Special

In C it is common to represent character strings as *null-terminated* or *byte counted* arrays. These special data types are indicated by inserting the symbol `0` (null-terminated) or `#` (byte counted) between the direction indicator (`<`, `>`, `=`) and the type (`T` or `C`) specification. For example, a pointer to a null-terminated input character string is coded as `<0T[]`, and an output one coded as `>0T[]`.

Note that while appending the array specifier '`[]`' is formally correct, because the presence of the special qualifier (`0` or `#`) *implies* an array, the '`[]`' may be omitted: `<0T`, `>0T`, `=#C`, etc.

Note also that the 0 and # specifiers may be used with data of all types and widths. For example, in the Classic Edition, `<0U2` may be useful for dealing with Unicode.

Type

The data type of the argument is represented by one of the symbols `i`, `u`, `c`, `t`, `f`, `a`, which may be specified in lower or upper case:

	Type	**Description**
`I`	Integer	The value is interpreted as a 2s complement signed integer.
`U`	Unsigned integer	The value is interpreted as an unsigned integer.
`C`	Character	The value is interpreted as a character. In the Unicode Edition, the value maps directly onto a Unicode code point. In the Classic Edition, the value is interpreted as an index into `⎕AV`. This means that `⎕AV` *positions* map onto corresponding ANSI *positions*. For example, with `⎕IO=0`: `⎕AV[35] = 's'`, maps to `ANSI[35] = '`
`T`	Translated character	The value is interpreted as a character. In the Unicode Edition, the value maps directly onto a Unicode code point. In the Classic Edition, the value is *translated* using standard Dyalog `⎕AV` to ANSI translation. This means that `⎕AV` *characters* map onto corresponding ANSI *characters.* For example, with `⎕IO=0`: `⎕AV[35] = 's'`, maps to `ANSI[115] = 's'`.
`F`	Float	The value is interpreted as an IEEE floating point number.
`A`	APL array	A pointer to the whole array (including header information) is passed. This type is used to communicate with DLL functions which have been written specifically to work with Dyalog APL. See the *User Guide* section on Writing Auxiliary Processors. Note that type A is always passed as a pointer, so is of the form `<A`, `=A` or `>A`.

Width

The type specifier may be followed by the width of the value in bytes. For example:

`I4`	4-byte signed integer.
`U2`	2-byte unsigned integer.
`F8`	8-byte floating point number.
`F4`	4-byte floating point number.

Type	**Possible values for Width**	**Default value for Width**
`I`	1, 2, 4, 8.	4 for 32-bit DLLs 8 for 64-bit DLLs
`U`	1, 2, 4, 8.	4 for 32-bit DLLs 8 for 32-bit DLLs
`C`	1,2,4	1
`T`	1,2,4	wide character(see below)
`F`	4, 8.	8
`A`	Not applicable.	

In the Unicode Edition, the default width is the width of a *wide character* according to the convention of the host operating system. This translates to T2 under Windows and T4 under Unix or Linux.

Examples

`I2`	16-bit integer
`<I4`	Pointer to input 4-byte integer
`U`	Default width unsigned integer.
`=F4`	Pointer to input/output 4-byte floating point number.

Arrays

Arrays are specified by following the basic data type with `[n]` or `[]`, where `n` indicates the number of elements in the array. In the C declaration, the number of elements in an array may be specified explicitly at compile time, or determined dynamically at runtime. In the latter case, the size of the array is often passed along with the array, in a separate argument. In this case, `n`, the number of elements is omitted from the specification. Note that C deals only in scalars and rank 1 (vector) arrays.

```
int vec[10];              // explicit vector length.
unsigned size, list[];    // undetermined length.
```

could be coded as:

`I[10]`	vector of 10 ints.
`U U[]`	unsigned integer followed by an array of unsigned integers.

Confusion sometimes arises over a difference in the declaration syntax between C and `⎕NA`. In C, an argument declaration may be given to receive a pointer to either a single scalar item, or to the first element of an array. This is because in C, the address of an array is deemed to be the address of its first element.

```
void foo (char *string);

char ch = 'a', ptr = "abc";

foo(&ch);          // call with address of scalar.
foo(ptr);          // call with address of array.
```

However, from APL's point of view, these two cases are distinct and if the function is to be called with the address of (pointer to) a *scalar*, it must be declared: `'<T'`. Otherwise, to be called with the address of an *array*, it must be declared: `'<T[]'`. Note that it is perfectly acceptable in such circumstances to define more than one name association to the same DLL function specifying different argument types:

```
      'FooScalar'⎕NA'mydll|foo <T'   ⋄ FooScalar'a'
      'FooVector'⎕NA'mydll|foo <T[]' ⋄ FooVector'abc'
```

Structures

Arbitrary data structures, which are akin to nested arrays, are specified using the symbols `{}`. For example, the code `{F8 I2}` indicates a structure comprised of an 8-byte *float* followed by a 2-byte *int*. Furthermore, the code `<{F8 I2}[3]` means an input pointer to an array of 3 such structures.

For example, this structure might be defined in C thus:

```
typedef struct
{
      double  f;
      short   i;
} mystruct;
```

A function defined to receive a count followed by an *input* pointer to an array of such structures:

```
void foo(unsigned count, mystruct *str);
```

An appropriate `⎕NA` declaration would be:

```
      ⎕NA'mydll.foo U <{F8 I2}[]'
```

A call on the function with two arguments - a count followed by a vector of structures:

```
      foo 4,⊂(1.4 3)(5.9 1)(6.5 2)(0 0)
```

Notice that for the above call, APL converts the two Boolean `(0 0)` elements to an 8-byte float and a 2-byte int, respectively.

Specifying Pointers Explicitly

⎕NA syntax enables APL to pass arguments to DLL functions by *value* or *address* as appropriate. For example if a function requires an integer followed by a *pointer* to an integer:

```
void fun(int valu, int *addr);
```

You might declare and call it:

```
      ⎕NA'mydll|fun I <I' ⋄ fun 42 42
```

The interpreter passes the *value* of the first argument and the *address* of the second one.

Two common cases occur where it is necessary to pass a pointer explicitly. The first is if the DLL function requires a *null pointer*, and the second is where you want to pass on a pointer which itself is a result from a DLL function.

In both cases, the pointer argument should be coded as `I4`. This causes APL to pass the pointer unchanged, *by value*, to the DLL function.

In the previous example, to pass a null pointer, (or one returned from another DLL function), you must code a separate ⎕NA definition.

```
      'fun_null'⎕NA'mydll|fun I I4' ⋄ fun_null 42 0
```

Now APL passes the *value* of the second argument (in this case 0 - the null pointer), rather than its address.

Using a Function

A DLL function may or may not return a result, and may take zero or more arguments. This syntax is reflected in the coding of the right argument of ⎕NA. Notice that the corresponding associated APL function is niladic or monadic (never dyadic), and that it *always* returns a vector result - a null one if there is no output from the function. See Result Vector section below. Examples of the various combinations are:

DLL function Non-result-returning:

```
⎕NA    'mydll|fn1'          ⍝ Niladic
⎕NA    'mydll|fn2 <0T'      ⍝ Monadic - 1-element arg
⎕NA    'mydll|fn3 =0T <0T'  ⍝ Monadic - 2-element arg
```

DLL function Result-returning:

```
⎕NA 'I4 mydll|fn4'          ⍝ Niladic
⎕NA 'I4 mydll|fn5 F8'       ⍝ Monadic - 1-element arg
⎕NA 'I4 mydll|fn6 >I4[] <0T'⍝ Monadic - 2-element arg
```

When the external function is called, the number of elements in the argument must match the number defined in the ⎕NA definition. Using the example functions defined above:

```
      fn1                   ⍝ Niladic Function.
      fn2 ⊂'Single String'  ⍝ 1-element arg
      fn3 'This' 'That'     ⍝ 2-element arg
```

Note in the second example, that you must enclose the argument string to produce a single item (nested) array in order to match the declaration. Dyalog converts the type of a numeric argument if necessary, so for example in fn5 defined above, a Boolean value would be converted to double floating point (F8) prior to being passed to the DLL function.

Pointer Arguments

When passing pointer arguments there are three cases to consider.

< **Input pointer:** In this case you must supply the data array itself as argument to the function. A pointer to its first element is then passed to the DLL function.

```
      fn2 ⊂'hello'
```

> **Output pointer:** Here, you must supply the **number of elements** that the output will need in order for APL to allocate memory to accommodate the resulting array.

```
      fn6 10 'world'  ⍝ 1st arg needs space for 10 ints.
```

Note that if you were to reserve fewer elements than the DLL function actually used, the DLL function would write beyond the end of the reserved array and may cause the interpreter to crash with a System Error.

= **Input/Output:** As with the input-only case, a pointer to the first element of the argument is passed to the DLL function. The DLL function then overwrites some or all of the elements of the array, and the new value is passed back as part of the result of the call. As with the output pointer case, if the input array were too short, so that the DLL wrote beyond the end of the array, the interpreter would almost certainly crash.

```
      fn3 '.....' 'hello'
```

Result Vector

In APL, a function cannot overwrite its arguments. This means that any output from a DLL function must be returned as part of the explicit result, and this includes output via 'output' or 'input/output' pointer arguments.

The general form of the result from calling a DLL function is a nested vector. The first item of the result is the defined explicit result of the external function, and subsequent items are implicit results from output, or input/output pointer arguments.

The length of the result vector is therefore: 1 (if the function was declared to return an explicit result) + the number of output or input/output arguments.

⎕NA Declaration	Result	Output Arguments	Result Length
`   mydll|fn1`	0		0
`   mydll|fn2 <0T`	0	0	0
`   mydll|fn3 =0T <0T`	0	1 0	1
`I4 mydll|fn4`	1		1
`I4 mydll|fn5 F8`	1	0	1
`I4 mydll|fn6 >I4[] <0T`	1	1 0	2

As a convenience, if the result would otherwise be a 1-item vector, it is disclosed. Using the third example above:

```
      ⍴fn3 '.....' 'abc'
5
```

`fn3` has no explicit result; its first argument is input/output pointer; and its second argument is input pointer. Therefore as the length of the result would be 1, it has been disclosed.

ANSI /Unicode Versions of Library Calls

Under Windows, most library functions that take character arguments, or return character results have two forms: one Unicode (Wide) and one ANSI. For example, a function such as `MessageBox()`, has two forms `MessageBoxA()` and `MessageBoxW()`. The `A` stands for ANSI (1-byte) characters, and the `W` for wide (2-byte Unicode) characters.

It is essential that you associate the form of the library function that is appropriate for the Dyalog Edition you are using, i.e. `MessageBoxA()` for the Classic Edition, but `MessageBoxW()` for the Unicode Edition.

To simplify writing portable code for both Editions, you may specify the character `*` instead of `A` or `W` at the end of a function name. This will be replaced by `A` in the Classic Edition and `W` in the Unicode Edition.

The default name of the associated function (if no left argument is given to `⎕NA`), will be without the trailing letter (`MessageBox`).

Type Definitions (typedefs)

The C language encourages the assignment of defined names to primitive and complex data types using its `#define` and `typedef` mechanisms. Using such abstractions enables the C programmer to write code that will be portable across many operating systems and hardware platforms.

Windows software uses many such names and Microsoft documentation will normally refer to the type of function arguments using defined names such as `HANDLE` or `LPSTR` rather than their equivalent C primitive types: `int` or `char*`.

It is beyond the scope of this manual to list *all* the Microsoft definitions and their C primitive equivalents, and indeed, DLLs from sources other than Microsoft may well employ their own distinct naming conventions.

In general, you should consult the documentation that accompanies the DLL in order to convert typedefs to primitive C types and thence to `⎕NA` declarations. The documentation may well refer you to the ‘include’ files which are part of the Software Development Kit, and in which the types are defined.

The following table of some commonly encountered Windows typedefs and their `⎕NA` equivalents might prove useful.

Windows typedef	**⎕NA equivalent**
HWND	I
HANDLE	I
GLOBALHANDLE	I
LOCALHANDLE	I
DWORD	U4
WORD	U2
BYTE	U1
LPSTR	=0T[] (note 1)
LPCSTR	<0T[] (note 2)
WPARAM	U
LPARAM	U4
LRESULT	I4
BOOL	I
UINT	U
ULONG	U4
ATOM	U2
HDC	I
HBITMAP	I
HBRUSH	I
HFONT	I
HICON	I
HMENU	I
HPALETTE	I
HMETAFILE	I
HMODULE	I
HINSTANCE	I
COLORREF	{U1[4]}
POINT	{I I}
POINTS	{I2 I2}
RECT	{I I I I}
CHAR	T or C

Notes

1. LPSTR is a pointer to a null-terminated string. The definition does not indicate whether this is input or output, so the safest coding would be =0T[] (providing the vector you supply for input is long enough to accommodate the result). You may be able to improve simplicity or performance if the documentation indicates that the pointer is 'input only' (<0T[]) or 'output only' (>0T[]). See **Direction** above.

2. LPCSTR is a pointer to a *constant* null-terminated string and therefore coding <0T[] is safe.

3. Note that the use of type T with default width ensures portability of code between Classic and Unicode Editions. In the Classic Edition, T (with no width specifier) implies 1-byte characters which are translated between ⎕AV and ASCII, while In the Unicode Edition, T (with no width specifier) implies 2-byte (Unicode) characters.

Dyalog32.dll

Included with Dyalog APL is a utility DLL which is called dyalog32.dll.

The DLL contains two functions: MEMCPY and STRNCPY.

MEMCPY

MEMCPY is an extremely versatile function used for moving arbitrary data between memory buffers.
Its C definition is:

```
void MEMCPY(                // copy memory
      void *to,             // target address
      void *fm,             // source address
      unsigned size         // number of bytes to copy
);
```

MEMCPY copies size bytes starting from source address fm, to destination address to. If the source and destination areas overlap, the result is undefined.

MEMCPY's versatility stems from being able to associate to it using many different type declarations.

Example

Suppose a global buffer (at address: `addr`) contains (`numb`) double floating point numbers. To copy these to an APL array, we could define the association:

```
      'doubles' ⎕NA 'dyalog32|MEMCPY >F8[] I4 U4'
      doubles numb addr (numb×8)
```

Notice that:

As the first argument to `doubles` is an output argument, we must supply the number of elements to reserve for the output data.

`MEMCPY` is defined to take the number of *bytes* to copy, so we must multiply the number of elements by the element size in bytes.

Example

Suppose that a database application requires that we construct a record in global memory prior to writing it to file. The record structure might look like this:

```
typedef struct {
      int empno;         // employee number.
      float salary;      // salary.
      char name[20];     // name.
} person;
```

Then, having previously allocated memory (`addr`) to receive the record, we can define:

```
      'prec' ⎕NA 'dyalog32|MEMCPY I4 <{I4 F4 T[20]} U4'
      prec addr(99 12345.60 'Charlie Brown
')(4+4+20)
```

STRNCPY

`STRNCPY` is used to copy null-terminated strings between memory buffers.
Its C definition is:

```
void STRNCPY(            // copy null-terminated string
      char *to,          // target address
      char *fm,          // source address
      unsigned size      // MAX number of chars to copy
);
```

`STRNCPY` copies a maximum of `size` characters from the null-terminated source string at address `fm`, to the destination address `to`. If the source and destination strings overlap, the result is undefined.

If the source string is shorter than `size`, null characters are appended to the destination string.

If the source string (including its terminating null) is longer than `size`, only `size` characters are copied and the resulting destination string is not null-terminated

Example

Suppose that a database application returns a pointer (`addr`) to a structure that contains two pointers to (max 20-char) null-terminated strings.

```
typedef struct {     // null-terminated strings:
      char *first;  // first name (max 19 chars + 1 null).
      char *last;   // last name. (max 19 chars + 1 null).
} name;
```

To copy the names *from* the structure:

```
      'get'⎕NA'dyalog32|STRNCPY >0T[] I4 U4'
      get 20 addr 20
Charlie
      get 20 (addr+4) 20
Brown
```

To copy data *from* the workspace *into* an already allocated (`new`) structure:

```
      'put'⎕NA'dyalog32|STRNCPY I4 <0T[] U4'
      put new 'Bo' 20
      put (new+4) 'Peep' 20
```

Notice in this example that you must ensure that names no longer than 19 characters are passed to `put`. More than 19 characters would not leave STRNCPY enough space to include the trailing null, which would probably cause the application to fail.

Examples

The following examples all use functions from the Microsoft Windows USER32.DLL.

This DLL should be located in a standard Windows directory, so you should not normally need to give the full path name of the library. However if trying these examples results in the error message 'FILE ERROR 1 No such file or directory', you must locate the DLL and supply the full path name (and possibly extension).

Example 1

The Windows function "`GetCaretBlinkTime`" retrieves the caret blink rate. It takes no arguments and returns an unsigned *int* and is declared as follows:

```
UINT GetCaretBlinkTime(void);
```

The following statements would provide access to this routine through an APL function of the same name.

```
      ⎕NA 'U User32|GetCaretBlinkTime'
      GetCaretBlinkTime
530

      ⎕NA 'U User32|GetCaretBlinkTime'
```

The following statement would achieve the same thing, but using an APL function called `BLINK`.

```
      'BLINK' ⎕NA 'U User32|GetCaretBlinkTime'
      BLINK
530
```

Example 2

The Windows function "`SetCaretBlinkTime`" sets the caret blink rate. It takes a single unsigned *int* argument, does not return a result and is declared as follows:

```
void SetCaretBlinkTime(UINT);
```

The following statements would provide access to this routine through an APL function of the same name :

```
      ⎕NA 'User32|SetCaretBlinkTime U'
      SetCaretBlinkTime 1000
```

Example 3

The Windows function "`MessageBox`" displays a standard dialog box on the screen and awaits a response from the user. It takes 4 arguments. The first is the window handle for the window that owns the message box. This is declared as an unsigned *int*. The second and third arguments are both pointers to null-terminated strings containing the message to be displayed in the Message Box and the caption to be used in the window title bar. The 4th argument is an unsigned *int* that specifies the Message Box type. The result is an *int* which indicates which of the buttons in the message box the user has pressed. The function is declared as follows:

```
int MessageBox(HWND, LPCSTR, LPCSTR, UINT);
```

The following statements provide access to this routine through an APL function of the same name. Note that the 2nd and 3rd arguments are both coded as input pointers to type T null-terminated character arrays which ensures portability between Editions.

```
      ⎕NA 'I User32|MessageBox* U <0T <0T U'
```

The following statement displays a Message Box with a stop sign icon together with 2 push buttons labelled OK and Cancel (this is specified by the value 19).

```
      MessageBox 0 'Message' 'Title' 19
```

The function works equally well in the Unicode Edition because the <0T specification is portable.

```
      MessageBox 0 'Το Μήνυμα' 'Ο Τίτλος' 19
```

Note that a simpler, portable (and safer) method for displaying a Message Box is to use Dyalog APL’s primitive `MsgBox` object.

Example 4

The Windows function "`FindWindow`" obtains the window handle of a window which has a given character string in its title bar. The function takes two arguments. The first is a pointer to a null-terminated character string that specifies the window's class name. However, if you are not interested in the class name, this argument should be a NULL pointer. The second is a pointer to a character string that specifies the title that identifies the window in question. This is an example of a case described above where two instances of the function must be defined to cater for the two different types of argument. However, in practice this function is most often used without specifying the class name. The function is declared as follows:

```
HWND FindWindow(LPCSTR, LPCSTR);
```

The following statement associates the APL function `FW` with the second variant of the FindWindow call, where the class name is specified as a NULL pointer. To indicate that APL is to pass the *value* of the NULL pointer, rather than its address, we need to code this argument as `I4`.

```
      'FW' ⎕NA 'U User32|FindWindow* I4 <0T'
```

To obtain the handle of the window entitled "CLEAR WS - Dyalog APL/W":

```
      ⎕←HNDL←FW 0 'CLEAR WS - Dyalog APL/W'
59245156
```

Example 5

The Windows function "`GetWindowText`" retrieves the caption displayed in a window's title bar. It takes 3 arguments. The first is an unsigned *int* containing the window handle. The second is a pointer to a buffer to receive the caption as a null-terminated character string. This is an example of an output array. The third argument is an *int* which specifies the maximum number of characters to be copied into the output buffer. The function returns an *int* containing the actual number of characters copied into the buffer and is declared as follows:

```
int GetWindowText(HWND, LPSTR, int);
```

The following associates the "`GetWindowText`" DLL function with an APL function of the same name. Note that the second argument is coded as "`>0T`" indicating that it is a pointer to a character output array.

```
      ⎕NA 'I User32|GetWindowText* U >0T I'
```

Now change the Session caption using `)WSID` :

```
      )WSID MYWS
was CLEAR WS
```

Then retrieve the new caption (max length 255) using window handle `HNDL` from the previous example:

```
      DISPLAY GetWindowText HNDL 255 255
.→-------------------------.
|    .→-------------------. |
| 19 |MYWS - Dyalog APL/W| |
|    '-------------------' |
'∊-------------------------'
```

There are three points to note. Firstly, the number 255 is supplied as the second argument. This instructs APL to allocate a buffer large enough for a 255-element character vector into which the DLL routine will write. Secondly, the result of the APL function is a nested vector of 2 elements. The first element is the result of the DLL function. The second element is the output character array.

Finally, notice that although we reserved space for 255 elements, the result reflects the length of the actual text (19).

An alternative way of coding and using this function is to treat the second argument as an input/output array.

e.g.

```
      ⎕NA 'I User32|GetWindowText* U =0T I'

      DISPLAY GetWindowText HNDL (255⍴' ') 255
.→-------------------------.
|    .→------------------. |
| 19 |MYWS - Dyalog APL/W| |
|    '-------------------' |
'∊-------------------------'
```

In this case, the second argument is coded as `=0T`, so when the function is called an array of the appropriate size must be supplied. This method uses more space in the workspace, although for small arrays (as in this case) the real impact of doing so is negligible.

Example 6

The function "`GetCharWidth`" returns the width of each character in a given range Its first argument is a device context (handle). Its second and third arguments specify font positions (start and end). The third argument is the resulting integer vector that contains the character widths (this is an example of an output array). The function returns a Boolean value to indicate success or failure. The function is defined as follows. Note that this function is provided in the library: GDI32.DLL.

```
BOOL GetCharWidth(HDC, UINT, UINT, int FAR*);
```

The following statements provide access to this routine through an APL function of the same name :

```
      ⎕NA 'U4 GDI32|GetCharWidth* I U U >I[]'

     'P'⎕WC'Printer'

      DISPLAY GetCharWidth ('P' ⎕WG 'Handle') 65 67 3
.→-------------.
|    .→-------. |
| 1  |50 50 50| |
|    '~-------' |
'∊-------------'
```

Note: `'P'⎕WG'Handle'` returns a 32-bit handle which, if the *top bit* is set, will appear in APL as a negative integer. Attempting to supply such a negative number as an argument to a DLL function when the argument is declared *unsigned* will result in a `DOMAIN ERROR`. Window handles should therefore be declared as `I` rather than `U`.

Example 7

The following example from the supplied workspace: `QUADNA.DWS` illustrates several techniques which are important in advanced `⎕NA` programming. Function `DllVersion` returns the major and minor version number for a given DLL.

In advanced DLL programming, it is often necessary to administer memory outside APL's workspace. In general, the procedure for such use is:

1. Allocate global memory.
2. Lock the memory.
3. Copy any DLL input information from workspace into memory.
4. Call the DLL function.
5. Copy any DLL output information from memory to workspace.
6. Unlock the memory.
7. Free the memory.

Notice that steps 1 and 7, and steps 2 and 6 complement each other. That is, if you allocate global system memory, you must free it after you have finished using it. If you continue to use global memory without freeing it, your system will gradually run out of resources. Similarly, if you lock memory (which you must do before using it), then you should unlock it before freeing it. Although on some versions of Windows, freeing the memory will include unlocking it, in the interests of good style, maintaining the symmetry is probably a good thing.

```
    ∇ version←DllVersion file;Alloc;Free;Lock;Unlock;Size
                  ;Info;Value;Copy;size;hndl;addr;buff;ok
[1]
[2]  'Alloc'⎕NA'U4 kernel32|GlobalAlloc U4 U4'
[3]   'Free'⎕NA'U4 kernel32|GlobalFree U4'
[4]   'Lock'⎕NA'U4 kernel32|GlobalLock U4'
[5] 'Unlock'⎕NA'U4 kernel32|GlobalUnlock U4'
[6]
[7]  'Size'⎕NA'U4 version|GetFileVersionInfoSize* <0T >U4'
[8]  'Info'⎕NA'U4 version|GetFileVersionInfo*<0T U4 U4 U4'
[9] 'Value'⎕NA'U4 version|VerQueryValue* U4 <0T >U4 >U4'
[10]
[11] 'Copy'⎕NA'dyalog32|MEMCPY >U4[] U4 U4'
[12]
[13]  :If ×size←⊃Size file 0             ⍝ Size of info
[14]  :AndIf ×hndl←Alloc 0 size          ⍝ Alloc memory
[15]    :If ×addr←Lock hndl              ⍝ Lock memory
[16]      :If ×Info file 0 size addr     ⍝ Version info
[17]        ok buff size←Value addr'\' 0 0 ⍝ Version value
[18]        :If ok
[19]          buff←Copy(size÷4)buff size   ⍝ Copy info
[20]          version←(2/2*16)⊤⊃2↓buff     ⍝ Split version
[21]        :EndIf
[22]      :EndIf
[23]      ok←Unlock hndl                 ⍝ Unlock memory
[24]    :EndIf
[25]    ok←Free hndl                     ⍝ Free memory
[26]  :EndIf
    ∇
```

Lines [2-11] associate APL function names with the DLL functions that will be used.

Lines [2-5] associate functions to administer global memory.
Lines [7-9] associate functions to extract version information from a DLL.
Line[11] associates `Copy` with `MEMCPY` function from **dyalog32.dll**.

Lines [13-26] call the DLL functions.

Line [13] requests the size of buffer required to receive version information for the DLL. A size of 0 will be returned if the DLL does not contain version information.

Notice that care is taken to balance memory allocation and release:

On line [14], the :If clause is taken only if the global memory allocation is successful, in which case (and only then) a corresponding Free is called on line [25].

Unlock on line[23] is called if and only if the call to Lock on line [15] succeeds.

A result is returned from the function *only* if all the calls are successful Otherwise, the calling environment will sustain a `VALUE ERROR`.

More Examples

```
⎕NA'U4 ADVAPI32 |RegCloseKey       U4'
⎕NA'I  ADVAPI32 |RegCreateKeyEx*   U <0T I <0T I I I >U >U'
⎕NA'U  ADVAPI32 |RegEnumValue*     U U >0T =U U >U >0T =U'
⎕NA'I  ADVAPI32 |RegOpenKey*       U <0T >U'
⎕NA'U4 ADVAPI32 |RegOpenKeyEx*     U4 <0T U4 U4 =U4'
⎕NA'I  ADVAPI32 |RegQueryValueEx*  U <0T U4 >U4 >0T =I4'
⎕NA'I4 ADVAPI32 |RegSetValueEx*    U <0T U4 U4 <0T U4'
⎕NA'I  DOS_U32  |Copy              <0T <0T'
⎕NA'U  DOS_U32  |Dir               <0T U >0T'
⎕NA'U  DOS_U32  |DirMore           U >0T'
⎕NA'   DOS_U32  |DirClose'
⎕NA'I  DOS_U32  |Rename            <0T <0T'
⎕NA'I  DOS_U32  |Erase             <0T'
⎕NA'   dyalog32 |STRNCPY           >0T I4 U4'
⎕NA'   dyalog32 |MEMCPY            >{U1[4]}[16] I4 U4'
⎕NA'I  gdi32    |AddFontResource*  <0T'
⎕NA'   GDI32    |BitBlt            U U U U U U U U U'
⎕NA'U4 gdi32    |GetPixel          U4 U4 U4'
⎕NA'U4 GDI32    |GetStockObject    U4'
⎕NA'I  gdi32    |RemoveFontResource* <0T'
⎕NA'U4 gdi32    |SetPixel          U4 U4 U4 U4'
⎕NA'U4 GLU32    |gluPerspective    F8 F8 F8 F8'
⎕NA'I  kernel32 |_lclose           U'
⎕NA'I  kernel32 |_lcreat           <0T I'
⎕NA'U4 kernel32 |_llseek           I U4 I'
⎕NA'I  kernel32 |_lopen            <0T I'
⎕NA'U  kernel32 |_lread            U >U1[] U'
⎕NA'U  kernel32 |_lwrite           U <U1[] U'
⎕NA'I  kernel32 |CopyFileA         <0T <0T I'
⎕NA'I4 kernel32 |GetEnvironmentStrings'
⎕NA'I4 kernel32 |GetLastError'
⎕NA'I  kernel32 |GetPrivateProfileInt* <0T <0T I <0T'
⎕NA'I  kernel32 |GetProfileString* <0T <0T <0T >0T I'
⎕NA'U2 kernel32 |GetSystemDirectory* >0T U2'
⎕NA'I2 KERNEL32 |GetTempPathA      U4 >0T'
⎕NA'U  KERNEL32 |GetWindowsDirectory* >0T U'
⎕NA'U  kernel32 |GlobalAlloc       U U4'
⎕NA'U  kernel32 |GlobalFree        U'
⎕NA'   Kernel32 |GlobalMemoryStatus ={U4 U4 U4 U4 U4 U4 U4 U4}'
⎕NA'U  KERNEL32 |WritePrivateProfileString* <0T <0T <0T <0T'
⎕NA'U4 OpenGL32 |glClearColor      F4 F4 F4 F4'
⎕NA'U4 OpenGL32 |glClearDepth      F4'
⎕NA'U4 OpenGL32 |glEnable          U4'
⎕NA'U4 OpenGL32 |glMatrixMode      U4'
⎕NA'U4 USER32   |ClientToScreen    U =U4[2]'
⎕NA'I  user32   |FindWindow*       I4 <0T'
⎕NA'   user32   |ShowWindow        I I'
⎕NA'I2 USER32   |GetAsyncKeyState  I'
⎕NA'U4 user32   |GetDC             U4'
⎕NA'I4 User32   |GetDialogBaseUnits'
⎕NA'I  user32   |GetFocus'
⎕NA'U4 user32   |GetSysColor       I'
⎕NA'I  user32   |GetSystemMetrics  I'
⎕NA'   USER32   |InvalidateRgn     I4 U4 U4'
⎕NA'I  user32   |MessageBox*       I <0T <0T I'
⎕NA'U4 user32   |ReleaseDC         U4 U4'
⎕NA'U4 USER32   |SendMessage*      I4 U4 U4 <I[]'
⎕NA'I2 user32   |SetFocus          I'
⎕NA'I  user32   |WinHelp*          I <0T I I4'
⎕NA'U  WINMM    |sndPlaySound*     <0T U'
```

Native File Append: {R}←X ⎕NAPPEND Y

This function appends the ravel of its left argument X to the end of the designated native file. X must be a simple homogeneous APL array. Y is a 1- or 2-element integer vector. Y[1] is a negative integer that specifies the tie number of a native file. The optional second element Y[2] specifies the data type to which the array X is to be converted before it is written to the file.

The shy file index result returned is the position within the file of the end of the record, which is also the start of the following one.

Unicode Edition

Unless you specify the data type in Y[2], a character array will by default be written using type 80.

If the data will not fit into the specified character width (bytes) ⎕NAPPEND will fail with a DOMAIN ERROR.

As a consequence of these two rules, you must specify the data type (either 160 or 320) in order to write Unicode characters whose code-point are in the range 256-65535 and >65535 respectively.

Example

```
      n←'test'⎕NCREATE 0

      'abc' ⎕nappend n

      'ταβέρνα'⎕nappend n
DOMAIN ERROR
      'ταβέρνα'⎕NAPPEND n
     ^

      'ταβέρνα'⎕NAPPEND n 160

      ⎕NREAD n 80 3 0
abc
      ⎕NREAD n 160 7
ταβέρνα
```

For compatibility with old files, you may specify that the data be converted to type 82 on output. The conversion (to ⎕AV indices) will be determined by the local value of ⎕AVU.

Name Classification: R←⎕NC Y

Y must be a simple character scalar, vector, matrix ,or vector of vectors that specifies a list of names. R is a simple numeric vector containing one element per name in Y.

Each element of R is the name class of the active referent to the object named in Y.

If Y is *simple*, a name class may be:

Name Class	**Description**
¯1	invalid name
0	unused name
1	Label
2	Variable
3	Function
4	Operator
9	Object (GUI, namespace, COM, .Net)

If Y is nested, a more precise analysis of name class is obtained whereby different types of functions (primitive, traditional defined functions, D-fns) are identified by a decimal extension. For example, defined functions have name class 3.1, D-fns have name class 3.2, and so forth. The complete set of name classification is as follows:

	Array (2)	**Functions (3)**	**Operators (4)**	**Namespaces (9)**
n.1	Variable	Traditional	Traditional	Created by ⎕NS
n.2	Field	D-fns	D-ops	Instance
n.3	Property	Derived Primitive		
n.4				Class
n.5		N/A		Interface
n.6	External Shared	External		External Class
n.7				External Interface

In addition, values in R are negative to identify names of methods, properties and events that are inherited through the *class hierarchy* of the current class or instance.

Variable (Name-Class 2.1)

Conventional APL arrays have name-class 2.1.

```
      NUM←88
      CHAR←'Hello World'

      ⎕NC ↑'NUM' 'CHAR'
2 2

      ⎕NC 'NUM' 'CHAR'
2.1 2.1

      'MYSPACE'⎕NS ''
      MYSPACE.VAR←10
      MYSPACE.⎕NC'VAR'
2
      MYSPACE.⎕NC⊂'VAR'
2.1
```

Field (Name-Class 2.2)

Fields defined by APL Classes have name-class 2.2.

```
:Class nctest
    :Field Public pubFld
    :Field pvtFld

    ∇ r←NC x
      :Access Public
      r←⎕NC x
    ∇
...
:EndClass ⍝ nctest

      ncinst←⎕NEW nctest
```

The name-class of a Field, whether Public or Private, *viewed from a Method that is executing within the Instance Space,* is 2.2.

```
      ncinst.NC'pubFld' 'pvtFld'
2.2 2.2
```

Note that an internal Method sees both Public and Private Fields in the Class Instance. However, when viewed from *outside* the instance, only public fields are visible

```
      ⎕NC 'ncinst.pubFld' 'ncinst.pvtFld'
¯2.2 0
```

In this case, the name-class is negative to indicate that the name has been exposed by the class hierarchy, rather than existing in the associated namespace which APL has created to contain the instance. The same result is returned if ⎕NC is executed inside this space:

```
      ncinst.⎕NC'pubFld' 'pvtFld'
¯2.2 0
```

Note that the names of Fields are reported as being *unused* if the argument to ⎕NC is simple.

```
      ncinst.⎕NC 2 6⍴'pubFldpvtFld'
0 0
```

Property (Name-Class 2.3)

Properties defined by APL Classes have name-class 2.3.

```
:Class nctest
    :Field pvtFld←99

    :Property pubProp
    :Access Public
        ∇ r←get
          r←pvtFld
        ∇
    :EndProperty

    :Property pvtProp
        ∇ r←get
          r←pvtFld
        ∇
    :EndProperty

    ∇ r←NC x
      :Access Public
      r←⎕NC x
    ∇
...
:EndClass ⍝ nctest

      ncinst←⎕NEW nctest
```

The name-class of a Property, whether Public or Private, *viewed from a Method that is executing within the Instance Space,* is 2.3.

```
      ncinst.NC'pubProp' 'pvtProp'
2.3 2.3
```

Note that an internal Method sees both Public and Private Properties in the Class Instance. However, when viewed from *outside* the instance, only Public Properties are visible

```
      ⎕NC 'ncinst.pubProp' 'ncinst.pvtProp'
¯2.3 0
```

In this case, the name-class is negative to indicate that the name has been exposed by the class hierarchy, rather than existing in the associated namespace which APL has created to contain the instance. The same result is returned if `⎕NC` is executed inside this space:

```
      ncinst.⎕NC 'pubProp' 'pvtProp'
¯2.3 0
```

Note that the names of Properties are reported as being *unused* if the argument to `⎕NC` is simple.

```
      ncinst.⎕NC 2 6⍴'pubProppvtProp'
0 0
```

External Properties (Name-Class 2.6)

Properties exposed by external objects (.Net and COM and the APL GUI) have name-class `¯2.6`.

```
      ⎕USING←'System'
      dt←⎕NEW DateTime (2006 1 1)
      dt.⎕NC 'Day' 'Month' 'Year'
¯2.6 ¯2.6 ¯2.6

      'ex' ⎕WC 'OLEClient' 'Excel.Application'
      ex.⎕NC 'Caption' 'Version' 'Visible'
¯2.6 ¯2.6 ¯2.6

     'f'⎕WC'Form'
      f.⎕NC'Caption' 'Size'
¯2.6 ¯2.6
```

Note that the names of such Properties are reported as being *unused* if the argument to `⎕NC` is simple.

```
      f.⎕NC 2 7⍴'CaptionSize   '
0 0
```

Defined Functions (Name-Class 3.1)

Traditional APL defined functions have name-class 3.1.

```
      ∇ R←AVG X
[1]     R←(+/X)÷⍴X
      ∇
       AVG ⍳100
50.5

      ⎕NC'AVG'
3
      ⎕NC⊂'AVG'
3.1

      'MYSPACE'⎕NS 'AVG'
       MYSPACE.AVG ⍳100
50.5

      MYSPACE.⎕NC'AVG'
3
      ⎕NC⊂'MYSPACE.AVG'
3.1
```

Note that a function that is simply cloned from a defined function by assignment retains its name-class.

```
      MEAN←AVG
      ⎕NC'AVG' 'MEAN'
3.1 3.1
```

Whereas, the name of a function that amalgamates a defined function with any other functions has the name-class of a Derived Function, i.e. 3.3.

```
      VMEAN←AVG∘,
      ⎕NC'AVG' 'VMEAN'
3.1 3.3
```

D-Fns (Name-Class 3.2)

D-Fns (Dynamic Functions) have name-class 3.2

```
      Avg←{(+/⍵)÷⍴⍵}

      ⎕NC'Avg'
3
      ⎕NC⊂'Avg'
3.2
```

Derived Functions (Name-Class 3.3)

Derived Functions and functions created by naming a Primitive function have name-class 3.3.

```
      PLUS←+
      SUM←+/
      CUM←PLUS\
      ⎕NC'PLUS' 'SUM' 'CUM'
3.3 3.3 3.3
      ⎕NC 3 4⍴'PLUSSUM CUM '
3 3 3
```

Note that a function that is simply cloned from a defined function by assignment retains its name-class. Whereas, the name of a function that amalgamates a defined function with any other functions has the name-class of a Derived Function, i.e. 3.3.

```
     ∇ R←AVG X
[1]    R←(+/X)÷⍴X
     ∇

      MEAN←AVG
      VMEAN←AVG∘,
      ⎕NC'AVG' 'MEAN' 'VMEAN'
3.1 3.1 3.3
```

External Functions (Name-Class 3.6)

Methods exposed by the Dyalog APL GUI and COM and .NET objects have name-class `¯3.6`. Methods exposed by External Functions created using `⎕NA` and `⎕SH` all have name-class `3.6`.

```
      'F'⎕WC'Form'

      F.⎕NC'GetTextSize' 'GetFocus'
¯3.6 ¯3.6

      'EX'⎕WC'OLEClient' 'Excel.Application'
      EX.⎕NC 'Wait' 'Save' 'Quit'
¯3.6 ¯3.6 ¯3.6

      ⎕USING←'System'
      dt←⎕NEW DateTime (2006 1 1)
      dt.⎕NC 'AddDays' 'AddHours'
¯3.6 ¯3.6

       'beep'⎕NA'user32|MessageBeep i'

      ⎕NC'beep'
3
      ⎕NC⊂'beep'
3.6
      'xutils'⎕SH''
      )FNS
avx     box     dbr     getenv  hex     ltom    ltov
mtol    ss      vtol
      ⎕NC'hex' 'ss'
3.6 3.6
```

Operators (Name-Class 4.1)

Traditional Defined Operators have name-class 4.1.

```
      ∇FILTER∇
     ∇ VEC←(P FILTER)VEC  ⍝ Select from VEC those elts ..
[1]    VEC←(P¨VEC)/VEC    ⍝ for which BOOL fn P is true.
     ∇

      ⎕NC'FILTER'
4
      ⎕NC⊂'FILTER'
4.1
```

D-Ops (Name-Class 4.2

D-Ops (Dynamic Operators) have name-class 4.2.

```
pred←{⎕IO ⎕ML←1 3   ⍝ Partitioned reduction.
     ⊃⍺⍺/¨(⍺/⍳⍴⍺)⊂⍵
     }

      2 3 3 2 +pred ⍳10
3 12 21 19

      ⎕NC'pred'
4
      ⎕NC⊂'pred'
4.2
```

External Events (Name-Class 8.6)

Events exposed by Dyalog APL GUI objects, COM and .NET objects have name-class `¯8.6`.

```
      f←⎕NEW'Form'('Caption' 'Dyalog GUI Form')

      f.⎕NC'Close' 'Configure' 'MouseDown'
¯8.6 ¯8.6 ¯8.6

      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')
      xl.⎕NL -8
 NewWorkbook  SheetActivate  SheetBeforeDoubleClick ...

      xl.⎕nc 'SheetActivate' 'SheetCalculate'
¯8.6 ¯8.6
```

```
      ⎕USING←'System.Windows.Forms,system.windows.forms.dll'
      ⎕NC,⊂'Form'
9.6
      Form.⎕NL -8
 Activated  BackgroundImageChanged  BackColorChanged ...
```

Namespaces (Name-Class 9.1)

Plain namespaces created using ⎕NS have name-class 9.1.

```
      'MYSPACE' ⎕NS ''
      ⎕NC'MYSPACE'
9
      ⎕NC⊂'MYSPACE'
9.1
```

Note however that a namespace created by cloning, where the right argument to ⎕NS is a ⎕OR of a namespace, retains the name-class of the original space.

```
      'CopyMYSPACE'⎕NS ⎕OR 'MYSPACE'
      'CopyF'⎕NS ⎕OR 'F'⎕WC'Form'

      ⎕NC'MYSPACE' 'F'
9.1 9.2
      ⎕NC'CopyMYSPACE' 'CopyF'
9.1 9.2
```

The Name-Class of .Net namespaces (visible through ⎕USING) is also 9.1

```
      ⎕USING←''
      ⎕NC 'System' 'System.IO'
9.1 9.1
```

Instances (Name-Class 9.2)

Instances of Classes created using ⎕NEW, and GUI objects created using ⎕WC all have name-class 9.2.

```
      MyInst←⎕NEW MyClass
      ⎕NC'MyInst'
9
      ⎕NC⊂'MyInst'
9.2
      UrInst←⎕NEW ⎕FIX ':Class'  ':EndClass'
      ⎕NC 'MyInst' 'UrInst'
9.2 9.2
```

```
      'F'⎕WC 'Form'
      'F.B' ⎕WC 'Button'
      ⎕NC 2 3⍴'F  F.B'
9 9
      ⎕NC'F' 'F.B'
9.2 9.2
      F.⎕NC'B'
9
      F.⎕NC⊂,'B'
9.2
```

Instances of COM Objects whether created using `⎕WC` or `⎕NEW` also have name-class 9.2.

```
      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')
      'XL'⎕WC'OLEClient' 'Excel.Application'
      ⎕nc'xl' 'XL'
9.2 9.2
```

The same is true of Instances of .Net Classes (Types) whether created using `⎕NEW` or `.New`.

```
      ⎕USING←'System'
      dt←⎕NEW DateTime (3↑⎕TS)
      DT←DateTime.New 3↑⎕TS
      ⎕NC 'dt' 'DT'
9.2 9.2
```

Note that if you remove the GUI component of a GUI object, using the Detach method, it reverts to a plain namespace.

```
      F.Detach
      ⎕NC⊂,'F'
9.1
```

Correspondingly, if you attach a GUI component to a plain namespace using the monadic form of `⎕WC`, it morphs into a GUI object

```
      F.⎕WC 'PropertySheet'
      ⎕NC⊂,'F'
9.2
```

Classes (Name-Class 9.4)

Classes created using the editor or `⎕FIX` have name-class 9.4.

```
      )ED ○MyClass

:Class MyClass
    ∇ r←NC x
      :Access Public Shared
      r←⎕NC x
    ∇
:EndClass ⍝ MyClass

      ⎕NC 'MyClass'
9
      ⎕NC⊂'MyClass'
9.4

      ⎕FIX ':Class UrClass'  ':EndClass'
      ⎕NC 'MyClass' 'UrClass'
9.4 9.4
```

Note that the name of the Class is visible to a Public Method in that Class, or an Instance of that Class.

```
      MyClass.NC'MyClass'
9
      MyClass.NC⊂'MyClass'
9.4
```

Interfaces (Name-Class 9.5)

Interfaces, defined by `:Interface ... :EndInterface` clauses, have name-class 9.5.

```
:Interface IGolfClub
:Property Club
    ∇ r←get
    ∇
    ∇ set
    ∇
:EndProperty

∇ Shank←Swing Params
∇

:EndInterface ⍝ IGolfClub

      ⎕NC 'IGolfClub'
9
      ⎕NC ⊂'IGolfClub'
9.5
```

External Classes (Name-Class 9.6)

External Classes (Types) .exposed by .Net have name-class 9.6.

```
      ⎕USING←'System' 'System.IO'

      ⎕NC 'DateTime' 'File' 'DirectoryInfo'
9.6 9.6 9.6
```

Note that referencing a .Net class (type) with ⎕NC, fixes the name of that class in the workspace and obviates the need for APL to repeat the task of searching for and loading the class when the name is next used.

External Interfaces (Name-Class 9.7)

External Interfaces exposed by .Net have name-class 9.7.

```
      ⎕USING←'System.Web.UI,system.web.dll'

      ⎕NC 'IPostBackDataHandler' 'IPostBackEventHandler'
9.7 9.7
```

Note that referencing a .Net Interface with ⎕NC, fixes the name of that Interface in the workspace and obviates the need for APL to repeat the task of searching for and loading the Interface when the name is next used.

Native File Create: {R}←X ⎕NCREATE Y

This function creates a new file. Under Windows the file is opened in compatibility mode. The name of the new file is specified by the left argument `X` which must be a simple character vector or scalar containing a valid pathname for the file. `Y` is 0 or a negative integer value that specifies an (unused) tie number by which the file may subsequently be referred.

The shy result of `⎕NCREATE` is the tie number of the new file.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

```
      tie←¯1+⌊/0,⎕NNUMS   ⍝ With next available number,
      file ⎕NCREATE tie   ⍝ ... create file.
```

to:

```
      tie←file ⎕NCREATE 0 ⍝ Create with first available
no.
```

Native File Erase: {R}←X ⎕NERASE Y

This function erases (deletes) a native file. `Y` is a negative integer tie number associated with a tied native file. `X` is a simple character vector or scalar containing the name of the same file and must be **identical** to the name used when it was opened by `⎕NCREATE` or `⎕NTIE`.

The shy result of `⎕NERASE` is the tie number that the erased file had.

Example

```
      file ⎕nerase file ⎕ntie 0
```

New Instance: R←⎕NEW Y

⎕NEW creates a new instance of the Class or .Net Type specified by Y.

Y must be a 1- or 2-item scalar or vector. The first item is a reference to a Class or to a .Net Type, or a character vector containing the name of a Dyalog GUI object. The second item, if specified, contains the argument to be supplied to the Class or Type *Constructor*.

The result R is a reference to a new instance of Class or Type Y.

For further information, see *Interface Guide*.

Class Example

```
:Class Animal
    ∇ Name nm
      :Access Public
      :Implements Constructor
      ⎕DF nm
    ∇
:EndClass ⍝ Animal

      Donkey←⎕NEW Animal 'Eeyore'
      Donkey
Eeyore
```

If ⎕NEW is called with just a Class reference (i.e. without parameters for the Constructor), the default constructor will be called. A default constructor is defined by a *niladic* function with the `:Implements Constructor` attribute. For example, the `Animal` Class may be redefined as:

```
:Class Animal
    ∇ NoName
      :Access Public
      :Implements Constructor
      ⎕DF 'Noname'
    ∇
    ∇ Name nm
      :Access Public
      :Implements Constructor
      ⎕DF nm
    ∇
:EndClass ⍝ Animal

      Horse←⎕NEW Animal
      Horse
Noname
```

.Net Examples

```
      ⎕USING←'System' 'System.Web.Mail,System.Web.dll'
      dt←⎕NEW DateTime (2006 1 1)
      msg←⎕NEW MailMessage
      ⎕NC 'dt' 'msg' 'DateTime' 'MailMessage'
9.2 9.2 9.6 9.6
```

Note that **.Net Types** are accessed as follows.

If the name specified by the first item of `Y` would otherwise generate a `VALUE ERROR`, and `⎕USING` has been set, APL attempts to load the Type specified by `Y` from the .Net assemblies (DLLs) specified in `⎕USING.` If successful, the name specified by `Y` is entered into the SYMBOL TABLE with a name-class of `9.6`. Subsequent references to that symbol (in this case `DateTime`) are resolved directly and do not involve any assembly searching.

```
      F←⎕NEW ⊂'Form'
      F←⎕NEW'Form'(('Caption' 'Hello')('Posn' (10 10)))

      ⎕NEW'Form'(('Caption' 'Hello')('Posn' (10 10)))
#.[Form]
```

Name List: R←{X}⎕NL Y

`Y` must be a simple numeric scalar or vector containing one or more of the values for name-class See also the system function `⎕NC`.

`X` is optional. If present, it must be a simple character scalar or vector. `R` is a list of the names of active objects whose name-class is included in `Y` in standard sorted order.

If *any* element of `Y` is negative, positive values in `Y` are treated as if they were negative, and `R` is a vector of character vectors. Otherwise, `R` is simple character matrix.

Furthermore, if `⎕NL` is being evaluated inside the namespace associated with a Class or an Instance of a Class, and any element of `Y` is negative, `R` includes the Public names exposed by the Base Class (if any) and all other Classes in the Class hierarchy.

If `X` is supplied, `R` contains only those names which begin with any character of `X`. Standard sorted order is the collation order of `⎕AV`.

If an element of Y is an integer, the names of all of the corresponding sub-name-classes are included in R. For example, if Y contains the value 2, the names of all variables (name-class 2.1), fields (2.2), properties (2.3) and external or shared variables (2.6) are obtained. Otherwise, only the names of members of the corresponding sub-name-class are obtained.

Examples:

```
      ⎕NL 2 3
A
FAST
FIND
FOO
V

      'AV' ⎕NL 2 3
A
V

      ⎕NL ¯9
 Animal  Bird  BirdBehaviour  Coin  Cylinder
DomesticParrot  Eeyore  FishBehaviour  Nickel  Parrot
Penguin  Polly  Robin
      ⎕NL ¯9.3 ⍝ Instances
 Eeyore  Nickel  Polly  Robin
      ⎕NL ¯9.4 ⍝ Classes
 Animal  Bird  Coin  Cylinder  DomesticParrot  Parrot
Penguin
      ⎕NL ¯9.5 ⍝ Interfaces
 BirdBehaviour  FishBehaviour
```

⎕NL can also be used to explore Dyalog GUI Objects, .Net types and COM objects.

Dyalog GUI Objects

⎕NL may be used to obtain lists of the Methods, Properties and Events provided by Dyalog APL GUI Objects.

```
     'F' ⎕WC 'Form'
      F.⎕NL -2 ⍝ Properties
 Accelerator  AcceptFiles  Active  AlphaBlend  AutoConf
Border  BCol  Caption ...

      F.⎕NL -3 ⍝ Methods
 Animate  ChooseFont  Detach  GetFocus  GetTextSize
ShowSIP  Wait

      F.⎕NL -8 ⍝ Events
Close  Create  DragDrop  Configure  ContextMenu  DropFiles
DropObjects  Expose  Help ...
```

.Net Classes (Types)

`⎕NL` can be used to explore .Net types.

When a reference is made to an undefined name, and `⎕USING` is set, APL attempts to load the Type from the appropriate .Net Assemblies. If successful, the name is entered into the symbol table with name-class 9.6.

```
      ⎕USING←'System'
      DateTime
(System.DateTime)
      ⎕NL -9
 DateTime
      ⎕NC,⊂'DateTime'
9.6
```

The names of the Properties and Methods of a .Net Type may then be obtained using `⎕NL`.

```
      DateTime.⎕NL -2 ⍝ Properties
 MaxValue  MinValue  Now  Today  UtcNow

      DateTime.⎕NL -3 ⍝ Methods
 get_Now  get_Today  get_UtcNow  op_Addition  op_Equality
...
```

In fact it is not necessary to make a separate reference first, because the expression `Type.⎕NL` (where `Type` is a .Net Type) is itself a reference to Type. So, (with `⎕USING` still set to `'System'`):

```
      Array.⎕NL -3
 BinarySearch  Clear  Copy  CreateInstance  IndexOf
LastIndexOf  Reverse  Sort

      ⎕NL -9
 Array  DateTime
```

Another use for `⎕NL` is to examine .Net *enumerations*. For example:

```
⎕USING←'System.Windows.Forms,system.windows.forms.dll'

      FormBorderStyle.⎕NL -2
Fixed3D  FixedDialog  FixedSingle  FixedToolWindow  None
Sizable  SizableToolWindow

      FormBorderStyle.FixedDialog.value__
3

      FormBorderStyle.({ω,[1.5]⍎¨ω,¨⊂'.value__'}⎕NL -2)
 Fixed3D          2
 FixedDialog      3
 FixedSingle      1
 FixedToolWindow  5
 None             0
 Sizable          4
 SizableToolWindow  6
```

COM Objects

Once a reference to a COM object has been obtained, `⎕NL` may be used to obtain lists of its Methods, Properties and Events.

```
      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')

      xl.⎕NL -2 ⍝ Properties
 _Default  ActiveCell  ActiveChart  ActiveDialog
ActiveMenuBar  ActivePrinter  ActiveSheet  ActiveWindow
...

      xl.⎕NL -3 ⍝ Methods
 _Evaluate  _FindFile  _Run2  _Wait  _WSFunction
ActivateMicrosoftApp  AddChartAutoFormat  AddCustomList
Browse  Calculate ...

      ⎕NL -9
 xl
```

Native File Lock: {R}←X ⎕NLOCK Y

This function assists the controlled update of shared native files by locking a range of bytes.

Locking enables controlled update of native files by co-operating users. A process requesting a lock on a region of a file will be *blocked* until that region becomes available. A *write-lock* is exclusive, whereas a *read-lock* is shared. In other words, any byte in a file may be in one of only three states:

- Unlocked
- Write-locked by exactly one process.
- Read-locked by any number of processes.

`Y` must be a simple integer scalar or vector containing 1, 2 or 3 items namely:

1. Tie number
2. Offset (from 0) of first byte of region. Defaults to 0
3. Number of bytes to lock. Defaults to maximum possible file size

`X` is optional. If present, it must be a simple integer scalar or vector containing 1 or 2 items, namely:

1. Type: 0: Unlock, 1:Read lock, 2:Write lock.
2. Timeout: Number of seconds to wait for lock until TIMEOUT error. Defaults to indefinite wait.

The shy result `R` is `Y`. To unlock the file, this value should subsequently be supplied in the right argument to `0⎕NLOCK`.

Examples:

```
      2 ⎕NLOCK ¯1          ⍝ write-lock whole file
      0 ⎕NLOCK ¯1          ⍝ unlock whole file.
      1 ⎕NLOCK ¯1          ⍝ read (share) lock whole file.
      2 ⎕NLOCK¨⎕NNUMS      ⍝ write-lock all files.
      0 ⎕NLOCK¨⎕NNUMS      ⍝ unlock all files.

      1 ⎕NLOCK ¯1 12 1     ⍝ read-lock byte 12.
      1 ⎕NLOCK ¯1 0 10     ⍝ read-lock first 10 bytes.
      2 ⎕NLOCK ¯1 20       ⍝ write-lock from byte 20 onwards.
      2 ⎕NLOCK ¯1 10 2     ⍝ write-lock 2 bytes from byte 10
      0 ⎕NLOCK ¯1 12 1     ⍝ remove lock from byte 12.
```

To lock the region immediately beyond the end of the file prior extending it:

```
    region←2 ⎕NLOCK ¯1, ⎕NSIZE ¯1  ⍝ write-lock from EOF.
    ... ⎕NAPPEND ¯1                ⍝ append bytes to file
    ... ⎕NAPPEND ¯1                ⍝ append bytes to file
    0 ⎕NLOCK ¯1,region             ⍝ release lock.
```

The left argument may have a second optional item that specifies a *timeout* value. If a lock has not been acquired within this number of seconds, the acquisition is abandoned and a `TIMEOUT` error reported.

```
    2 10 ⎕nlock ¯1      ⍝ wait up to 10 seconds for lock.
```

Notes:

There is no *per-byte* cost associated with region locking. It takes the same time to lock/unlock a region, irrespective of that region's size.

Different file servers implement locks in slightly different ways. For example on some systems, locks are *advisory*. This means that a write lock on a region precludes other locks intersecting that region, but doesn't stop reads or writes across the region. On the other hand, *mandatory* locks block both other locks *and* read/write operations. `⎕NLOCK` will just pass the server's functionality along to the APL programmer without trying to standardise it across different systems.

All locks on a file will be removed by `⎕NUNTIE`.

Blocked locking requests can be freed by a strong interrupt. Under Windows, this operation is performed from the Dyalog APL pop-up menu in the system tray.

Errors

In this release, an attempt to unlock a region that contains bytes that have not been locked results in a `DOMAIN` error.

A `LIMIT ERROR` results if the operating system lock daemon has insufficient resources to honour the locking request.

Some systems support only write locks. In this case an attempt to set a read lock will generate a DOMAIN ERROR, and it may be appropriate for the APL programmer to trap the error and apply a write lock.

No attempt will be made to detect deadlock. Some servers do this and if such a condition is detected, a DEADLOCK error (1008) will be reported.

Native File Names: R←⎕NNAMES

This niladic function reports the names of all currently open native files. R is a character matrix. Each row contains the name of a tied native file padded if necessary with blanks. The names are **identical** to those that were given when opening the files with ⎕NCREATE or ⎕NTIE. The rows of the result are in the order in which the files were tied.

Native File Numbers: R←⎕NNUMS

This niladic function reports the tie numbers associated with all currently open native files. R is an integer vector of negative tie numbers. The elements of the result are in the order in which the files were tied.

Enqueue Event: {R}←{X}⎕NQ Y

This system function generates an event or invokes a method.

While APL is executing, events occur "naturally" as a result of user action or of communication with other applications. These events are added to the event queue as and when they occur, and are subsequently removed and processed one by one by ⎕DQ. ⎕NQ provides an "artificial" means to generate an event and is analogous to ⎕SIGNAL.

If the left argument X is omitted or is 0, ⎕NQ adds the event specified by Y to the bottom of the event queue. The event will subsequently be processed by ⎕DQ when it reaches the top of the queue.

If X is 1, the event is actioned **immediately** by ⎕NQ itself and is processed in exactly the same way as it would be processed by ⎕DQ. For example, if the event has a callback function attached, ⎕NQ will invoke it directly. See ⎕DQ for further details.

Note that it is not possible for one thread to use 1 ⎕NQ to send an event to another thread.

If X is 2 and the name supplied is the name of an event, ⎕NQ performs the default processing for the event immediately, but does **not** invoke a callback function if there is one attached.

If X is 2 and the name supplied is the name of a (Dyalog APL) method, ⎕NQ invokes the method. Its (shy) result is the result produced by the method.

If X is 3, ⎕NQ invokes a method in an OLE Control. The (shy) result of ⎕NQ is the result produced by the method.

If X is 4, ⎕NQ signals an event from an ActiveXControl object to its host application. The (shy) result of ⎕NQ is the result returned by the host application and depends upon the syntax of the event. This case is only applicable to ActiveXControl objects.

Y is a nested vector containing an event message. The first two elements of Y are :

Y[1] : Object name -	a character vector
Y[2] : Event Type -	a numeric scalar or character vector which specifies an event or method.

`Y[1]` must contain the name of an existing object. If not, `⎕NQ` terminates with a `VALUE ERROR`. If `Y[2]` specifies a standard event type, subsequent elements must conform to the structure defined for that event type. If not, `⎕NQ` terminates with a `SYNTAX ERROR`. If `Y[2]` specifies a non-standard event type, `Y[3]` onwards (if present) may contain arbitrary information. Although any event type not listed herein may be used, numbers in the range 0-1000 are reserved for future extensions.

If `⎕NQ` is used monadically, or with a left argument of 0, its (shy) result is always an empty character vector. If a left argument of 1 is specified, `⎕NQ` returns `Y` unchanged or a modified `Y` if the callback function returns its modified argument as a result. If the left argument is 2, `⎕NQ` returns either the value 1 or a value that is appropriate.

Examples

```
      ⍝ Send a keystroke ("A") to an Edit Field
      ⎕NQ 'TEST.ED' 'KeyPress' 'A'

      ⍝ Iconify all top-level Forms
      {⎕NQ ⍵ 'StateChange' 1}¨'Form'⎕WN'.'

      ⍝ Set the focus to a particular field
      ⎕NQ 'TEST.ED3' 40

      ⍝ Throw a new page on a printer
      1 ⎕NQ 'PR1' 'NewPage'


      ⍝ Terminate ⎕DQ under program control

      'TEST'⎕WC 'Form' ... ('Event' 1001 1)
      ...
      ⎕DQ 'TEST'
      ...
      ⎕NQ 'TEST' 1001  ⍝ From a callback

      ⍝ Call GetItemState method for a TreeView 'F.TV'
      +2 ⎕NQ'F.TV' 'GetItemState' 6
96

      +2 ⎕NQ'.' 'GetEnvironment' 'Dyalog'
c:\Z\2\dyalog82
```

Nested Representation: R←⎕NR Y

Y must be a simple character scalar or vector which represents the name of a function or a defined operator.

If Y is a name of a defined function or defined operator, R is a vector of text vectors. The first element of R contains the text of the function or operator header. Subsequent elements contain lines of the function or operator. Elements of R contain no unnecessary blanks, except for leading indentation of control structures and the blanks which precede comments.

If Y is the name of a variable, a locked function or operator, an external function or a namespace, or is undefined, R is an empty vector.

Example

```
      ∇R←MEAN X    ⍝ Average
[1]   R←(+/X)÷⍴X
      ∇

      +F←⎕NR'MEAN'
  R←MEAN X    ⍝Average   R←(+/X)÷⍴X

      ⍴F
2
      DISPLAY F
.→-----------------------------------------.
| .→----------------------.  .→-----------. |
| | R←MEAN X    ⍝ Average|  | R←(+/X)÷⍴X| |
| '----------------------'  '-----------' |
'∊-----------------------------------------'
```

The definition of ⎕NR has been extended to names assigned to functions by specification (←), and to local names of functions used as operands to defined operators. In these cases, the result of ⎕NR is identical to that of ⎕CR except that the representation of defined functions and operators is as described above.

Example

```
      AVG←MEAN∘,

      +F←⎕NR'AVG'
   R←MEAN X    ⍝ Average   R←(+/X)÷⍴X  ∘,

      ⍴F
3
      DISPLAY F
.→-----------------------------------------------.
| .→-------------------------------------.       |
| | .→---------------------.  .→----------. | ∘ , |
| | | R←MEAN X    ⍝ Average|  | R←(+/X)÷⍴X| | - - |
| | '----------------------'  '-----------' |     |
| '∊-------------------------------------'       |
'∊-----------------------------------------------'
```

Native File Read: R←⎕NREAD Y

This monadic function reads data from a native file. Y is a 3- or 4-element integer vector whose elements are as follows:

[1]	negative tie number,
[2]	conversion code (see below),
[3]	count,
[4]	start byte, counting from 0.

Y[2] specifies conversion to an APL internal form as follows. Note that the internal formats for character arrays differ between the Unicode and Classic Editions.

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	8 × count
80	**count**	**8 bits character**	**count**
82[1]	count	8 bits character	count
83	**count**	**8 bits integer**	**count**
160	**2 × count**	**16-bits character**	**count**
163	**2 × count**	**16 bits integer**	**count**
320	**4 × count**	**32-bits character**	**count**
323	4 × count	32 bits integer	count
645	8 × count	64bits floating	count

Unicode Edition : Conversion Codes

[1] Conversion code 82 is permitted in the Unicode Edition for compatibility and causes 1-byte data on file to be *translated* (according to ⎕NXLATE) from ⎕AV indices into normal (Unicode) characters of type 80, 160 or 320.

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	8 × count
82	count	8 bits character	count
83	**count**	**8 bits integer**	**count**
163	**2 × count**	**16 bits integer**	**count**
323	4 × count	32 bits integer	count
645	8 × count	64bits floating	count

Classic Edition : Conversion Codes

Note that types **80, 160** and **320** and **83** and **163** are exclusive to Dyalog APL.

If Y[4] is omitted, data is read starting from the current position in the file (initially, 0).

Example

```
      DATA←⎕NREAD ¯1 160 (0.5×⎕NSIZE ¯1) 0 ⍝ Unicode
      DATA←⎕NREAD ¯1 82 (⎕NSIZE ¯1) 0      ⍝ Classic
```

Native File Rename: {R}←X ⎕NRENAME Y

⎕NRENAME is used to rename a native file.

Y is a negative integer tie number associated with a tied native file. X is a simple character vector or scalar containing a valid (and unused) file name.

The shy result of ⎕NRENAME is the tie number of the renamed file.

Native File Replace: {R}←X ⎕NREPLACE Y

⎕NREPLACE is used to write data to a native file, replacing data which is already there.

X must be a simple homogeneous APL array containing the data to be written.

Y is a 2- or 3-element integer vector whose elements are as follows:

[1]	negative tie number,
[2]	start byte, counting from 0, at which the data is to be written,
[3]	conversion code (optional).

See ⎕NREAD for a list of valid conversion codes.

The shy result is the position within the file of the end of the record, or, equivalently, the start of the following one. Used, for example, in:

```
      ⍝ Replace sequentially from indx.
      {⍺ ⎕NREPLACE tie ⍵}/vec,indx
```

Unicode Edition

Unless you specify the data type in Y[2], a character array will by default be written using type 80. .

If the data will not fit into the specified character width (bytes) `⎕NREPLACE` will fail with a `DOMAIN ERROR`.

As a consequence of these two rules, you must specify the data type (either 160 or 320) in order to write Unicode characters whose code-point are in the range 256-65535 and >65535 respectively.

Example

```
      n←'test'⎕NTIE 0 ⍝ See ⎕NAPPEND example

      ⎕NREAD n 80 3 0
abc
      ⎕NREAD n 160 7
ταβέρνα

      ⎕←'εστιατόριο'⎕NREPLACE n 3
DOMAIN ERROR
      ⎕←'εστιατόριο'⎕NREPLACE n 3
     ^

      ⎕←'εστιατόριο'⎕NREPLACE n 3 160
23
      ⎕NREAD n 80 3 0
abc
      ⎕NREAD n 160 10
εστιατόριο
```

For compatibility with old files, you may specify that the data be converted to type 82 on output. The conversion (to `⎕AV` indices) will be determined by the local value of `⎕AVU`.

Native File Resize: {R}←X ⎕NRESIZE Y

This function changes the size of a native file.

Y is a negative integer tie number associated with a tied native file.

X is a single integer value that specifies the new size of the file in bytes. If X is smaller than the current file size, the file is truncated. If X is larger than the current file size, the file is extended and the value of additional bytes is undefined.

The shy result of ⎕NRESIZE is the tie number of the resized file.

Create Namespace: {R}←{X}⎕NS Y

If specified, X must be a simple character scalar or vector identifying the name of a namespace.

Y is either a character array which represents a list of names of objects to be copied into the namespace, or is an array produced by the ⎕OR of an existing namespace. In the first case, Y must be a simple character scalar, vector, matrix or a nested vector of character vectors identifying zero or more workspace objects to be copied into the namespace X. The identifiers in X and Y may be simple names or compound names separated by '.' and including the names of the special namespaces '#', '##' and ''.

The namespace X is created if it doesn't already exist. If the name is already in use for an object other than a namespace, APL issues a DOMAIN ERROR.

If X is omitted, an unnamed namespace is created.

The objects identified in the list Y are copied into the namespace X.

If X is specified, the result R is the full name (starting #. or ⎕SE.) of the namespace X. If X is omitted, the result R is a namespace reference, or *ref*, to an unnamed namespace.

Examples

```
      +'X'⎕NS''               ë Create namespace X.
#.X
      +'X'⎕NS'VEC' 'UTIL.DISP'ë Copy VEC and DISP to X.
#.X
      )CS X                   ë Change to namespace X.
#.X
      +'Y'⎕NS'#.MAT' '##.VEC' ë Create #.X.Y &copy into it
#.X.Y
      +'#.UTIL'⎕NS'Y.MAT'     ë Copy MAT from Y to #.UTIL.
#.UTIL
      +'#'⎕NS'Y'              ë Copy namespace Y to root.
#
      +''⎕NS'#.MAT'           ë Copy MAT to current space.
#.X
      +''⎕NS''                ë Display current namespace.
#.X
      +'Z'⎕NS ⎕OR'Y'          ë Create namespace from ⎕OR.
#.X.Z

      NONAME←⎕NS ''           ë Create unnamed namespace
      NONAME
#.[Namespace]

      DATA←⎕NS¨4ρ⊂''          ë Create 4-element vector of
                              ë distinct unnamed
namespaces
      DATA
 #.[Namespace]  #.[Namespace]  #.[Namespace]
#.[Namespace]
```

Namespace Indicator: R←⎕NSI

R is a nested vector of character vectors containing the names of the spaces from which functions in the state indicator were called (⍴⎕NSI←→⍴⎕SI).

Note that ⎕NSI contains the names of spaces *from which* functions were called not those *in which* they are currently running.

Example

```
      )OBJECTS
XX      YY

      ⎕VR 'YY.FOO'
     ∇ FOO
[1]    ⎕SE.GOO
     ∇
      ⎕VR'⎕SE.GOO'
     ∇ GOO
[1]    ⎕SI,[1.5]⎕NSI
     ∇

      )CS XX
#.XX
      #.YY.FOO
 GOO  #.YY
 FOO  #.XX
```

Native File Size: R←⎕NSIZE Y

This reports the size of a native file.

Y is a negative integer tie number associated with a tied native file. The result R is the size of the file in bytes.

Native File Tie: {R}←X ⎕NTIE Y

⎕NTIE opens a native file.

X is a simple character vector or scalar containing a valid pathname for an existing native file.

Y is a 1- or 2-element vector. Y[1] is a negative integer value that specifies an (unused) tie number by which the file may subsequently be referred. Y[2] is optional and specifies the mode in which the file is to be opened. This is an integer value calculated as the sum of 2 codes. The first code refers to the type of access needed from users who have already tied the native file. The second code refers to the type of access you wish to grant to users who subsequently try to open the file while you have it open.

Needed from existing users		**Granted to subsequent users**	
0	read access	0	compatibility mode
1	write access	16	no access (exclusive)
2	read and write access	32	read access
		48	write access
		64	read and write access

On Unix systems, the first code (16|mode) is passed to the open(2) call as the access parameter. See include file fcntl.h for details.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

```
      tie←¯1+⌊/0,⎕NNUMS    ⍝ With next available number,
      file ⎕NTIE tie       ⍝ ... tie file.
```

to:

```
      tie←file ⎕NTIE 0     ⍝ Tie with first available no.
```

Example

```
ntie←{                    ⍝ tie file and return tie no.
    ⍺←2+64                ⍝ default all access.
    ⍵ ⎕ntie 0 ⍺           ⍝ return new tie no.
}
```

Null Item: R←⎕NULL

This is a reference to a null item, such as may be returned across the COM interface to represent a null value. An example might be the value of an empty cell in a spreadsheet.

`⎕NULL` may be used in any context that accepts a namespace reference, in particular:

- As the argument to a defined function
- As an item of an array.

```
As the argument to those primitive functions that take
character data arguments, for example: =, ≠, ≡,≢ , ⍴, ⊃,
⊂, ...
```

Example

```
      'EX'⎕WC'OLEClient' 'Excel.Application'
      WB←EX.Workbooks.Open 'simple.xls'

      (WB.Sheets.Item 1).UsedRange.Value2
 [Null]  [Null]  [Null]  [Null]  [Null]
 [Null]    Year  [Null]  [Null]  [Null]
 [Null]    1999    2000    2001    2002
 [Null]  [Null]  [Null]  [Null]  [Null]
 Sales      100      76     120     150
 [Null]  [Null]  [Null]  [Null]  [Null]
 Costs       80      60     100     110
 [Null]  [Null]  [Null]  [Null]  [Null]
 Margin      20      16      20      40
```

To determine which of the cells are filled , you can compare the array with `⎕NULL`.

```
      ⎕NULL≢¨(WB.Sheets.Item 1).UsedRange.Value2
0 0 0 0 0
0 1 0 0 0
0 1 1 1 1
0 0 0 0 0
1 1 1 1 1
0 0 0 0 0
1 1 1 1 1
0 0 0 0 0
1 1 1 1 1
```

Native File Untie: {R}←⎕NUNTIE Y

This closes one or more native files. Y is a scalar or vector of negative integer tie numbers. The files associated with elements of Y are closed. Native file untie with a zero length argument (⎕NUNTIE ⍬) flushes all file buffers to disk - see ⎕FUNTIE for more explanation.

The shy result of ⎕NUNTIE is a vector of tie numbers of the files **actually untied**.

Native File Translate: {R}←{X}⎕NXLATE Y

This associates a character translation vector with a native file or, if Y is 0, with the use by ⎕DR.

A translate vector is a 256-element vector of integers from 0-255. Each element maps the corresponding ⎕AV position onto an ANSI character code.

For example, to map ⎕AV[17+⎕IO] onto ANSI 'a' (code 97), element 17 of the translate vector is set to 97.

⎕NXLATE is a non-Unicode (Classic Editon) feature and is retained in the Unicode Edition, only for compatibility.

Y is either a negative integer tie number associated with a tied native file or 0. If Y is negative, monadic ⎕NXLATE returns the current translation vector associated with the corresponding native file. If specified, the left argument X is a 256-element vector of integers that specifies a new translate vector. In this case, the old translate vector is returned as a shy result. If Y is 0, it refers to the translate vector used by ⎕DR to convert to and from character data.

The system treats a translate vector with value (⍳256)-⎕IO as meaning *no translation* and thus provides raw input/output bypassing the whole translation process.

The default translation vector established at ⎕NTIE or ⎕NCREATE time, maps ⎕AV characters to their corresponding ANSI positions and is derived from the mapping defined in the current output translation table (normally WIN.DOT)

Between them, ANSI and RAW translations should cater for most uses.

Unicode Edition

`⎕NXLATE` is relevant in the Unicode Edition only to process Native Files that contain characters expressed as indices into `⎕AV`, such as files written by the Classic Edition.

In the Unicode Edition, when reading data from a Native File using conversion code 82, incoming bytes are translated first to `⎕AV` indices using the translation table specified by `⎕NXLATE`, and then to type 80, 160 or 320 using `⎕AVU`. When writing data to a Native File using conversion code 82, characters are converted using these two translation tables in reverse.

Sign Off APL: ⎕OFF

This niladic system function terminates the APL session, returning to the shell command level . The active workspace does not replace the last continuation workspace.

Object Representation: R←⎕OR Y

`⎕OR` converts a function, operator or namespace to a special form, described as its *object representation*, that may be assigned to a variable and/or stored on a component file. Classes and Instances are however outside the domain of `⎕OR`.

Taking the `⎕OR` of a function or operator is an extremely fast operation as it simply changes the type information in the object's header, leaving its internal structure unaltered. Converting the object representation back to an executable function or operator using `⎕FX` is also very fast. `⎕OR` is therefore the recommended form for storing functions and operators on component files and is significantly faster than using `⎕CR`, `⎕VR` or `⎕NR`.

`⎕OR` may also be used to convert a namespace (either a plain namespace or a named GUI object created by `⎕WC`) into a form that can be stored in a variable or on a component file. The namespace may be reconstructed using `⎕NS` or `⎕WC` with its original name or with a new one. `⎕OR` may therefore be used to *clone* a namespace or GUI object.

`Y` must be a simple character scalar or vector which contains the name of an APL object.

If `Y` is the name of a variable, the result `R` is its value. In this case, `R←⎕OR Y` is identical to `R←⍎Y`.

Otherwise, R is a special form of the name Y, re-classified as a variable. The rank of R is 0 (R is scalar), and the depth of R is 1. These unique characteristics distinguish the result of ⎕OR from any other object. The type of R (∊R) is itself. Note that although R is scalar, it may not be index assigned to an element of an array unless it is enclosed.

If Y is the name of a function or operator, R is in the domain of the monadic functions Depth (≡), Disclose (⊃), Enclose (⊂), Rotate(⌽), Transpose(⍉), Index ing([ ]), Format (⍕), Identity (+), Shape (⍴), Type (∊) and Unique (∪), of the dyadic functions Assignment (←), Without (~), Index Of (⍳), Intersection (∩), Match (≡), Membership (∊), Not Match (≢) and Union (∪), and of the monadic system functions Canonical Representation (⎕CR), Cross-Reference (⎕REFS), Fix (⎕FX), Format (⎕FMT), Nested Representation (⎕NR) and Vector Representation (⎕VR).

Nested arrays which include the object representations of functions and operators are in the domain of many mixed functions which do not use the values of items of the arrays.

Note that a ⎕OR object can be transmitted through an 'APL-style' TCP socket. This technique may be used to transfer objects including namespaces between APL sessions.

The object representation forms of namespaces produced by ⎕OR may not be used as arguments to any primitive functions. The only operations permitted for such objects (or arrays containing such objects) are ⎕EX, ⎕FAPPEND, ⎕FREPLACE, ⎕NS, and ⎕WC.

Example

```
      F←⎕OR ⎕FX'R←FOO' 'R←10'

      ⍴F

      ⍴⍴F
0
      ≡F
1
      F≡∊F
1
```

The display of the ⎕OR form of a function or operator is a listing of the function or operator. If the ⎕OR form of a function or operator has been enclosed, then the result will display as the operator name preceded by the symbol ∇. It is permitted to apply ⎕OR to a locked function or operator. In this instance the result will display as for the enclosed form.

Examples

```
        F
      ∇ R←FOO
[1]     R←10
      ∇

      ⊂F
 ∇FOO

      ⎕LOCK'FOO'

      ⎕OR'FOO'
∇FOO

      A←⍳5

      A[3]←⊂F

      A
1 2  ∇FOO  4 5
```

For the ⎕OR forms of two functions or operators to be considered identical, their unlocked display forms must be the same, they must either both be locked or unlocked, and any monitors, trace and stop vectors must be the same.

Example

```
      F←⎕OR ⎕FX 'R←A PLUS B' 'R←A+B'

      F≡⎕OR 'PLUS'
1

      1 ⎕STOP 'PLUS'

      F≡⎕OR 'PLUS'
0
```

Namespace Examples

The following example sets up a namespace called UTILS, copies into it the contents of the UTIL workspace, then writes it to a component file:

```
      )CLEAR
CLEAR WS
      )NS UTILS
#.UTILS
      )CS UTILS
#.UTILS
      )COPY UTIL
C:\WDYALOG\WS\UTIL SAVED FRI MAR 17 12:48:06 1995
      )CS
#
      'ORTEST' ⎕FCREATE 1
      (⎕OR'UTILS')⎕FAPPEND 1
```

The namespace can be restored with ⎕NS, using either the original name or a new one:

```
      )CLEAR
CLEAR WS
      'UTILS' ⎕NS ⎕FREAD 1 1
#.UTILS
      )CLEAR
CLEAR WS
      'NEWUTILS' ⎕NS ⎕FREAD 1 1
#.NEWUTILS
```

This example illustrates how ⎕OR can be used to clone a GUI object; in this case a Group containing some Button objects. Note that ⎕WC will accept **only** a ⎕OR object as its argument (or preceded by the "Type" keyword). You may not specify any other properties in the same ⎕WC statement, but you must instead use ⎕WS to reset them afterwards.

```
     'F'⎕WC'Form'
     'F.G1' ⎕WC 'Group' '&One' (10 10)(80 30)
     'F.G1.B2'⎕WC'Button' '&Blue' (40 10)('Style' 'Radio')
     'F.G1.B3'⎕WC'Button' '&Green' (60 10)('Style' 'Radio')
     'F.G1.B1'⎕WC'Button' '&Red' (20 10)('Style' 'Radio')
     'F.G2' ⎕WC ⎕OR 'F.G1'
     'F.G2' ⎕WS ('Caption' 'Two')('Posn' 10 60)
```

Note too that ⎕WC and ⎕NS may be used interchangeably to rebuild *pure* namespaces or GUI namespaces from a ⎕OR object. You may therefore use ⎕NS to rebuild a Form or use ⎕WC to rebuild a pure namespace that has no GUI components.

Search Path: ⎕PATH

⎕PATH is a simple character vector representing a blank-separated list of namespaces. It is approximately analogous to the PATH variable in Windows or UNIX

The ⎕PATH variable can be used to identify a namespace in which commonly used utility functions reside. Functions or operators (**NOT** variables) which are copied into this namespace and *exported* (see ⎕EXPORT) can then be used directly from anywhere in the workspace without giving their full path names.

Example

To make the DISPLAY function available directly from within any namespace.

```
      ⍝ Create and reference utility namespace.
      ⎕PATH←'⎕se.util'⎕ns''
      ⍝ Copy DISPLAY function from UTIL into it.
      'DISPLAY'⎕se.util.⎕cy'UTIL'
      ⍝ (Remember to save the session to file).
```

In detail, ⎕PATH works as follows:

When a reference to a name cannot be found in the current namespace, the system searches for it from left to right in the list of namespaces indicated by ⎕PATH. In each namespace, if the name references a defined function (or operator) *and* the export type of that function is non-zero (see ⎕EXPORT), then it is used to satisfy the reference. If the search exhausts all the namespaces in ⎕PATH without finding a qualifying reference, the system issues a VALUE ERROR in the normal manner.

The special character ↑ stands for the list of namespace ancestors:

```
       ## ##.## ##.##.## ...
```

In other words, the search is conducted upwards through enclosing namespaces, emulating the static scope rule inherent in modern block-structured languages.

Note that the ⎕PATH mechanism is used ONLY if the function reference cannot be satisfied in the current namespace. This is analogous to the case when the Windows or UNIX PATH variable begins with a '.'.

Examples

```
   ⎕PATH              Search in ...

1. '⎕se.util'          Current space,   then
                       ⎕se.util,        then
                       VALUE ERROR

2. '↑'                 Current space
                       Parent space: ##
                       Parent's parent space:  ##.##
                       ...
                       Root: # (or ⎕se if current space
                                was inside ⎕se)
                       VALUE ERROR

3. 'util ↑ ⎕se.util'   Current space
                       util (relative to current space)
                       Parent space: ##
                       ...
                       Root: # or ⎕se
                       ⎕se.util
                       VALUE ERROR
```

Note that `⎕PATH` is a *session* variable. This means that it is workspace-wide and survives `)LOAD` and `)CLEAR`. It can of course, be localised by a defined function or operator.

Program Function Key: R←{X}⎕PFKEY Y

⎕PFKEY is a system function that sets or queries the programmable function keys. ⎕PFKEY associates a sequence of keystrokes with a function key. When the user subsequently presses the key, it is as if he had typed the associated keystrokes one by one.

Y is an integer scalar in the range 0-255 specifying a programmable function key. If X is omitted the result R is the current setting of the key. If the key has not been defined previously, the result is an empty character vector.

If X is specified it is a simple or nested character vector defining the new setting of the key. The value of X is returned in the result R.

The elements of X are either character scalars or 2-element character vectors which specify Input Translate Table codes.

Programmable function keys are recognised in any of the three types of window (SESSION, EDIT and TRACE) provided by the Dyalog APL development environment. ⎕SR operates with the 'raw' function keys and ignores programmed settings.

Note that key definitions can reference other function keys.

The size of the buffer associated with ⎕PFKEY is specified by the *pfkey_size* parameter.

Examples

```
      (')FNS',⊂'ER') ⎕PFKEY 1
 )FNS  ER

      DISPLAY ⎕PFKEY 1
.→-------------.
|          .→-. |
| ) F N S |ER| |
| - - - - '--' |
'∊-------------'

      (')VARS',⊂'ER') ⎕PFKEY 2
 )VARS  ER
      'F1' 'F2' ⎕PFKEY 3   ⍝ Does )FNS and )VARS
 F1  F2
```

Print Precision: ⎕PP

⎕PP is the number of significant digits in the display of numeric output.

⎕PP may be assigned any integer value in the range 1 to 17. The value in a clear workspace is 10. Note that in all Versions of Dyalog APL prior to Version 11.0, the maximum value for ⎕PP was 16.

⎕PP is used to format numbers displayed directly. It is an implicit argument of monadic function Format (⍕), monadic ⎕FMT and for display of numbers via ⎕ and ⍞ output. ⎕PP is ignored for the display of integers.

Examples:

```
      ⎕PP←10

      ÷3 6
0.3333333333 0.1666666667

      ⎕PP←3

      ÷3 6
0.333 0.167
```

If ⎕PP is set to its maximum value of 17, floating-point numbers may be converted between binary and character representation without loss of precision. In particular, if ⎕PP is 17 and ⎕CT is 0 (to ensure exact comparison), for any floating-point number N the expression N=⍎⍕N is true. Note however that *denormal* numbers are an exception to this rule.

Numbers, very close to zero, in the range 2.2250738585072009E¯308 to 4.9406564584124654E¯324 are called *denormal* numbers.

Such numbers can occur as the result of calculations and are displayed correctly. However, denormals cannot be specified as literals and are converted to zero on input.

Numbers below the lower end of this range (4.94E¯324) are indistinguishable from zero in IEEE double floating point format.

Print Width: ⎕PW

⎕PW is the maximum number of output characters per line before folding the display.

⎕PW may be assigned any integer value in the range 30 to 32767.

If an attempt is made to display a line wider than ⎕PW, then the display will be folded at or before the ⎕PW width and the folded portions indented 6 spaces. The display of a simple numeric array may be folded at a width less than ⎕PW so that individual numbers are not split.

⎕PW only affects output, either direct or through ⎕ output. It does not affect the result of the function Format (⍕), of the system function ⎕FMT, or output through the system functions ⎕ARBOUT and ⎕ARBIN, or output through ⍞.

Session window is resized. In these circumstances, a value assigned to ⎕PW will only be effective until the Session Window is next resized.

Examples

```
      ⎕PW←30

      ⎕←3⍴÷3
0.3333333333 0.3333333333
      0.3333333333
```

Cross References: R←⎕REFS Y

Y must be a simple character scalar or vector, identifying the name of a function or operator, or the object representation form of a function or operator (see ⎕OR). R is a simple character matrix, with one name per row, of identified names in the function or operator in Y excluding distinguished names of system constants, variables or functions.

Example

```
      ⎕VR'OPTIONS'
     ∇ OPTIONS;OPTS;INP
[1]   ⍝ REQUESTS AND EXECUTES AN OPTION
[2]    OPTS ←'INPUT' 'REPORT' 'END'
[3]   IN:INP←ASK'OPTION:'
[4]    →EXρ⍨(⊂INP)∊OPTS
[5]    'INVALID OPTION. SELECT FROM',OPTS ⋄ →IN
[6]   EX:→EX+OPTSι⊂INP
[7]    INPUT ⋄ →IN
[8]    REPORT ⋄ →IN
[9]   END:
     ∇

    ⎕REFS'OPTIONS'
ASK
END
EX
IN
INP
INPUT
OPTIONS
OPTS
REPORT
```

If Y is locked or is an External Function, R contains its name only. For example:

```
      ⎕LOCK 'OPTIONS' ⋄ ⎕REFS 'OPTIONS'
OPTIONS
```

If Y is the name of a primitive, external or derived function, R is an empty matrix with shape 0 0.

Random Link: ⎕RL

⎕RL is used or set to establish a base for generating random numbers.

⎕RL may be assigned any integer value in the range 1 to 2147483646. The value in a clear workspace is 16807.

Repeatable results can be obtained from Roll or Deal if ⎕RL is set to a particular value first.

⎕RL is used and set implicitly by the functions Roll and Deal (?).

Examples

```
      ⎕RL
16807

      ?9 9 9
2 7 5

      ?9
7

      ⎕RL
984943658

      ⎕RL←16807
      ?9 9 9
2 7 5
```

Response Time Limit: ⎕RTL

A non-zero value in ⎕RTL places a time limit, in seconds, for input requested via ⍞ , ⎕ARBIN , and ⎕SR. ⎕RTL may be assigned any integer in the range 0 to 32767. The value is a clear workspace is 0.

Example

```
      ⎕RTL←5 ⋄ ⍞←'FUEL QUANTITY?' ⋄ R←⍞
FUEL QUANTITY?
TIMEOUT
      ⎕RTL←5 ⋄ ⍞←'FUEL QUANTITY?' ⋄ R←⍞
```

Save Workspace: {R}←{X}⎕SAVE Y

Y must be a simple character scalar or vector, identifying a workspace name. R is a simple logical scalar. The active workspace is saved with the given name in Y. In the active workspace, the value 1 is returned. The result is suppressed if not used or assigned.

The optional left argument X is either 0 or 1. If X is omitted or 1, the saved version of the workspace has execution suspended at the point of exit from the ⎕SAVE function. If the saved workspace is subsequently loaded by ⎕LOAD, execution is resumed, and the value 0 is returned if the result is used or assigned, or otherwise the result is suppressed. In this case, the latent expression value (⎕LX) is ignored.

If X is 0, the workspace is saved without any State Indicator in effect. The effect is the same as if you first executed)RESET and then)SAVE. In this case, when the workspace is subsequently loaded, the value of the latent expression (⎕LX) is honoured if applicable.

A DOMAIN ERROR is reported if the name in Y is not a valid workspace name, or the reference is to an unauthorised directory.

⎕SAVE will fail and issue DOMAIN ERROR if any threads (other than the root thread 0) are running.

Note that the values of all system variables (including ⎕SM) and all GUI objects are saved.

Example

```
      (⊃'SAVED' 'ACTIVE' [⎕IO+⎕SAVE'TEMP']),' WS'
ACTIVE WS
      ⎕LOAD 'TEMP'
SAVED WS
```

Screen Dimensions: R←⎕SD

⎕SD is a 2-element integer vector containing the number of rows and columns on the screen, or in the USER window.

For asynchronous terminals under UNIX, the screen size is taken from the terminal database **terminfo** or **termcap**.

In window implementations of Dyalog APL, ⎕SD reports the current size (in characters) of the USER window or the current size of the SM object, whichever is appropriate.

Session Namespace: ⎕SE

⎕SE is a system namespace. Its GUI components (MenuBar, ToolBar, and so forth) define the appearance and behaviour of the APL Session window and may be customised to suit individual requirements.

⎕SE is maintained separately from the active workspace and is not affected by)LOAD or)CLEAR. It is therefore useful for containing utility functions. The contents of ⎕SE may be saved in and loaded from a .DSE file.

See *User Guide* for further details.

Execute (UNIX) Command: {R}←⎕SH Y

⎕SH executes a UNIX shell or a Windows Command Procesor. ⎕SH is a synonym of ⎕CMD. Either function may be used in either environment (UNIX or Windows) with exactly the same effect. ⎕SH is probably more natural for the UNIX user. This section describes the behaviour of ⎕SH and ⎕CMD under UNIX. See ⎕CMD for a discussion of the behaviour of these system functions under Windows.

Y must be a simple character scalar or vector representing a UNIX shell command. R is a nested vector of character vectors.

Y may be any acceptable UNIX command. It could cause another process to be entered, such as sed or vi. If the command does not return a result, R is ⊂'' but the result is suppressed if not explicitly used or assigned. If the command has a non-zero exit code, then APL will signal a DOMAIN ERROR. If the command returns a result and has a zero exit code, then each element of R will be a line from the standard output (stdout) of the command. Output from standard error (stderr) is not captured unless redirected to stdout.

Examples

```
      ⎕SH'ls'
FILES WS temp

      ⎕SH 'rm WS/TEST'

      ⎕SH 'grep bin /etc/passwd ; exit 0'
bin:!:2:2::/bin:

      ⎕SH 'apl MYWS <inputfile >out1 2>out2 &'
```

Start (UNIX) Auxiliary Processor: X ⎕SH Y

Used dyadically, ⎕SH starts an Auxiliary Processor. The effect, as far as the APL user is concerned, is identical under both Windows and UNIX although there are differences in the method of implementation. ⎕SH is a synonym of ⎕CMD. Either function may be used in either environment (UNIX or Windows) with exactly the same effect. ⎕SH is probably more natural for the UNIX user. This section describes the behaviour of ⎕SH and ⎕CMD under UNIX. See ⎕CMD for a discussion of the behaviour of these system functions under Windows.

X must be a simple character vector. Y may be a simple character scalar or vector, or a nested character vector.

⎕SH loads the Auxiliary Processor from the file named by X using a search-path defined by the environment variable WSPATH.

The effect of starting an AP is that one or more **external functions** are defined in the workspace. These appear as locked functions and may be used in exactly the same way as regular defined functions.

When an external function is used in an expression, the argument(s) (if any) are **piped** to the AP for processing. If the function returns a result, APL halts while the AP is processing and waits for the result. If not it continues processing in parallel.

The syntax of dyadic ⎕SH is similar to the UNIX execl(2) system call, where 'taskname' is the name of the auxiliary processor to be executed and arg0 through argn are the parameters of the calling line to be passed to the task, viz.

```
      'taskname' ⎕SH 'arg0' 'arg1' ... 'argn'
```

See *User Guide* for further information.

Examples

```
      'xutils' ⎕SH 'xutils' 'ss' 'dbr'

      '/bin/sh' ⎕SH 'sh' '-c' 'adb test'
```

Shadow Name: ⎕SHADOW Y

Y must be a simple character scalar, vector or matrix identifying one or more APL names. For a vector Y, names are separated by one or more blanks. For a matrix Y, each row is taken to be a single name.

Each valid name in Y is shadowed in the most recently invoked defined function or operator, as though it were included in the list of local names in the function or operator header. The class of the name becomes 0 (undefined). The name ceases to be shadowed when execution of the shadowing function or operator is completed. Shadow has no effect when the state indicator is empty.

If a name is ill-formed, or if it is the name of a system constant or system function, DOMAIN ERROR is reported.

If the name of a top-level GUI object is shadowed, it is made inactive.

Example

```
      ⎕VR'RUN'
     ∇ NAME RUN FN
[1]    ⍝ RUNS FUNCTION NAMED <NAME> DEFINED
[2]    ⍝ FROM REPRESENTATION FORM <FN>
[3]     ⎕SHADOW NAME
[4]     ⍎⎕FX FN
     ∇

      0 ⎕STOP 'RUN'

      'FOO' RUN 'R←FOO' 'R←10'
10

RUN[0]

      )SINL
RUN[0]*       FOO      FN       NAME

      →⎕LC

      FOO
VALUE ERROR
      FOO
      ^
```

State Indicator: R←⎕SI

R is a nested vector of vectors giving the names of the functions or operators in the execution stack.

Example

```
      )SI
PLUS[2]*
.
MATDIV[4]
FOO[1]*
⍎

      ⎕SI
 PLUS  MATDIV  FOO

      (⍴⎕LC)=⍴⎕SI
1
```

If execution stops in a callback function, ⎕DQ will appear on the stack, and may occur more than once

```
      )SI
ERRFN[7]*
⎕DQ
CALC
⎕DQ
MAIN
```

To edit the function on the top of the stack:

```
      ⎕ED ⊃⎕SI
```

The name of the function which called this one:

```
      ⊃1↓⎕SI
```

To check if the function ∆N is pendent:

```
     ((⊂∆N)∊1↓⎕SI)/'Warning : ',∆N,' is pendent'
```

See also ⎕XSI.

Signal Event: {X}⎕SIGNAL Y

Y must be a single positive integer scalar or an empty vector. X is optional. If present, X must be a simple character scalar or vector, or an object reference. If Y is en empty vector, nothing is signalled.

Y is taken to be an event number in the range 1-999. X is an optional text message. If omitted, the standard event message shown for the corresponding event number in Figure 7(i) or 7(ii) is assumed. If there is no standard message, a message of the form `ERROR NUMBER n` is composed, where n is the event number in Y.

The effect of the system function is to interrupt execution. The state indicator is cut back to exit from the function or operator containing the line that invoked ⎕SIGNAL or the Execute (⍎) expression that invoked ⎕SIGNAL, and an error is then generated.

An error interrupt may be trapped if the system variable ⎕TRAP is set to intercept the event. Otherwise, the standard system action is taken (which may involve cutting back the state indicator further if there are locked functions or operators in the state indicator). The standard event message is replaced by the text given in X, if present.

Example

```
      ⎕VR'DIVIDE'
     ∇ R←A DIVIDE B;⎕TRAP
[1]    ⎕TRAP←11 'E' '→ERR'
[2]    R←A÷B ⋄ →0
[3]   ERR:'DIVISION ERROR' ⎕SIGNAL 11
     ∇

      2 4 6 DIVIDE 0
DIVISION ERROR
      2 4 6 DIVIDE 0
     ^
```

If you are using the Microsoft .Net Framework, you may use ⎕SIGNAL to throw an exception by specifying a value of 90 in Y. In this case, if you specify the optional left argument X , it must be a reference to a .Net object that is or derives from the Microsoft .Net class System.Exception. The following example illustrates a *constructor* function CTOR that expects to be called with a value for ⎕IO (0 or 1)

```
     ∇ CTOR IO;EX
[1]    :If IO∊0 1
[2]        ⎕IO←IO
[3]    :Else
[4]        EX←ArgumentException.New'IO must be 0 or 1'
[5]        EX ⎕SIGNAL 90
[6]    :EndIf
     ∇
```

Size of Object: R←⎕SIZE Y

Y must be a simple character scalar, vector or matrix. A scalar or vector is treated as a single row matrix. Each row is taken to be an APL name. R is a simple integer vector of non-negative elements and of shape 1↑¯2↑1 1,⍴Y.

If the name in a row of Y identifies an object with an active referent, the workspace required in bytes by that object is returned in the corresponding element of R. Otherwise, 0 is returned in that element of R.

The result returned for an external variable is the space required to store the external array. The result for a system constant, variable or function is 0. The result returned for a GUI object gives the amount of workspace needed to store it, but excludes the space required for its children.

Note: Wherever possible, Dyalog APL 'shares' the whole or part of a workspace object rather than generates a separate copy.

Examples

```
      ⎕VR 'FOO'
     ∇ R←FOO
[1]    R←10
     ∇

      A←⍳10

      'EXT/ARRAY' ⎕XT'E' ⋄ E←⍳20

      ⎕SIZE ↑'A' 'FOO' 'E' 'UND'
28 76 120 0
```

Screen Map: ⎕SM

⎕SM is a system variable that defines a character-based user interface (as opposed to a graphical user interface). In versions of Dyalog APL that support asynchronous terminals, ⎕SM defines a **form** that is displayed on the **USER SCREEN**. The implementation of ⎕SM in "window" environments is compatible with these versions. In Dyalog APL/X, ⎕SM occupies its own separate window on the display, but is otherwise equivalent. In versions of Dyalog APL with GUI support, ⎕SM either occupies its own separate window (as in Dyalog APL/X) or, if it exists, uses the window assigned to the SM object. This allows ⎕SM to be used in a GUI application in conjunction with other GUI components.

In general ⎕SM is a nested matrix containing between 3 and 13 columns. Each row of ⎕SM represents a **field**; each column a **field attribute**.

The columns have the following meanings :

Column	Description	Default
1	Field Contents	N/A
2	Field Position - Top Row	N/A
3	Field Position - Left Column	N/A
4	Window Size - Rows	0
5	Window Size - Columns	0
6	Field Type	0
7	Behaviour	0
8	Video Attributes	0
9	Active Video Attributes	¯1
10	Home Element - Row	1
11	Home Element - Column	1
12	Scrolling Group - Vertical	0
13	Scrolling Group - Horizontal	0

With the exception of columns 1 and 8, all elements in ⎕SM are integer scalar values.

Elements in column 1 (Field Contents) may be :

a) A numeric scalar

b) A numeric vector

c) A 1-column numeric matrix

d) A character scalar

e) A character vector

f) A character matrix (rank 2)

g) A nested matrix defining a sub-form whose structure and contents must conform to that defined for ⎕SM as a whole. This definition is recursive. Note however that a sub-form must be a matrix - a vector is not allowed.

Elements in column 8 (Video Attributes) may be :

a) An integer scalar that specifies the appearance of the entire field.

b) An integer array of the same shape as the field contents. Each element specifies the appearance of the corresponding element in the field contents.

Screen Management (DOS & Async Terminals)

Dyalog APL for UNIX systems (Async terminals) manages two screens; the SESSION screen and the USER screen. If the SESSION screen is current, an assignment to ⎕SM causes the display to switch to the USER screen and show the form defined by ⎕SM.

If the USER screen is current, any change in the value of ⎕SM is immediately reflected by a corresponding change in the appearance of the display. However, an assignment to ⎕SM that leaves its value unchanged has no effect.

Dyalog APL automatically switches to the SESSION screen for default output, if it enters immediate input mode (6-space prompt), or through use of ⎕ or ⍞. This means that typing

```
      ⎕SM ← expression
```

in the APL session will cause the screen to switch first to the USER screen, display the form defined by ⎕SM, and then switch back to the SESSION screen to issue the 6-space prompt. This normally happens so quickly that all the user sees is a flash on the screen. To retain the USER screen in view it is necessary to issue a call to ⎕SR or for APL to continue processing
e.g.

```
      ⎕SM ← expression  ⋄  ⎕SR 1
```
or
```
      ⎕SM ← expression  ⋄  ⎕DL 5
```

Screen Management (Window Versions)

In Dyalog APL/X, and optionally in Dyalog APL/W, ⎕SM is displayed in a separate **USER WINDOW** on the screen. In an end-user application this may be the only Dyalog APL window. However, during development, there will be a SESSION window, and perhaps EDIT and TRACE windows too.

The USER Window will only accept input during execution of ⎕SR. It is otherwise "output-only". Furthermore, during the execution of ⎕SR it is the only active window, and the SESSION, EDIT and TRACE Windows will not respond to user input.

Screen Management (GUI Versions)

In versions of Dyalog APL that provide GUI support, there is a special SM object that defines the position and size of the window to be associated with ⎕SM. This allows character-mode applications developed for previous versions of Dyalog APL to be migrated to and integrated with GUI environments without the need for a total re-write.

Effect of Localisation

Like all system variables (with the exception of ⎕TRAP) ⎕SM is subject to "pass-through localisation". This means that a localised ⎕SM assumes its value from the calling environment. The localisation of ⎕SM does not, of itself therefore, affect the appearance of the display. However, reassignment of a localised ⎕SM causes the new form to overlay rather than replace whatever forms are defined further down the stack. The localisation of ⎕SM thus provides a simple method of defining pop-up forms, help messages, etc.

The user may edit the form defined by ⎕SM using the system function ⎕SR. Under the control of ⎕SR the user may change the following elements in ⎕SM which may afterwards be referenced to obtain the new values.

Column 1 :	Field Contents
Column 10 :	Home Element - Row (by scrolling vertically)
Column 11 :	Home Element - Column (by scrolling horizontally)

Screen Read: R←{X}⎕SR Y

⎕SR is a system function that allows the user to edit or otherwise interact with the form defined by ⎕SM.

In versions of Dyalog APL that support asynchronous terminals, if the current screen is the SESSION screen, ⎕SR immediately switches to the USER SCREEN and displays the form defined by ⎕SM.

In Dyalog APL/X, ⎕SR causes the input cursor to be positioned in the USER window. During execution of ⎕SR, only the USER Window defined by ⎕SM will accept input and respond to the keyboard or mouse. The SESSION and any EDIT and TRACE Windows that may appear on the display are dormant.

In versions of Dyalog APL with GUI support, a single SM object may be defined. This object defines the size and position of the ⎕SM window, and allows ⎕SM to be used in conjunctions with other GUI components. In these versions, ⎕SR acts as a superset of ⎕DQ (see ⎕DQ) but additionally controls the character-based user interface defined by ⎕SM.

Y is an integer vector that specifies the fields which the user may visit. In versions with GUI support, Y may additionally contain the names of GUI objects with which the user may also interact.

If specified, X may be an enclosed vector of character vectors defining EXIT_KEYS or a 2-element nested vector defining EXIT_KEYS and the INITIAL_CONTEXT.

The result R is the EXIT_CONTEXT.

Thus the 3 uses of ⎕SR are :

```
      EXIT_CONTEXT ← ⎕SR FIELDS

      EXIT_CONTEXT ← (⊂EXIT_KEYS)⎕SR FIELDS

      EXIT_CONTEXT ← (EXIT_KEYS)(INITIAL_CONTEXT)⎕SR FIELDS
```

FIELDS

If an element of Y is an integer scalar, it specifies a field as the index of a row in ⎕SM (if ⎕SM is a vector it is regarded as having 1 row).

If an element of Y is an integer vector, it specifies a sub-field. The first element in Y specifies the top-level field as above. The next element is used to index a row in the form defined by ⊃⎕SM[Y[1];1] and so forth.

If an element of Y is a character scalar or vector, it specifies the name of a top-level GUI object with which the user may also interact. Such an object must be a "top-level" object, i.e. the Root object ('.') or a Form or pop-up Menu. This feature is implemented ONLY in versions of Dyalog APL with GUI support.

EXIT_KEYS

Each element of EXIT_KEYS is a 2-character code from the Input Translate Table for the keyboard. If the user presses one of these keys, ⎕SR will terminate and return a result.

If EXIT_KEYS is not specified, it defaults to :

```
       'ER' 'EP' 'QT'
```

which (normally) specifies <Enter>, <Esc> and <Shift+Esc>.

INITIAL_CONTEXT

This is a vector of between 3 and 6 elements with the following meanings and defaults:

Element	Description	Default
1	Initial Field	N/A
2	Initial Cursor Position - Row	N/A
3	Initial Cursor Position - Col	N/A
4	Initial Keystroke	''
5	(ignored)	N/A
6	Changed Field Flags	0

Structure of INITIAL_CONTEXT

INITIAL_CONTEXT[1] specifies the field in which the cursor is to be placed. It is an integer scalar or vector, and must be a member of Y. It must not specify a field which has ÂÞÝÝÔÓ behaviour (64), as the cursor is not allowed to enter such a field.

INITIAL_CONTEXT[2 3] are integer scalars which specify the initial cursor position within the field in terms of row and column numbers.

INITIAL_CONTEXT[4] is either empty, or a 2-element character vector specifying the initial keystroke as a code from the Input Translate Table for the keyboard.

INITIAL_CONTEXT[5] is ignored. It is included so that the EXIT_CONTEXT result of one call to ⎕SR can be used as the INITIAL_CONTEXT to a subsequent call.

INITIAL_CONTEXT[6] is a boolean scalar or vector the same length as Y. It specifies which of the fields in Y has been modified by the user.

EXIT_CONTEXT

The result `EXIT_CONTEXT` is a 6 or 9-element vector whose first 6 elements have the same structure as the `INITIAL_CONTEXT`. Elements 7-9 **only** apply to those versions of Dyalog APL that provide mouse support.

Element	Description
1	Final Field
2	Final Cursor Position - Row
3	Final Cursor Position - Col
4	Terminating Keystroke
5	Event Code
6	Changed Field Flags
7	Pointer Field
8	Pointer Position - Row
9	Pointer Position - Col

Structure of the Result of `⎕SR`

`EXIT_CONTEXT[1]` contains the field in which the cursor was when `⎕SR` terminated due to the user pressing an exit key or due to an event occurring. It is an integer scalar or vector, and a member of `Y`.

`EXIT_CONTEXT[2 3]` are integer scalars which specify the row and column position of the cursor within the field `EXIT_CONTEXT[1]` when `⎕SR` terminated.

`EXIT_CONTEXT[4]` is a 2-element character vector specifying the last keystroke pressed by the user before `⎕SR` terminated. Unless `⎕SR` terminated due to an event, `EXIT_CONTEXT[4]` will contain one of the exit keys defined by `X`. The keystroke is defined in terms of an Input Translate Table code.

`EXIT_CONTEXT[5]` contains the **sum** of the event codes that caused `⎕SR` to terminate. For example, if the user pressed a mouse button on a `ÂÞÝÝÔÓ` field (event code 64) **and** the current field has `ÒÔÇÍÊÍÈÇ` behaviour (event code 2) `EXIT_CONTEXT[5]` will have the value 66.

`EXIT_CONTEXT[6]` is a boolean scalar or vector the same length as `Y`. It specifies which of the fields in `Y` has been modified by the user during **this** `⎕SR`, ORed with `INITIAL_CONTEXT[6]`. Thus if the `EXIT_CONTEXT` of one call to `⎕SR` is fed back as the `INITIAL_CONTEXT` of the next, `EXIT_CONTEXT[6]` records the fields changed since the start of the process.

EXIT_CONTEXT (Window Versions)

`⎕SR` returns a 9-element result **ONLY** if it is terminated by the user pressing a mouse button. In this case :

`EXIT_CONTEXT[7]` contains the field over which the mouse pointer was positioned when the user pressed a button. It is an integer scalar or vector, and a member of `Y`.

`EXIT_CONTEXT[8 9]` are integer scalars which specify the row and column position of the mouse pointer within the field `EXIT_CONTEXT[7]` when `⎕SR` terminated.

Source: R←⎕SRC Y

`⎕SRC` returns the script that defines the Class `Y`.

`Y` must be a reference to a Class.
`R` is a vector of character vectors containing the script that was used to define Class `Y`.

```
      )ED ○MyClass

:Class MyClass
     ∇ Make Name
       :Implements Constructor
       ⎕DF Name
     ∇
:EndClass ⍝ MyClass

      Z←⎕SRC MyClass
      ⍴Z
6
      ⍴¨Z
 14  15  29  14  5  19
      6 1⍴Z
 :Class MyClass
     ∇ Make Name
       :Implements Constructor
       ⎕DF Name
     ∇
 :EndClass ⍝ MyClass
```

State Indicator Stack: R←⎕STACK

R is a two-column matrix, with one row per entry in the State Indicator.

Column 1 :	⎕OR form of user defined functions or operators on the State Indicator. Null for entries that are not user defined functions or operators.
Column 2 :	Indication of the type of the item on the stack.

space	user defined function or operator
⍎	execute level
⎕	evaluated input
*	desk calculator level
⎕DQ	in callback function
other	primitive operator

Example

```
      )SI
PLUS[2]*
.
MATDIV[4]
FOO[1]*
⍎

      ⎕STACK
         *
∇PLUS
         .
∇MATDIV
         *
∇FOO
         ⍎
         *

      ⍴⎕STACK
8 2

      (⍴⎕LC)=⍴⎕STACK
0
```

Pendent defined functions and operators may be edited in Dyalog APL with no resulting SI damage. However, only the visible definition is changed; the pendent version on the stack is retained until its execution is complete. When the function or operator is displayed, only the visible version is seen. Hence ⎕STACK is a tool which allows the user to display the form of the actual function or operator being executed.

Example

To display the version of MATDIV currently pendent on the stack:

```
      ⊃⎕STACK[4;1]
     ∇ R←A MATDIV B
[1]    ⍝ Divide matrix A by matrix B
[2]     C←A⌹B
[3]    ⍝ Check accuracy
[4]     D←⌊0.5+A PLUS.TIMES B
     ∇
```

State of Object: R←⎕STATE Y

Y must be a simple character scalar or vector which is taken to be the name of an APL object. The result returned is a nested vector of 4 elements as described below. ⎕STATE supplies information about shadowed or localised objects that is otherwise unobtainable.

1⊃R	Boolean vector, element set to 1 if and only if this level shadows Y. Note: (⍴1⊃R)=⍴⎕LC
2⊃R	Numeric vector giving the stack state of this name as it entered this level: Note: (⍴2⊃R)=⍴⎕LC

0	not on stack
1	suspended
2	pendent (may also be suspended)
3	active (may also be pendent or suspended)

3⊃R	Numeric vector giving the name classification of Y as it entered this level. Note: (⍴3⊃R)=+/1⊃R
4⊃R	Vector giving the contents of Y before it was shadowed at this level. Note: (⍴4⊃R)=+/0≠3⊃R

Example

```
      ⎕FMT∘⎕OR¨'FN1' 'FN2' 'FN3'
    ∇ FN1;A;B;C         ∇ FN2;A;C               ∇ FN3;A
[1]   A←1          [1]   A←'HELLO'         [1]   A←100
[2]   B←2          [2]   B←'EVERYONE'      [2]   ∘
[3]   C←3          [3]   C←'HOW ARE YOU?'      ∇
[4]   FN2          [4]   FN3
    ∇                   ∇

      )SI
FN3[2]*
FN2[4]
FN1[4]

      ⎕STATE 'A'
 1 1 1  0 0 0  2 2 0   HELLO  1

      R←⎕STATE '⎕TRAP'
```

Set Stop: {R}←X ⎕STOP Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. X must be a simple non-negative integer scalar or vector. R is a simple integer vector of non-negative elements. X identifies the numbers of lines in the function or operator named by Y on which a stop control is to be placed. Numbers outside the range of line numbers in the function or operator (other than 0) are ignored. The number 0 indicates that a stop control is to be placed immediately prior to exit from the function or operator. If X is empty, all existing stop controls are cancelled. The value of X is independent of ⎕IO.

R is a vector of the line numbers on which a stop control has been placed in ascending order. The result is suppressed unless it is explicitly used or assigned.

Examples

```
      +(0,⍳10) ⎕STOP 'FOO'
0 1
```

Existing stop controls in the function or operator named by Y are cancelled before new stop controls are set:

```
      +1 ⎕STOP 'FOO'
1
```

All stop controls may be cancelled by giving `X` an empty vector:

```
      ⍴'' ⎕STOP 'FOO'
0

      ⍴⍬ ⎕STOP 'FOO'
0
```

Attempts to set stop controls in a locked function or operator are ignored.

```
      ⎕LOCK'FOO'

      +0 1 ⎕STOP'FOO'
```

The effect of `⎕STOP` when a function or operator is invoked is to suspend execution at the beginning of any line in the function or operator on which a stop control is placed immediately before that line is executed, and immediately before exiting from the function or operator if a stop control of 0 is set. Execution may be resumed by a branch expression. A stop control interrupt (1001) may also be trapped - see the `⎕TRAP` system variable.

Example

```
      ⎕FX'R←FOO' 'R←10'

      0 1 ⎕STOP'FOO'

      FOO
FOO[1]

      R
VALUE ERROR
      R
      ^

      →1
FOO[0]

      R
10

      →⎕LC
10
```

Query Stop: R←⎕STOP Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. R is a simple non-negative integer vector of the line numbers of the function or operator named by Y on which stop controls are set, shown in ascending order. The value 0 in R indicates that a stop control is set immediately prior to exit from the function or operator.

Example

```
      ⎕STOP'FOO'
0 1
```

Set Access Control: R←X ⎕SVC Y

This system function sets access control on one or more shared variables.

Y is a character scalar, vector, or matrix containing names of shared variables. Each name may optionally be paired with its surrogate. If so, the surrogate must be separated from the name by at least one space.

X may be a 4-element boolean vector which specifies the access control to be applied to all of the shared variables named in Y. Alternatively, X may be a 4-column boolean matrix whose rows specify the access control for the corresponding name in Y. X may also be a scalar or a 1-element vector. If so, it treated as if it were a 4-element vector with the same value in each element.

Each shared variable has a current access control vector which is a 4-element boolean vector. A 1 in each of the four positions has the following impact :

[1] You cannot **set** a new value for the shared variable until after an intervening use or set by your partner.

[2] Your partner cannot **set** a new value for the shared variable until after an intervening use or set by you.

[3] You cannot **use** the value of the shared variable until after an intervening set by your partner.

[4] Your partner cannot **use** the value of the shared variable until after an intervening set by you.

The effect of ⎕SVC is to reset the access control vectors for each of the shared variables named in Y by OR-ing the values most recently specified by your partner with the values in X. This means that you cannot reset elements of the control vector which your partner has set to 1.

Note that the initial value of your partner's access control vector is normally 0 0 0 0. However, if it is a non-APL client application that has established a hot DDE link, its access control vector is defined to be 1 0 0 1. This inhibits either partner from setting the value of the shared variable twice, without an intervening use (or set) by the other. This prevents loss of data which is deemed to be desirable from the nature of the link. (An application that requests a hot link is assumed to require every value of the shared variable, and not to miss any). Note that APL's way of inhibiting another application from setting the value twice (without an intervening use) is to delay the acknowledgement of the DDE message containing the second value until the variable has been used by the APL workspace. An application that waits for an acknowledgement will therefore hang until this happens. An application that does not wait will carry on obliviously.

The result R is a boolean vector or matrix, corresponding to the structure of X, which contains the new access control settings. If Y refers to a name which is not a shared variable, or if the surrogate name is mis-spelt, the corresponding value in R is 4⍴0.

Examples

```
      1 0 0 1 ⎕SVC 'X'
1 0 0 1

      1 ⎕SVC 'X EXTNAME'
1 1 1 1

      (2 4⍴1 0 0 1 0 1 1 0) ⎕SVC ↑'ONE' 'TWO'
1 1 1 1
0 1 1 0
```

Query Access Control: R←⎕SVC Y

This system function queries the access control on one or more shared variables.

Y is a character scalar, vector, or matrix containing names of shared variables. Each name may optionally be paired with its surrogate. If so, the surrogate must be separated from the name by at least one space.

If Y specifies a single name, the result R is a boolean vector containing the current effective access control vector. If Y is a matrix of names, R is a boolean matrix whose rows contain the current effective access control vectors for the corresponding row in Y.

For further information, see the preceding section on setting the access control vector.

Example

```
      ⎕SVC 'X'
0 0 0 0
```

Shared Variable Offer: R←X ⎕SVO Y

This system function offers to share one or more variables with another APL workspace or with another application. Shared variables are implemented using Dynamic Data Exchange (**DDE**) and may be used to communicate with any other application that supports this protocol. See *Interface Guide* for further details.

Y is a character scalar, vector or matrix. If it is a vector it contains a name and optionally an external name or surrogate. The first name is the name used internally in the current workspace. The external name is the name used to make the connection with the partner and, if specified, must be separated from the internal name by one or more blanks. If the partner is another application, the external name corresponds to the DDE **item** specified by that application. If the external name is omitted, the internal name is used instead. The internal name must be a valid APL name and be either undefined or be the name of a variable. There are no such restrictions on the content of the external name.

Instead of an external name, Y may contain the special symbol '⍎' separated from the (internal) name by a blank. This is used to implement a mechanism for sending `DDE_EXECUTE` messages, and is described at the end of this section.

If Y is a scalar, it specifies a single 1-character name. If Y is a matrix, each row of Y specifies a name and an optional external name as for the vector case.

The left argument `X` is a character vector or matrix. If it is a vector, it contains a string that defines the **protocol**, the **application** to which the shared variable is to be connected, and the **topic** of the conversation. These three components are separated by the characters `':'` and `'|'` respectively. The protocol is currently always `'DDE'`, but future implementations of Dyalog APL may support additional communications protocols if applicable. If `Y` specifies more than one name, `X` may be a vector or a matrix with one row per row in `Y`.

If the shared variable offer is a general one (server), `X`, or the corresponding row of `X`, should contain `'DDE:'`.

The result `R` is a numeric scalar or vector with one element for each name in `Y` and indicates the "degree of coupling". A value of 2 indicates that the variable is fully coupled (via a warm or hot DDE link) with a shared variable in another APL workspace, or with a DDE item in another application. A value of 1 indicates that there is no connection, or that the second application rejected a warm link. In this case, a transfer of data may have taken place (via a cold link) but the connection is no longer open. Effectively, APL treats an application that insists on a cold link as if it immediately retracts the sharing after setting or using the value, whichever is appropriate.

Examples

```
      'DDE:' ⎕SVO 'X'
1

      'DDE:' ⎕SVO 'X SALES_92'
1

      'DDE:' ⎕SVO ↑'X SALES_92' 'COSTS_92'
1 1

      'DDE:DYALOG|SERV_WS' ⎕SVO 'X'
2

      'DDE:EXCEL|SHEET1' ⎕SVO 'DATA R1C1:R10C12'
2
```

A special syntax is used to provide a mechanism for sending `DDE_EXECUTE` messages to another application. This case is identified by specifying the `'⍎'` symbol in place of the external name. The subsequent assignment of a character vector to a variable shared with the external name of `'⍎'` causes the value of the variable to be transmitted in the form of a `DDE_EXECUTE` message. The value of the variable is then reset to 1 or 0 corresponding to a positive or negative acknowledgement from the partner. In most (if not all) applications, commands transmitted in `DDE_EXECUTE` messages must be enclosed in square brackets `[]`. For details, see the relevant documentation for the external application.

Examples :

```
      'DDE:EXCEL|SYSTEM' ⎕SVO 'X ⍎'
2

      X←'[OPEN("c:\mydir\mysheet.xls")]'
      X
1

      X←'[SELECT("R1C1:R5C10")]'
      X
1
```

Query Degree of Coupling: R←⎕SVO Y

This system function returns the current degree of coupling for one or more shared variables.

Y is a character scalar, vector or matrix. If it is a vector it contains a shared variable name and optionally its external name or surrogate separated from it by one of more blanks.

If Y is a scalar, it specifies a single 1-character name. If Y is a matrix, each row of Y specifies a name and an optional external name as for the vector case.

If Y specifies a single name, the result R is a 1-element vector whose value 0, 1 or 2 indicates its current degree of coupling. If Y specifies more than one name, R is a vector whose elements indicate the current degree of coupling of the variable specified by the corresponding row in Y. A value of 2 indicates that the variable is fully coupled (via a warm or hot DDE link) with a shared variable in another APL workspace, or with a DDE item in another application. A value of 1 indicates that you have offered the variable but there is no such connection, or that the second application rejected a warm link. In this case, a transfer of data may have taken place (via a cold link) but the connection is no longer open. A value of 0 indicates that the name is not a shared variable.

Examples

```
      ⎕SVO 'X'
2
      ⎕SVO ↑'X SALES' 'Y' 'JUNK'
2 1 0
```

Shared Variable Query: R←⎕SVQ Y

This system function is implemented for compatibility with other versions of APL but currently performs no useful function. Its purpose is to obtain a list of outstanding shared variable offers made to you, to which you have not yet responded.

Using DDE as the communication protocol, it is not possible to implement ⎕SVQ effectively.

Shared Variable Retract Offer: R←⎕SVR Y

This system function terminates communication via one or more shared variables, or aborts shared variable offers that have not yet been accepted.

Y is a character scalar, vector or matrix. If it is a vector it contains a shared variable name and optionally its external name or surrogate separated from it by one of more blanks. If Y is a scalar, it specifies a single 1-character name. If Y is a matrix, each row of Y specifies a name and an optional external name as for the vector case.

If Y specifies a single name, the result R is a 4-element vector indicating the state of the variable prior to retraction. If Y specifies more than one name, R is a matrix whose rows indicate the previous state of the variable specified by the corresponding row in Y.

See ⎕SVS for further information on the possible states of a shared variable.

Shared Variable State: R←⎕SVS Y

This system function returns the current state of one or more shared variables.

Y is a character scalar, vector or matrix. If it is a vector it contains a shared variable name and optionally its external name or surrogate separated from it by one of more blanks. If Y is a scalar, it specifies a single 1-character name. If Y is a matrix, each row of Y specifies a name and an optional external name as for the vector case.

If Y specifies a single name, the result R is a 4-element vector indicating its current state. If Y specifies more than one name, R is a matrix whose rows indicate the current state of the variable specified by the corresponding row in Y.

There are four possible shared variable states :

`0 0 1 1` :	means that you and your partner are both aware of the current value, and neither has since reset it. This is also the initial value of the state when the link is first established.
`1 0 1 0` :	means that you have reset the shared variable and your partner has not yet used it. This state can only occur if both partners are APL workspaces.
`0 1 0 1` :	means that your partner has reset the shared variable but that you have not yet used it.
`0 0 0 0` :	the name is not that of a shared variable.

Examples

```
      ⎕SVS 'X'
0 1 0 1

      ⎕SVS ↑'X SALES' 'Y' 'JUNK'
0 0 1 1
1 0 1 0
0 0 0 0
```

Terminal Control: (⎕ML) R←⎕TC

`⎕TC` is a deprecated feature and is replaced by ⎕UCS (see note).

`⎕TC` is a simple three element vector. If `⎕ML < 3` this is ordered as follows:

`⎕TC[1]` - Backspace
`⎕TC[2]` - Linefeed
`⎕TC[3]` - Newline

Note that `⎕TC≡⎕AV[⎕IO+⍳3]` for `⎕ML< 3` .

If `⎕ML ≥ 3` the order of the elements of `⎕TC` is instead compatible with IBM's APL2:

`⎕TC[1]` - Backspace
`⎕TC[2]` - Newline
`⎕TC[3]` - Linefeed

Elements of ⎕TC beyond 3 are not defined but are reserved.

Note

With the introduction of ⎕UCS in Version 12.0, the use of ⎕TC is discouraged and it is strongly recommended that you generate control characters using ⎕UCS instead. This recommendation holds true even if you continue to use the Classic Edition.

Control Character	**Old**	**New**
Backspace	⎕TC[1]	⎕UCS 8
Linefeed	⎕TC[2] (⎕ML<3) ⎕TC[3] (⎕ML≥3)	⎕UCS 10
Newline	⎕TC[3] (⎕ML<3) ⎕TC[2] (⎕ML≥3)	⎕UCS 13

Thread Child Numbers: R←⎕TCNUMS Y

Y must be a simple array of integers representing thread numbers.

The result R is a simple integer vector of the child threads of each thread of Y.

Examples

```
      ⎕TCNUMS 0
2 3

      ⎕TCNUMS 2 3
4 5 6 7 8 9
```

Get Tokens: {R}←{X} ⎕TGET Y

Y must be a simple integer scalar or vector that specifies one or more tokens, each with a specific non-zero token type, that are to be retrieved from the pool.

X is an optional time-out value in seconds.

Shy result R is a sacalr or vector containing the values of the tokens of type Y that have been retrieved from the token pool.

Note that types of the tokens in the pool may be positive or negative, and the elements of Y may also be positive or negative.

A request (`⎕TGET`) for a *positive* token will be satisifed by the presence of a token in the pool with the same positive or negative type. If the pool token has a positive type, it will be removed from the pool. If the pool token has a negative type, it will remain in the pool. N*egatively* typed tokens will therefore satisfy an infinite number of requests for their positive equivalents. Note that a request for a positive token will remove one if it is present, before resorting to its negative equivalent

A request for a negative token type will only be satisifed by the presence of a negative token type in the pool, and that token will be removed.

If, when a thread calls `⎕TGET`, the token pool satisfies **all** of the tokens specified by `Y`, the function returns immediately with a (shy) result that contains the values associated with the pool tokens. Otherwise, the function will block (wait) until **all** of the requested tokens are present or until a timeout (as specified by `X`) occurs.

For example, if the pool contains only tokens of type 2:

```
    ⎕TGET 2 4        ⍝ blocks waiting for a 4-token ...
```

The `⎕TGET` operation is atomic in the sense that no tokens are taken from the pool until **all** of the requested types are present. While this last example is waiting for a 4-token, other threads could take any of the remaining 2-tokens.

Note also, that repeated items in the right argument are distinct. The following will block until there are at least 3 × 2-tokens in the pool:

```
    ⎕TGET 3/2       ⍝ wait for 3 × 2-tokens ...
```

The pool is administered on a first-in-first-out basis. This is significant only if tokens of the same type are given distinct values. For example:

```
    ⎕TGET ⎕TPOOL              ⍝ empty pool.

    'ABCDE'⎕TPUT¨2 2 3 2 3  ⍝ pool some tokens.

    +⎕TGET 2 3
AC

    +⎕TGET 2 3
BE
```

Timeout is signalled by the return of an empty numeric vector `⍬` (zilde). By default, the value of a token is the same as its type. This means that, unless unless you have explicitly set the value of a token to `⍬`, a `⎕TGET` result of `⍬` unambiguously identifies a timeout.

Beware - the following statement will wait forever and can only be terminated by an interrupt.

```
    ⎕TGET 0       ⍝ wait forever ...
```

Note too that if a thread waiting to ⎕TGET tokens is ⎕TKILLed, the thread disappears without removing any tokens from the pool. Conversely, if a thread that has removed tokens from the pools is ⎕TKILLed, the tokens are not returned to the pool.

This Space: R←⎕THIS

⎕THIS returns a reference to the current namespace, i.e. to the space in which it is referenced.

If NC9 is a reference to any object whose name-class is 9, then:

```
      NC9≡NC9.⎕THIS
1
```

Examples

```
      ⎕THIS
#
      'X'⎕NS ''
      X.⎕THIS
#.X
     'F'⎕WC'Form'
     'F.B'⎕WC'Button'
      F.B.⎕THIS
#.F.B

      Polly←⎕NEW Parrot
      Polly.⎕THIS
#.[Parrot]
```

An Instance may use ⎕THIS to obtain a reference to its own Class:

```
    Polly.(⊃⊃⎕CLASS ⎕THIS)
#.Parrot
```

or a function (such as a Constructor or Destructor) may identify or enumerate all other Instances of the same Class:

```
      Polly.(⍴⎕INSTANCES⊃⊃⎕CLASS ⎕THIS)
1
```

Current Thread Identity: R←⎕TID

R is a simple integer scalar whose value is the number of the current thread.

Examples

```
      ⎕TID     ⍝ Base thread number
0

      ⍎&'⎕TID' ⍝ Thread number of async ⍎.
1
```

Kill Thread: {R}←{X}⎕TKILL Y

Y must be a simple array of integers representing thread numbers to be terminated. X is a boolean single, defaulting to 1, which indicates that all descendant threads should also be terminated.

The shy result R is a vector of the numbers of all threads that have been terminated.

The **base thread** 0 is always excluded from the cull.

Examples

```
      ⎕TKILL 0            ⍝ Kill background threads.

      ⎕TKILL ⎕TID         ⍝ Kill self and descendants.

      0 ⎕TKILL ⎕TID       ⍝ Kill self only.

      ⎕TKILL ⎕TCNUMS ⎕TID ⍝ Kill descendants.
```

Current Thread Name: ⎕TNAME

The system variable ⎕TNAME reports and sets the name of the current APL thread. This name is used to identify the thread in the Tracer.

The default value of ⎕TNAME is an empty character vector.

You may set ⎕TNAME to any valid character vector, but it is recommended that control characters (such as ⎕AV[⎕IO]) be avoided.

Example:

```
      ⎕TNAME←'Dylan'
      ⎕TNAME
Dylan
```

Thread Numbers: R←⎕TNUMS

⎕TNUMS reports the numbers of all current threads.

R is a simple integer vector of the base thread and all its living descendants.

Example

```
      ⎕TNUMS
0 2 4 5 6 3 7 8 9
```

Token Pool: R←⎕TPOOL

R is a simple scalar or vector containing the token types for each of the tokens that are currently in the token pool.

The following (⎕ML=0) function returns a 2-column snapshot of the contents of the pool. It does this by removing and replacing all of the tokens, restoring the state of the pool exactly as before. Coding it as a single expression guarantees that snap is atomic and cannot disturb running threads.

```
      snap←{(⎕TGET ⍵){(⍉↑⍵ ⍺){⍺}⍺ ⎕TPUT¨⍵}⍵}

      snap ⎕TPOOL
1       hello world
2                 2
3                 2
2  three-type token
2                 2
```

Put Tokens: {R}←{X} ⎕TPUT Y

Y must be a simple integer scalar or vector of non-zero token types.

X is an optional array of values to be stored in each of the tokens specified by Y.

Shy result R is a vector of thread numbers (if any) unblocked by the ⎕TPUT.

Examples

```
    ⎕TPUT 2 3 2      ⍝ put a 2-token, a 3-token and another
                       2-token into the pool.

    88 ⎕TPUT 2       ⍝ put another 2-token into the pool
                       this token has the value 88.

    'Hello'⎕TPUT ¯4  ⍝ put a ¯4-token into the pool with
                       the value 'Hello'.
```

If X is omitted, the *value* associated with each of the tokens added to the pool is the same as its *type*.

Note that you cannot put a 0-token into the pool; 0-s are removed from Y.

Set Trace: {R}←X ⎕TRACE Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. X must be a simple non-negative integer scalar or vector. R is a simple integer vector of non-negative elements.

X identifies the numbers of lines in the function or operator named by Y on which a trace control is to be placed. Numbers outside the range of line numbers in the function or operator (other than 0) are ignored. The number 0 indicates that a trace control is to be placed immediately prior to exit from the function or operator. The value of X is independent of ⎕IO.

Example

```
      +(0,⍳10) ⎕TRACE'FOO'
0 1
```

Existing trace controls in the function or operator named by Y are cancelled before new trace controls are set:

```
      + 1 ⎕TRACE'FOO'
1
```

All trace controls may be cancelled by giving X an empty vector:

```
      ⍴⍬ ⎕TRACE 'FOO'
0
```

Attempts to set trace controls in a locked function or operator are ignored.

```
      ⎕LOCK 'FOO'
      +1 ⎕TRACE 'FOO'
```

The effect of trace controls when a function or operator is invoked is to display the result of each complete expression for lines with trace controls as they are executed, and the result of the function if trace control 0 is set. If a line contains expressions separated by ⋄, the result of each complete expression is displayed for that line after execution.

The result of a complete expression is displayed even where the result would normally be suppressed. In particular:

1. the result of a branch statement is displayed;
2. the result (*pass-through value*) of assignment is displayed;
3. the result of a function whose result would normally be suppressed is displayed;

For each traced line, the output from ⎕TRACE is displayed as a two element vector, the first element of which contains the function or operator name and line number, and the second element of which takes one of two forms.

1. The result of the line, displayed as in standard output.
2. → followed by a line number.

Example

```
      ⎕VR 'DSL'
     ∇ R←DSL SKIP;A;B;C;D
[1]    A←2×3+4
[2]    B←(2 3⍴'ABCDEF')A
[3]    →NEXT×⍳SKIP
[4]    'SKIPPED LINE'
[5]   NEXT:C←'one' ⋄ D←'two'
[6]   END:R←C D
     ∇

      (0,⍳6) ⎕TRACE 'DSL'

      DSL 1
 DSL[1]  14
 DSL[2]   ABC   14
          DEF
 DSL[3]  →5
 DSL[5]  one
 DSL[5]  two
 DSL[6]   one   two
 DSL[0]   one   two
 one  two
```

Query Trace: R←⎕TRACE Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. R is a simple non-negative integer vector of the line numbers of the function or operator named by Y on which trace controls are set, shown in ascending order. The value 0 in R indicates that a trace control is set to display the result of the function or operator immediately prior to exit.

Example

```
      ⎕TRACE'DSL'
0 1 2 3 4 5 6
```

Trap Event: ⎕TRAP

This is a non-simple vector. An item of ⎕TRAP specifies an action to be taken when one of a set of events occurs. An item of ⎕TRAP is a 2 or 3 element vector whose items are simple scalars or vectors in the following order:

1. an integer vector whose value is one or more event codes selected from the list in the Figure on the following two pages.
2. a character scalar whose value is an action code selected from the letters C, E, N or S.
3. if element 2 is the letter C or E, this item is a character vector forming a valid APL expression or series of expressions separated by ⋄. Otherwise, this element is omitted.

An EVENT may be an APL execution error, an interrupt by the user or the system, a control interrupt caused by the ⎕STOP system function, or an event generated by the ⎕SIGNAL system function.

When an event occurs, the system searches for a trap definition for that event. The most local ⎕TRAP value is searched first, followed by successive shadowed values of ⎕TRAP, and finally the global ⎕TRAP value. Separate actions defined in a single ⎕TRAP value are searched from **left to right**. If a trap definition for the event is found, the defined action is taken. Otherwise, the normal system action is followed.

The ACTION code identifies the nature of the action to be taken when an associated event occurs. Permitted codes are interpreted as follows:

C	Cutback	The state indicator is 'cut back' to the environment in which the ⎕TRAP is locally defined (or to immediate execution level). The APL expression in element 3 of the same ⎕TRAP item is then executed.
E	Execute	The APL expression in element 3 of the same ⎕TRAP item is executed in the environment in which the event occurred.
N	Next	The event is excluded from the current ⎕TRAP definition. The search will continue through further localised definitions of ⎕TRAP.
S	Stop	Stops the search and causes the normal APL action to be taken in the environment in which the event occurred.

Code	Event
0	**Any event** in range 1-999
1	WS FULL
2	SYNTAX ERROR
3	INDEX ERROR
4	RANK ERROR
5	LENGTH ERROR
6	VALUE ERROR
7	FORMAT ERROR
10	LIMIT ERROR
11	DOMAIN ERROR
12	HOLD ERROR
16	NONCE ERROR
18	FILE TIE ERROR
19	FILE ACCESS ERROR
20	FILE INDEX ERROR
21	FILE FULL
22	FILE NAME ERROR
23	FILE DAMAGED
24	FILE TIED
25	FILE TIED REMOTELY
26	FILE SYSTEM ERROR
28	FILE SYSTEM NOT AVAILABLE
30	FILE SYSTEM TIES USED UP
31	FILE TIE QUOTA USED UP
32	FILE NAME QUOTA USED UP
34	FILE SYSTEM NO SPACE
35	FILE ACCESS ERROR - CONVERTING FILE
38	FILE COMPONENT DAMAGED

Figure 6(i): ⎕TRAP Event Codes

See *Chapter 8* for further details.

Code	Event
52	FIELD CONTENTS RANK ERROR
53	FIELD CONTENTS TOO MANY COLUMNS
54	FIELD POSITION ERROR
55	FIELD SIZE ERROR
56	FIELD CONTENTS/TYPE MISMATCH
57	FIELD TYPE/BEHAVIOUR UNRECOGNISED
58	FIELD ATTRIBUTES RANK ERROR
59	FIELD ATTRIBUTES LENGTH ERROR
60	FULL-SCREEN ERROR
61	KEY CODE UNRECOGNISED
62	KEY CODE RANK ERROR
63	KEY CODE TYPE ERROR
70	FORMAT FILE ACCESS ERROR
71	FORMAT FILE ERROR
72	NO PIPES
76	PROCESSOR TABLE FULL
84	TRAP ERROR
90	EXCEPTION
92	TRANSLATION ERROR
200-499	Reserved for distributed auxiliary processors
500-999	User-defined events
1000	**Any event** in range 1001-1006
1001	Stop vector
1002	Weak interrupt
1003	INTERRUPT
1005	EOF INTERRUPT
1006	TIMEOUT
1007	RESIZE (Dyalog APL/X, Dyalog APL/W)
1008	DEADLOCK

Figure 6(i): `⎕TRAP` Event Codes (Continued)

See `:Trap` as an alternative 'control structured' error trapping mechanism.

Examples

```
      ⎕TRAP←⊂(3 4 5) 'E' 'ERROR' ⋄ ρ⎕TRAP
1

      ⎕TRAP
  3 4 5 E ERROR
```

Items may be specified as scalars. If there is only a single trap definition, it need not be enclosed. However, the value of `⎕TRAP` will be rigorously correct:

```
      ⎕TRAP←11 'E' '→LAB'

      ⎕TRAP
  11 E →ERR

      ρ⎕TRAP
1
```

The value of `⎕TRAP` in a clear workspace is an empty vector whose prototype is `0ρ(⍬ '' '')`. A convenient way of cancelling a `⎕TRAP` definition is:

```
      ⎕TRAP←0ρ⎕TRAP
```

Event codes 0 and 1000 allow all events in the respective ranges 1-999 and 1000-1006 to be trapped. Specific event codes may be excluded by the `N` action (which must precede the general event action):

```
      ⎕TRAP←(1 'N')(0 'E' '→GENERR')
```

The 'stop' action is a useful mechanism for cancelling trap definitions during development of applications.

The 'cut-back' action is useful for returning control to a known point in the application system when errors occur. The following example shows a function that selects and executes an option with a general trap to return control to the function when an untrapped event occurs:

```
     ∇ SELECT;OPT;⎕TRAP
[1]    ⍝ Option selection and execution
[2]    ⍝ A general cut-back trap
[3]     ⎕TRAP←(0 1000)'C' '→ERR'
[4]    INP:⍞←'OPTION : ' ⋄ OPT←(OPT≠' ')/OPT←9↓⍞
[5]     →EX⍴⍨(⊂OPT)∊Options ⋄ 'INVALID OPTION' ⋄ →INP
[6]    EX:⍎OPT ⋄ →INP
[7]    ERR:ERROR∆ACTION ⋄ →INP
[8]    END:
     ∇
```

User-defined events may be signalled through the `⎕SIGNAL` system function. A user-defined event (in the range 500-999) may be trapped explicitly, or implicitly by the event code 0.

Example

```
      ⎕TRAP←500 'E' '''USER EVENT 500 - TRAPPED'''

      ⎕SIGNAL 500
USER EVENT 500 - TRAPPED
```

Token Requests: R←⎕TREQ Y

Y is a simple scalar or vector of thread numbers.

R is a vector containing the concatenated token requests for all the threads specified in Y. This is effectively the result of catenating all of the right arguments together for all threads in Y that are currently executing ⎕TGET.

Example

```
      ⎕TREQ ⎕TNUMS    ⍝ tokens required by all threads.
```

Time Stamp: R←⎕TS

This is a seven element vector which identifies the clock time set on the particular installation as follows:

⎕TS[1]	- Year
⎕TS[2]	- Month
⎕TS[3]	- Day
⎕TS[4]	- Hour
⎕TS[5]	- Minute
⎕TS[6]	- Second
⎕TS[7]	- Millisecond

Example

```
      ⎕TS
1989 7 11 10 42 59 123
```

Note that on some systems, where time is maintained only to the nearest second, a zero is returned for the seventh (millisecond) field.

Wait for Threads to Terminate: R←⎕TSYNC Y

Y must be a simple array of thread numbers.

If Y is a simple scalar, R is an array, the result (if any) of the thread.

If Y is a simple non-scalar, R has the same shape as Y, and result is an array of enclosed thread results.

The interpreter detects a potential deadlock if a number of threads wait for each other in a cyclic dependency. In this case, the thread that attempts to cause the deadlock issues error number 1008: DEADLOCK.

Examples

```
      dup←{⍵ ⍵}         ⍝ Duplicate

      ⎕←dup&88          ⍝ Show thread number
11
88 88

      ⎕TSYNC dup&88     ⍝ Wait for result
88 88

      ⎕TSYNC,dup&88
 88 88

      ⎕TSYNC dup&1 2 3
 1 2 3  1 2 3

      ⎕TSYNC dup&¨1 2 3
 1 1  2 2  3 3

      ⎕TSYNC ⎕TID       ⍝ Wait for self
DEADLOCK
      ⎕TSYNC ⎕TID
      ^

      ⎕EN
1008
```

Unicode Convert: R←{X} ⎕UCS Y

⎕UCS converts (Unicode) characters into integers and vice versa.

The optional left argument X is a character vector containing the name of a variable-length Unicode encoding scheme which must be one of:

- 'UTF-8'
- 'UTF-16'
- 'UTF-32'

If not, a DOMAIN ERROR is issued.

If X is omitted, Y is a simple character or integer array, and the result R is a simple integer or character array with the same rank and shape as Y.

If X is specified, Y must be a simple character or integer vector, and the result R is a simple integer or character vector.

Monadic ⎕UCS

Used monadically, ⎕UCS simply converts characters to Unicode code points and vice-versa.

With a few exceptions, the first 256 Unicode code points correspond to the ANSI character set.

```
      ⎕UCS 'Hello World'
72 101 108 108 111 32 87 111 114 108 100

      ⎕UCS 2 11⍴72 101 108 108 111 32 87 111 114 108 100
Hello World
Hello World
```

The code points for the Greek alphabet are situated in the 900's:

```
      ⎕UCS 'καλημέρα ελλάδα'
954 945 955 951 956 941 961 945 32 949 955 955 940 948 945
```

Thanks to work done by the APL Standards committee in the previous millennium, Unicode also contains the APL character set. For example:

```
      ⎕UCS 123 40 43 47 9077 41 247 9076 9077 125
{(+/⍵)÷⍴⍵}
```

Dyadic ⎕UCS

Dyadic ⎕UCS is used to translate between Unicode characters and one of three standard variable-length Unicode encoding schemes, UTF-8, UTF-16 and UTF-32. These represent a Unicode character string as a vector of 1-byte (UTF-8), 2-byte (UTF-16) and 4-byte (UTF-32) signed integer values respectively.

```
      'UTF-8' ⎕UCS 'ABC'
65 66 67
      'UTF-8' ⎕UCS 'ABCÆØÅ'
65 66 67 195 134 195 152 195 133
      'UTF-8' ⎕UCS 195 134, 195 152, 195 133
ÆØÅ

      'UTF-8' ⎕UCS 'γεια σου'
206 179 206 181 206 185 206 177 32 207 131 206 191 207 133
      'UTF-16' ⎕UCS 'γεια σου'
947 949 953 945 32 963 959 965
      'UTF-32' ⎕UCS 'γεια σου'
947 949 953 945 32 963 959 965
```

Because integers are *signed*, numbers greater than 127 will be represented as 2-byte integers (type 163), and are thus not suitable for writing directly to a native file. To write the above data to file, the easiest solution is to use ⎕UCS to convert the data to 1-byte characters and append this data to the file:

```
      (⎕UCS 'UTF-8' ⎕UCS 'ABCÆØÅ') ⎕NAPPEND tn
```

Note regarding UTF-16: For most characters in the first plane of Unicode (0000-FFFF), UTF-16 and UCS-2 are identical. However, UTF-16 has the potential to encode all Unicode characters, by using more than 2 bytes for characters outside plane 1.

```
      'UTF-16' ⎕UCS 'ABCÆØÅ⍒⍋'
65 66 67 198 216 197 9042 9035
      ⎕←unihan←⎕UCS (2×2*16)+⍳3 ⍝ x20001-x20003[2]
𠀁𠀂𠀃
      'UTF-16' ⎕UCS unihan
55360 56321 55360 56322 55360 56323
```

Translation Error

⎕UCS will generate TRANSLATION ERROR (event number 92) if the argument cannot be converted or, in the Classic Edition, if the result is not in ⎕AV.

[2] See for example http://unicode.org/cgi-bin/GetUnihanData.pl?codepoint=20001

Using (Microsoft .Net Search Path): ⎕USING

⎕USING specifies a list of Microsoft .Net Namespaces that are to be searched for a reference to a .Net class.

⎕USING is a vector of character vectors, each element of which specifies the name of a .Net Namespace followed optionally by a comma (,) and the Assembly in which it is to be found.

If the Assembly is defined in the *global assembly cache*, you need only specify its name. If not, you must specify a full or relative pathname.

If the Microsoft .Net Framework is installed, the System namespace in `mscorlib.dll` is automatically loaded when Dyalog APL starts. To access this namespace, it is not necessary to specify the name of the Assembly.

⎕USING has *namespace scope*. If the local value of ⎕USING is anything other than empty, and you reference a name that would otherwise generate a VALUE ERROR, APL searches the list of .Net Namespaces and Assemblies specified by ⎕USING for a class of that name. If it is found, an entry for the class is added to the symbol table in the current space and the class is used as specified. Note that subsequent references to that class in the current space will be identified immediately.

If ⎕USING is empty (its default value in a CLEAR WS) no such search is performed.

Note that when you assign a value to ⎕USING, you may specify a simple character vector or a vector of character vectors.

Examples:

```
      ⎕USING←'System'
      DISPLAY ⎕USING
.→---------.
| .→-----. |
| |System| |
| '------' |
'∊---------'

      ⎕USING,←⊂'System.Windows.Forms,System.Windows.Forms.dll'
      ⎕USING,←⊂'System.Drawing,System.Drawing.dll'
```

An Assembly may contain top-level classes which are not packaged into .Net Namespaces. In this case, you omit the Namespace name. For example:

```
      ⎕USING←,⊂',.\LoanService.dll'
```

Verify & Fix Input: R←{X}⎕VFI Y

Y must be a simple character scalar or vector. X is optional. If present, X must be a simple character scalar or vector. R is a nested vector of length two whose first item is a simple logical vector and whose second item is a simple numeric vector of the same length as the first item of R.

Y is the character representation of a series of numeric constants. If X is omitted, adjacent numeric strings are separated by one or more blanks. Leading and trailing blanks and separating blanks in excess of one are redundant and ignored. If X is present, X specifies one or more alternative separating characters. Blanks in leading and trailing positions in Y and between numeric strings separated also by the character(s) in X are redundant and ignored. Leading, trailing and adjacent occurrences of the character(s) in X are not redundant. The character 0 is implied in Y before a leading character, after a trailing character, and between each adjacent pair of characters specified by X.

The length of the items of R is the same as the number of identifiable strings (or implied strings) in Y separated by blank or the value of X. An element of the first item of R is 1 where the corresponding string in Y is a valid numeric representation, or 0 otherwise. An element of the second item of R is the numeric value of the corresponding string in Y if it is a valid numeric representation, or 0 otherwise.

Examples

```
      ⎕VFI '12.1 1E1 1A1 ¯10'
 1 1 0 1  12.1 10 0 ¯10

      ⊃(//⎕VFI'12.1 1E1 1A1 ¯10')
12.1 10 ¯10

      ','⎕VFI'3.9,2.4,,76,'
 1 1 1 1 1  3.9 2.4 0 76 0

      '⋄'⎕VFI'1 ⋄ 2 3 ⋄ 4 '
 1 0 1  1 0 4
      θθ≡⎕VFI''
1
```

Vector Representation: R←⎕VR Y

Y must be a simple character scalar or vector which represents the name of a function or defined operator.

If Y is the name of a defined function or defined operator, R is a simple character vector containing a character representation of the function or operator with each line except the last terminated by the newline character (⎕TC[3]). Its display form is as follows:

1. the header line starts at column 8 with the ∇ symbol in column 6,
2. the line number for each line of the function starts in column 1,
3. the statement contained in each line starts at column 8 except for labelled lines or lines beginning with ⍝ which start at column 7,
4. the header line and statements contain no redundant blanks beyond column 7 except that the ⋄ separator is surrounded by single blanks, control structure indentation is preserved and comments retain embedded blanks as originally defined,
5. the last line shows only the ∇ character in column 6.

If Y is the name of a variable, a locked function or operator, an external function, or is undefined, R is an empty vector.

Example

```
      ⍴V←⎕VR'PLUS'
128

      V
     ∇ R←{A}PLUS B
[1]   ⍝ MONADIC OR DYADIC +
[2]    →DYADIC⍴⍨2=⎕NC'A' ⋄ R←B ⋄ →END
[3]   DYADIC:R←A+B ⋄ →END
[4]   END:
     ∇
```

The definition of ⎕VR has been extended to names assigned to functions by specification (←), and to local names of functions used as operands to defined operators. In these cases, the result of ⎕VR is identical to that of ⎕CR except that the representation of defined functions and operators is as described above.

Example

```
      AVG←MEAN∘,

      +F←⎕VR'AVG'
      ∇ R←MEAN X    ⍝ Arithmetic mean
[1]     R←(+/X)÷⍴X
     ∇ ∘,

      ⍴F
3

      DISPLAY F
.→--------------------------------------------.
| .→-----------------------------------.      |
| |     ∇ R←MEAN X    ⍝ Arithmetic mean| ∘ ,  |
| |[1]    R←(+/X)÷⍴X                   | - -  |
| |     ∇                              |      |
| '------------------------------------'      |
'∊--------------------------------------------'
```

Workspace Available: R←⎕WA

This is a simple integer scalar. It identifies the total available space in the active workspace area given as the number of bytes it could hold.

A side effect of using ⎕WA is an internal reorganisation of the workspace and process memory, as follows:

1. Any un-referenced memory is discarded. This process, known as *garbage collection*, is required because whole cycles of refs can become un-referenced.
2. Numeric arrays are *demoted* to their tightest form. For example, a simple numeric array that happens to contain only values 0 or 1, is demoted or *squeezed* to have a ⎕DR type of 11 (Boolean).
3. All remaining used memory blocks are copied to the low-address end of the workspace, leaving a single free block at the high-address end. This process is known as *compaction*.
4. Workspace above a small amount (1/16 of the configured maximum workspace size) of working memory is returned to the Operating System. On a Windows system, you can see the process size changing by using Task Manager.

Example

```
      ⎕WA
261412
```

Windows Create Object: {R}←{X}⎕WC Y

This system function creates a GUI **object**. `Y` is either a vector which specifies **properties** that determine the new object's appearance and behaviour, or the `⎕OR` of a GUI object that exists or previously existed. `X` is a character vector which specifies the name of the new object, and its position in the object hierarchy.

If `X` is omitted, `⎕WC` attaches a GUI component to the current namespace, retaining any functions, variables and other namespaces that it may contain. Monadic `⎕WC` is discussed in detail at the end of this section.

If `Y` is a nested vector each element specifies a property. The `Type` property (which specifies the class of the object) **must** be specified. Most other properties take default values and need not be explicitly stated. Properties (including `Type`) may be declared either positionally or with a keyword followed by a value. Note that `Type` must always be the first property specified. Properties are specified positionally by placing their values in `Y` in the order prescribed for an object of that type.

If `Y` is a result of `⎕OR`, the new object is a complete copy of the one from which the `⎕OR` was made, including any child objects, namespaces, functions and variables that it contained at that time.

The shy result `R` is the full name (starting #. or `⎕SE.`) of the namespace `X`.

An object's name is specified by giving its full pathname in the object hierarchy. At the top of the hierarchy is the `Root` object whose name is "`.`". Below "`.`" there may be one or more "top-level" objects. The names of these objects follow the standard rules for other APL objects as described in *Chapter 1*.

Names for sub-objects follow the same rules except that the character "`.`" is used as a delimiter to indicate parent/child relationships.

The following are examples of legal and illegal names :

Legal	Illegal
`FORM1`	`FORM 1`
`form_23`	`form#1`
`Form1.Gp1`	`1_Form`
`F1.g2.b34`	`Form+1`

If X refers to the name of an APL variable, label, function, or operator, a DOMAIN ERROR is reported. If X refers to the name of an existing GUI object or namespace, the existing one is replaced by the new one. The effect is the same as if it were deleted first.

If Y refers to a non-existent property, or to a property that is not defined for the type of object X, a DOMAIN ERROR is reported. A DOMAIN ERROR is also reported if a value is given that is inconsistent with the corresponding property. This can occur for example, if Y specifies values positionally and in the wrong order.

A "top-level" object created by ⎕WC whose name is localised in a function/operator header, is deleted on exit from the function/operator. All objects, including sub-objects, can be deleted using ⎕EX.

GUI objects are named **relative** to the current namespace, so the following examples are equivalent:

```
      'F1.B1' ⎕WC 'Button'

is equivalent to :

      )CS F1
#.F1
      'B1' ⎕WC 'Button'
      )CS
#
```

is equivalent to :

```
      'B1' F1.⎕WC 'Button'
```

Examples

```
⍝ Create a default Form called F1

      'F1' ⎕WC 'Form'

⍝ Create a Form with specified properties (by position)
⍝   Caption = "My Application"  (Title)
⍝   Posn    = 10 30  (10% down, 30% across)
⍝   Size    = 80 60  (80% high, 60% wide)

      'F1' ⎕WC 'Form' 'My Application' (10 30)(80 60)

⍝ Create a Form with specified properties (by keyword)
⍝   Caption = "My Application"  (Title)
⍝   Posn    = 10 30  (10% down, 30% across)
⍝   Size    = 80 60  (80% high, 60% wide)

      PROPS←⊂'Type' 'Form'
      PROPS,←⊂'Caption' 'My Application'
      PROPS,←⊂'Posn' 10 30
      PROPS,←⊂'Size' 80 60
      'F1' ⎕WC PROPS

⍝ Create a default Button (a pushbutton) in the Form F1

      'F1.BTN' ⎕WC 'Button'

⍝ Create a pushbutton labelled "Ôk"
⍝ 10% down and 10% across from the start of the FORM
⍝ with callback function FOO associated with EVENT 30
⍝ (this event occurs when the user presses the button)

      'F1.BTN'⎕WC'Button' '&Ok' (10 10)('Event' 30 'FOO')
```

Monadic `⎕WC` is used to *attach* a GUI component to an existing object. The existing object must be a pure namespace or a GUI object. The operation may be performed by changing space to the object or by running `⎕WC` *inside* the object using the *dot* syntax. For example, the following statements are equivalent.

```
      )CS F
#.F
      ⎕WC 'Form'  ⍝ Attach a Form to this namespace


      )CS
#
      F.⎕WC'Form' ⍝ Attach a Form to namespace F
```

Windows Get Property: R←{X}⎕WG Y

This system function returns property values for a GUI object.

X is a namespace reference or a character vector containing the name of the object. Y is a character vector or a vector of character vectors containing the name(s) of the properties whose values are required. The result R contains the current values of the specified properties. If Y specifies a single property name, a single property value is returned. If Y specifies more than one property, R is a vector with one element per name in Y.

If X refers to a non-existent GUI name, a VALUE ERROR is reported. If Y refers to a non-existent property, or to a property that is not defined for the type of object X, a DOMAIN ERROR is reported.

GUI objects are named **relative** to the current namespace. A null value of X (referring to the namespace in which the function is being evaluated) may be omitted. The following examples are equivalent:

```
      'F1.B1' ⎕WG 'Caption'
      'B1' F1.⎕WG 'Caption'
      '' F1.B1.⎕WG 'Caption'
      F1.B1.⎕WG 'Caption'
```

Examples

```
      'F1' ⎕WC 'Form' 'TEST'

      'F1' ⎕WG 'Caption'
TEST

      'F1' ⎕WG 'MaxButton'
1

      'F1' ⎕WG 'Size'
50 50

      DISPLAY 'F1' ⎕WG 'Caption' 'MaxButton' 'Size'
.→-----------------.
| .→---.   .→----. |
| |TEST| 1 |50 50| |
| '----'   '~----' |
'∊-----------------'
```

Windows Child Names: R←{X}⎕WN Y

This system function reports the names of the objects whose parent is `Y`. This is a character scalar or vector containing the name of an existing GUI object.

The optional left argument `X` is a character vector which specifies the `Type` of object to be reported.

The result `R` is a vector of character vectors containing the names of the direct children of `Y` which are of `Type X`, or all of them if `X` is not specified. The names of objects further down the tree are not returned, but can be obtained by recursive use of `⎕WN`.

If `Y` refers to a non-existent GUI name, a `VALUE ERROR` is reported.

Note that `⎕WN` reports **only** those child objects visible from the current thread.

GUI objects are named **relative** to the current namespace. The following examples are equivalent:

```
      ⎕WN 'F1.B1'
      F1.⎕WN 'B1'
      F1.B1.⎕WN ''
```

Examples

```
      ⎕WN'.'
 FORM1  MENU1

      ⎕WN'FORM1'
 CANCEL  GROUP1  MENUBAR  OK

      'Button'⎕WN'FORM1'
 CANCEL  OK
```

Windows Set Property: {X}⎕WS Y

This system function resets property values for a GUI object.

X is a namespace reference or a character vector containing the name of the object. Y defines the property or properties to be changed and the new value or values. If a single property is to be changed, Y is a vector whose first element Y[1] is a character vector containing the property name. If Y is of length 2, Y[2] contains the corresponding property value. However, if the property value is itself a numeric or nested vector, its elements may be may be specified in Y[2 3 4 ...] instead of as a single nested element in Y[2]. If Y specifies more than one property, they may be declared either positionally or with a keyword followed by a value. Properties are specified positionally by placing their values in Y in the order prescribed for an object of that type. Note that the first property in Y must always be specified with a keyword because the Type property (which is expected first) may not be changed using ⎕WS.

If X refers to a non-existent GUI name, a VALUE ERROR is reported. If Y refers to a non-existent property, or to a property that is not defined for the type of object X, or to a property whose value may not be changed by ⎕WS, a DOMAIN ERROR is reported.

GUI objects are named **relative** to the current namespace. A null value of X (referring to the namespace in which the function is being evaluated) may be omitted. The following examples are equivalent:

```
      'F1.B1' ⎕WS 'Caption' '&Ok'
      'B1' F1.⎕WS 'Caption' '&Ok'
      '' F1.B1.⎕WS 'Caption' '&Ok'
      F1.B1.⎕WS 'Caption' '&Ok'
```

Examples

```
      'F1' ⎕WC 'Form'  ⍝ A default Form

      'F1' ⎕WS 'Active' 0

      'F1' ⎕WS 'Caption' 'My Application'

      'F1' ⎕WS 'Posn' 0 0

      'F1' ⎕WS ('Active' 1)('Event' 'Configure' 'FOO')

      'F1' ⎕WS 'Junk' 10
DOMAIN ERROR

      'F1' ⎕WS 'MaxButton' 0
DOMAIN ERROR
```

Workspace Identification: ⎕WSID

This is a simple character vector. It contains the identification name of the active workspace. If a new name is assigned, that name becomes the identification name of the active workspace, provided that it is a correctly formed name.

See page 1 for workspace naming conventions.

It is useful, though not essential, to associate workspaces with a specific directory in order to distinguish workspaces from other files.

The value of ⎕WSID in a clear workspace is 'CLEAR WS'.

Example

```
      ⎕WSID
CLEAR WS

      ⎕WSID←'WS/MYWORK'     (UNIX)

      ⎕WSID←'B:\WS\MYWORK'   (Windows)
```

Window Expose: ⎕WX

⎕WX is a system variable that determines:

a) whether or not the names of properties, methods and events provided by a Dyalog APL GUI object are exposed.
b) certain aspects of behaviour of .Net and COM objects.

The permitted values of ⎕WX are 0, 1, or 3. Considered as a sum of bit flags, the first bit in ⎕WX specifies (a), and the second bit specifies (b).

If ⎕WX is 1 (1st bit is set), the names of properties, methods and events are exposed as reserved names in GUI namespaces and can be accessed directly by name. This means that the same names may not be used for global variables in GUI namespaces.

If ⎕WX is 0, these names are hidden and may only be accessed indirectly using ⎕WG and ⎕WS.

If ⎕WX is 3 (2nd bit is also set) COM and .Net objects adopt the Version 11 behaviour, as opposed to the behaviour in previous versions of Dyalog APL.

Note that it is the value of ⎕WX in the object itself, rather than the value of ⎕WX in the calling environment, that determines its behaviour.

The value of ⎕WX in a clear workspace is defined by the default_wx parameter (see User Guide) which itself defaults to 3.

⎕WX has namespace scope and may be localised in a function header. This allows you to create a utility namespace or utility function in which the exposure of objects is known and determined, regardless of its global value in the workspace.

Extended State Indicator: R←⎕XSI

R is a nested vector of character vectors giving the full path names of the functions or operators in the execution stack. Note that if a function has changed space, its original (home) space is reported, rather than its current one.

Example

In the following, function foo in namespace x has called goo in namespace y. Function goo has then changed space (⎕CS) to namespace z where it has been suspended:

```
      )si
[z] y.goo[2]*
x.foo[1]
```

⎕XSI reports the full path name of each function:

```
      ⎕xsi
 #.y.goo  #.x.foo
```

This can be used for example, to edit all functions in the stack, irrespective of the current namespace by typing: `⎕ed ⎕xsi`

See also ⎕SI.

Set External Variable: X ⎕XT Y

`Y` must be a simple character scalar or vector which is taken to be a variable name. `X` must be a simple character scalar or vector which is taken to be a file reference. The name given by `Y` is identified as an EXTERNAL VARIABLE associated with an EXTERNAL ARRAY whose value may be stored in file identified by `X`. See *User Guide* for file naming conventions under Windows and UNIX.

If `Y` is the name of a defined function or operator, a label or a namespace in the active workspace, a `DOMAIN ERROR` is reported.

Example

```
      'EXT\ARRAY' ⎕XT 'V'
```

If the file reference does not exist, the external variable has no value until a value is assigned:

```
      V
VALUE ERROR
      V
      ^
```

A value assigned to an external variable is stored in file space, not within the workspace:

```
      ⎕WA
2261186

      V←⍳100000

      ⎕WA
2261186
```

There are no specific restrictions placed on the use of external variables. They must conform to the normal requirements when used as arguments of functions or as operands of operators. The essential difference between a variable and an external variable is that an external variable requires only temporary workspace for an operation to accommodate (usually) a part of its value.

Examples

```
      +/V
15

      V[3]←⊂'ABC'

      V
1 2  ABC  4 5

      ⍴¨V
     3
```

Assignment allows the structure or the value of an external variable to be changed without fully defining the external array in the workspace.

Examples

```
      V,←⊂2 4⍴⍳8

      ⍴V
6

      V[6]
1 2 3 4
5 6 7 8

      V[1 2 4 5 6]×←10

      V
10 20  ABC  40 50  10 20 30 40
                   50 60 70 80
```

An external array is (usually) preserved in file space when the name of the external variable is disassociated from the file. It may be re-associated with any valid variable name.

Example

```
      ⎕EX'V'

      'EXT\ARRAY'⎕XT'F'

      F
10 20  ABC  40 50  10 20 30 40
                   50 60 70 80
```

In Unix versions, if X is an empty vector, the external array is associated with a temporary file which is erased when the array is disassociated.

Example

```
      ''⎕XT'TEMP'

      TEMP←⍳10

      +/TEMP×TEMP
385

      ⎕EX'TEMP'
```

An external array may be erased using the native file function: `⎕NERASE`.

In a multi-user environment (UNIX or a Windows LAN) a new file associated with an external array is created with access permission for owner read/write. An existing file is opened for exclusive use (by the owner) if the permissions remain at this level. If the access permissions allow any other users to read and write to the file, the file is opened for shared use. In Unix versions, access permissions may be modified using the appropriate Operating System command, or in Windows using the supplied function `XVAR` from the UTIL workspace.

Query External Variable: R←⎕XT Y

`Y` must be a simple character scalar or vector which is taken to be a variable name. `R` is a simple character vector containing the file reference of the external array associated with the variable named by `Y`, or the null vector if there is no associated external array.

Example

```
      ⎕XT'V'
EXT\ARRAY

      ⍴⎕XT'G'
0
```

CHAPTER 7

System Commands

System commands are **not** executable APL expressions. They provide services or information associated with the workspace and the **external environment**.

Command	**Description**
`)CLASSES`	List classes
`)CLEAR`	Clear the workspace
`)CMD Y`	Execute a Windows Command
`)CONTINUE`	Save a Continue workspace and terminate APL
`)COPY {Y}`	Copy objects from another workspace
`)CS {Y}`	Change current namespace
`)DROP {Y}`	Drop named workspace
`)ED Y`	Edit object(s)
`)ERASE Y`	Erase object(s)
`)EVENTS`	List events of GUI namespace or object
`)FNS {Y}`	List user defined Functions
`)HOLDS`	Display Held tokens
`)LIB {Y}`	List workspaces in a directory
`)LOAD {Y}`	Load a workspace
`)NS {Y}`	Create a global Namespace
`)METHODS`	List methods in GUI namespace or object
`)OBJECTS {Y}`	List global namespaces
`)OBS {Y}`	List global namespaces (alternative form)
`)OFF`	Terminate the APL session
`)OPS {Y}`	List user defined Operators
`)PCOPY {Y}`	Perform Protected Copy of objects
`)PROPS`	List properties of GUI namespace or object
`)RESET`	Reset the state indicator
`{ }` indicates that the parameter(s) denoted by `Y` are optional.	

Figure 7(i) : System Commands

Command	Description
`)SAVE {Y}`	Save the workspace
`)SH {Y}`	Execute a (UNIX) Shell command
`)SI`	State Indicator
`)SINL`	State Indicator with local Name Lists
`)TID {Y}`	Switch current Thread Identity
`)VARS {Y}`	List user defined global Variables
`)WSID {Y}`	Workspace Identification
`)XLOAD Y`	Load a workspace; do not execute `⎕LX`
`{ }` indicates that the parameter(s) denoted by `Y` are optional.	

Figure 7(i) : System Commands

Command Presentation

System commands may be entered from immediate execution mode or in response to the prompt `⎕`: within evaluated input. All system commands begin with the symbol `)`, known as a right parenthesis. All system commands may be entered in upper or lower case. Each command is described in alphabetical order in this chapter.

List Classes:)CLASSES

This command lists the names of APL Classes in the active workspace.

Example:

```
      )CLEAR
clear ws
      )ED ○MyClass

:Class MyClass
    ∇ Make Name
      :Implements Constructor
      ⎕DF Name
    ∇
:EndClass ⍝ MyClass

      )CLASSES
MyClass
      )COPY OO YourClass
.\OO saved Sun Jan 29 18:32:03 2006
      )CLASSES
MyClass YourClass
      ⎕NC 'MyClass' 'YourClass'
9.4 9.4
```

Clear Workspace:)CLEAR

This command clears the active workspace and gives the report "`clear ws`". The active workspace is lost. The name of a clear workspace is `CLEAR WS`. System variables are initialised with their default values as described in *Chapter 5*.

In GUI implementations of Dyalog APL, `)CLEAR` expunges all GUI objects, discards any unprocessed events in the event queue and resets the properties of the `Root` object `'.'` to their default values.

Example

```
      )CLEAR
clear ws
```

Windows Command Processor:)CMD cmd

This command allows Windows Command Processor or UNIX shell commands to be given from APL. `)CMD` is a synonym of `)SH`. Either command may be given in either environment (Windows or UNIX) with exactly the same effect. `)CMD` is probably more natural for the Windows user. This section describes the behaviour of `)CMD` and `)SH` under Windows. See `)SH` for a discussion of the behaviour of these commands under UNIX.

The system functions `⎕CMD` and `⎕SH` provide similar facilities but may be executed from within APL code.

Note that under Windows, you may not execute `)CMD` without a command. If you wish to, you can easily open a new Command Prompt window outside APL.

Example

```
      )CMD DIR

 Volume in drive C has no label
 Directory of C:\PETE\WS

.            <DIR>     5-07-94  3.02p
..           <DIR>     5-07-94  3.02p
SALES    DWS    110092 5-07-94  3.29p
EXPENSES DWS    154207 5-07-94  3.29p
```

If **cmd** issues prompts and expects user input, it is **ESSENTIAL** to explicitly redirect input and output to the console. If this is done, APL detects the presence of a "`>`" in the command line and runs the command processor in a visible window and does not direct output to the pipe. If you fail to do this your system will appear to hang because there is no mechanism for you to receive or respond to the prompt.

Example

```
      )CMD DATE <CON >CON
```
(Command Prompt window appears)
```
Current date is Wed 19-07-1995
Enter new date (dd-mm-yy): 20-07-95
```
(Command Prompt window disappears)

Implementation Notes

The argument of)CMD is simply passed to the appropriate command processor for execution and its output is received using an *unnamed pipe*.

By default,)CMD will execute the string ('cmd.exe /c',Y) where Y is the argument given to)CMD. However, the implementation permits the use of alternative command processors as follows.

Before execution, the argument is prefixed and postfixed with strings defined by the APL parameters CMD_PREFIX and CMD_POSTFIX. The former specifies the name of your command processor and any parameters that it requires. The latter specifies a string which may be required. If CMD_PREFIX is not defined, it defaults to the name defined by the environment variable COMSPEC followed by "\c". If COMSPEC is not defined, it defaults to COMMAND.COM or CMD.EXE as appropriate. If CMD_POSTFIX is not defined, it defaults to an empty vector.

Save Continuation:)CONTINUE

This command saves the active workspace under the name CONTINUE and ends the Dyalog APL session.

When you subsequently start another Dyalog APL session, the CONTINUE workspace is loaded automatically. When a CONTINUE workspace is loaded, the latent expression (if any) is NOT executed.

Note that the values of all system variables (including ⎕SM) and GUI objects are also saved in CONTINUE.

Copy Workspace:)COPY {ws {nms}}

This command brings all or selected global objects `nms` from a stored workspace with the given name. A stored workspace is one which has previously been saved with the system command `)SAVE` or the system function `⎕SAVE`. See page 1 for the rules for specifying a workspace name.

If the list of names is excluded, all defined objects (including namespaces) are copied.

If the workspace name identifies a valid, readable workspace, the system reports the workspace name, "`saved`" and the date and time when the workspace was last saved.

Examples

```
      )COPY WS/UTILITY
WS/UTILITY saved Mon Nov  1 13:11:19 1992

      )COPY TEMP ⎕LX FOO X A.B.C
./TEMP saved Mon Nov  1 14:20:47 1992
not found X
```

Copied objects are defined at the global level in the active workspace. Existing global objects in the active workspace with the same name as a copied object are replaced. If the copied object replaces either a function in the state indicator, or an object that is an operand of an operator in the state indicator, or a function whose left argument is being executed, the original object remains defined until its execution is completed or it is no longer referenced by an operator in the state indicator. If the workspace name is not valid or does not exist or if access to the workspace is not authorised, the system reports `ws not found`.

You may copy an object from a namespace by specifying its full pathname. The object will be copied to the current namespace in the active workspace, losing its original parent and gaining a new one in the process. You may only copy a GUI object into a namespace that is a suitable parent for that object. For example, you could only copy a Group object from a saved workspace if the current namespace in the active workspace is itself a Form, SubForm or Group.

If the workspace name identifies a file that is not a workspace, the system reports `bad ws`.

If the source workspace is too large to be loaded , the system reports `ws too large`.

When copying data between Classic and Unicode Editions, `)COPY` will fail with `TRANSLATION ERROR` if *any* object in the source workspace fails conversion between Unicode and `⎕AV` indices, whether or not that object is specified by `nms`. See `⎕AVU` for further details.

If "ws" is omitted, the file open dialog box is displayed and all objects copied from the selected workspace.

If the list of names is included, the names of system variables may also be included and copied into the active workspace. The global referents will be copied.

If an object is not found in the stored workspace, the system reports `not found` followed by the name of the object.

If the list of names includes the name of:

a) an Instance of a Class but not the Class itself
b) a Class but not a Class upon which it depends
c) an array or a namespace that contains a ref to another namespace, but not the namespace to which it refers

the dependant object(s) **will also be copied** but will be **unnamed** and **hidden**. In such as case, the system will issue a warning message.

For example, if a saved workspace named CFWS contains a Class named `#.CompFile` and an Instance (of `CompFile`) named `icf`,

```
      )COPY CFWS icf
.\CFWS saved Fri Mar 03 10:21:36 2006
copied object created an unnamed copy of class #.CompFile
```

The existence of a hidden copy can be confusing, especially if it is a hidden copy of an object which had a name which is in use in the current workspace. In the above example, if there is a class called `CompFile` in the workspace into which `icf` is copied, the copied instance may *appear* to be an instance of the *visible* `CompFile`, but it will actually be an instance of the hidden `CompFile` - which may have very different (or perhaps worse: very slightly different) characteristics to the named version.

If you copy a Class without copying its Base Class, the Class can be used (it will use the invisible copy of the Base Class), but if you edit the Class, you will either be unable to save it because the editor cannot find the Base Class, or - if there is a visible Class of that name in the workspace - it will be used as the Base Class. In the latter case, the invisible copy which was brought in by `)COPY` will now disappear, since there are no longer any references to it - and if these two Base Classes were different, the behaviour of the derived Class will change (and any changes made to the invisible Base Class since it was copied will be lost).

Change Space:)CS {nm}

`)CS` changes the current space to the **global** namespace `nm`.

If no `name` is given, the system changes to the top level (Root) namespace. If `name` is not the name of a global namespace, the system reports the error message `Namespace does not exist.`

`name` may be either a simple name or a compound name separated by `'.'`, including one of the special names `'#'` (Root) or `'##'` (Parent).

Examples

```
      )CS
#
      )CS X
#.X
      )CS Y.Z
#.X.Y.Z
      )CS ##
#.X.Y
      )CS #.UTIL
#.UTIL
```

Drop Workspace:)DROP {ws}

This command removes the specified workspace from disk storage. See page 1 for information regarding the rules for specifying a workspace name.

If "`ws`" is omitted, a file open dialog box is displayed to elicit the workspace name.

Example

```
      )DROP WS/TEMP
Thu Sep 17 10:32:18 1998
```

Edit Object:)ED nms

`)ED` invokes the Dyalog APL editor and opens an Edit window for each of the objects specified in `nms`.

If a name specifies a new symbol it is taken to be a function/operator. However, if a name is localised in a suspended function/operator but is otherwise undefined, it is assumed to be a vector of character vectors.

The type of a new object may be specified explicitly by preceding its name with an appropriate symbol as follows :

∇	function/operator
→	simple character vector
∊	vector of character vectors
-	character matrix
⍟	Namespace script
○	Class script
∘	Interface

The first object named becomes the top window on the stack. See *User Guide* for details. `)ED` ignores names which specify GUI objects.

Examples

```
      )ED MYFUNCTION

      )ED ∇FOO -MAT ∊VECVEC
```

Erase Object:)ERASE nms

This command erases named global defined objects (functions, operators, variables, namespaces and GUI objects) from the active workspace or current namespace.

If a named object is a function or operator in the state indicator, or the object is an operand of an operator in the state indicator, or the object is a function whose left argument is being executed, the object remains defined until its execution is completed or it is no longer referenced by an operator in the state indicator. However, the name is available immediately for other uses.

If a named object is a GUI object, the object and all its children are deleted and removed from the screen.

If an object is not erased for any reason, the system reports `not found` followed by the name of the object.

Erasing objects such as external functions may have other implications: see the entry for expunge object (`⎕EX`) for details.

Example

```
      )ERASE FOO A ⎕IO
not found ⎕IO
```

List Events:)EVENTS

The `)EVENTS` system command lists the Events that may be generated by the object associated with the current space.

For example:

```
      ⎕CS 'BB' ⎕WC 'BrowseBox'

      )EVENTS
Close   Create  FileBoxCancel   FileBoxOK
```

`)EVENTS` produces no output when executed in a pure (non-GUI) namespace, for example:

```
      ⎕CS 'X' ⎕NS ''
      )EVENTS
```

List Global Defined Functions:)FNS {nm}

This command displays the names of global defined functions in the active workspace or current namespace. Names are displayed in `⎕AV` collation order. If a name is included after the command, only those names starting at or after the given name in collation order are displayed.

Examples

```
      )FNS
ASK DISPLAY GET PUT ZILCH
      )FNS G
GET PUT ZILCH
```

Display Held Tokens:)HOLDS

System command)HOLDS displays a list of tokens which have been acquired or requested by the :Hold control structure.

Each line of the display is of the form:

token: acq req req ...

Where acq is the number of *the* thread that has acquired the token, and req is the number of *a* thread which is requesting it. For a token to appear in the display, a thread (and only one thread) must have acquired it, whereas any number of threads can be requesting it.

Example

Thread 300's attempt to acquire token 'blue' results in a deadlock:

```
300:DEADLOCK
Sema4[1] :Hold 'blue'
        ^

      )HOLDS
blue:   100
green:  200     100
red:    300     200     100
```

- Blue has been acquired by thread 100.
- Green has been acquired by 200 and requested by 100.
- Red has been acquired by 300 and requested by 200 and 100.

The following cycle of dependencies has caused the deadlock:

```
Thread 300 attempts to acquire blue,      300 → blue
which is owned by 100,                      ↑      ↓
which is waiting for red,                 red ←  100
```
which is owned by 300.

List Workspace Library:)LIB {dir}

This command lists the names of Dyalog APL workspaces contained in the given directory.

Example

```
      )LIB WS
MYWORK TEMP
```

If a directory is not given, the workspaces on the user's APL workspace path (`WSPATH`)are listed. In this case, the listing is divided into sections identifying the directories concerned. The current directory is identified as "`.`".

Example

```
      )LIB
.
        PDTEMP  WORK   GRAPHICS
C:\DYALOG\WS
        DISPLAY GROUPS
```

Load Workspace:)LOAD {ws}

This command causes the named stored workspace to be loaded. The current active workspace is lost.

If "ws" is a full or relative pathname, only the specified directory is examined. If not, the APL workspace path (WSPATH as specified in APL.INI) is traversed in search of the named workspace. A stored workspace is one which has previously been saved with the system command)SAVE or the system function ⎕SAVE. If ‘ws’ is omitted, the File Open dialog box is displayed.

If the workspace name is not valid or does not exist or if access to the workspace is not authorised, the system reports "ws not found". If the workspace name identifies a file or directory that is not a workspace, the system reports workspace name "is not a ws". If successfully loaded, the system reports workspace name "saved", followed by the date and time when the workspace was last saved. If the workspace is too large to be loaded into the APL session, the system reports "ws too large". After loading the workspace, the latent expression (⎕LX) is executed unless APL was invoked with the -x option.

If the workspace contains any GUI objects whose Visible property is 1, these objects will be displayed. If the workspace contains a non-empty ⎕SM but does not contain an SM GUI object, the form defined by ⎕SM will be displayed in a window on the screen.

Holding the Ctrl key down while entering a)LOAD command or selecting a workspace from the session file menu now causes the incoming latent expression to be *traced.*

Holding the Shift key down while selecting a workspace from the session file menu will *prevent* execution of the latent expression.

Example

```
      )LOAD SMDEMO
/usr/dyalog/WS/SMDEMO saved Wed Sep 6 21:46:27 1989
Type HOWDEMO for help
```

List Methods:)METHODS

The `)METHODS` system command lists the Methods that apply to the object associated with the current space.

For example:

```
      ⎕CS 'F' ⎕WC 'Form'
      )METHODS
Animate ChooseFont   Detach  GetFocus    GetTextSize Wait
```

`)METHODS` produces no output when executed in a pure (non-GUI) namespace, for example:

```
      ⎕CS 'X' ⎕NS ''
      )METHODS
```

Create Namespace:)NS {nm}

)NS creates a **global** namespace and displays its full name, nm.

nm may be either a simple name or a compound name separated by '.', including one of the special names '#' (Root) or '##' (Parent).

If name does not start with the special Root space identifier '#', the new namespace is created relative to the current one.

If name is already in use for a workspace object other than a namespace, the command fails and displays the error message Name already exists.

If name is an existing namespace, no change occurs.

)NS with no nm specification displays the current namespace.

Examples

```
      )NS
#

      )NS W.X
#.W.X

      )CS W.X
#.W.X

      )NS Y.Z
#.W.X.Y.Z

      )NS
#.W.X
```

List Global Namespaces:)OBJECTS {nm}

This command displays the names of global **namespaces** in the active workspace. Names are displayed in the `⎕AV` collating order. If a name is included after the command, only those names starting at or after the given name in collating order are displayed. Namespaces are objects created using `⎕NS`, `)NS` or `⎕WC` and have name class 9.

Note: `)OBS` can be used as an **alternative** to `)OBJECTS`

Examples

```
      )OBJECTS
FORM1   UTIL    WSDOC   XREF

      )OBS W
WSDOC   XREF
```

List Global Namespaces:)OBS {nm}

This command is the same as the `)OBJECTS` command above.

Sign Off APL:)OFF

This command terminates the APL session, returning to the Operating System command processor or shell.

List Global Defined Operators:)OPS {nm}

This command displays the names of global defined operators in the active workspace or current namespace. Names are displayed in `⎕AV` collation order. If a name is included after the command, only those names starting at or after the given name in collation order are displayed.

Examples

```
      )OPS
AND DOIF DUAL ELSE POWER

      )OPS E
ELSE POWER
```

Protected Copy:)PCOPY {ws {nms}}

This command brings all or selected global objects from a stored workspace with the given name provided that there is no existing global usage of the name in the active workspace. A stored workspace is one which has previously been saved with the system command `)SAVE` or the system function `⎕SAVE`.

`)PCOPY` does not copy `⎕SM`. This restriction may be removed in a later release.

If the workspace name is not valid or does not exist or if access to the workspace is not authorised, the system reports "`ws not found`". If the workspace name identifies a file that is not a workspace, or is a workspace with an invalid version number (one that is greater than the version of the current APL) the system reports "`bad ws`". See page 1 for the rules for specifying a workspace name.

If the workspace name is the name of a valid, readable workspace, the system reports the workspace name, "`saved`", and the date and time that the workspace was last saved.

If the list of names is excluded, all global defined objects (functions and variables) are copied. If an object is not found in the stored workspace, the system reports "`not found`" followed by the name of the object. If an object cannot be copied into the active workspace because there is an existing referent, the system reports "`not copied`" followed by the name of the object.

For further information, see `)COPY`.

Examples

```
      )PCOPY WS/UTILITY
WS/UTILITY saved Mon Nov  1 13:11:19 1993
not copied COPIED IF
not copied COPIED JOIN

      )PCOPY TEMP FOO X
./TEMP saved Mon Nov  1 14:20:47 1993
not found X
```

List Properties:)PROPS

The)PROPS system command lists the Properties of the object associated with the current space.

For example:

```
      ⎕CS 'BB' ⎕WC 'BrowseBox'

      )PROPS
BrowseFor       Caption ChildList       Data    Event
EventList       HasEdit KeepOnClose     MethodList
PropList        StartIn Target  Translate       Type
```

)PROPS produces no output when executed in a pure (non GUI) namespace, for example:

```
      ⎕CS 'X' ⎕NS ''
      )PROPS
```

Reset State Indicator:)RESET

This command cancels all suspensions recorded in the state indicator and discards any unprocessed events in the event queue.

)RESET also performs an internal re-organisation of the workspace and process memory. See ⎕WA (Workspace Availability) for details.

Example

```
      )SI
FOO[1]*
⍎
FOO[1]*

      )RESET

      )SI
```

Save Workspace:)SAVE {ws}

This command compacts (see ⎕WA - Workspace Available for details) and saves the active workspace

The workspace is saved with its state of execution intact. A stored workspace may subsequently be loaded with the system command)LOAD or the system function ⎕LOAD, and objects may be copied from a stored workspace with the system commands)COPY or)PCOPY or the system function ⎕CY.

This command may fail with one of the following error messages:

a) `unacceptable char`
The given workspace name was ill-formed.

b) `not saved this ws is WSID`
An attempt was made to change the name of the workspace for the save, and that workspace already existed.

c) `not saved this ws is CLEAR WS`
The active workspace was `CLEAR WS` and no attempt was made to change the name.

d) `cannot create`
The user does not have access to create the file OR the workspace name conflicts with an existing non-workspace file.

e) `cannot save with windows open`
A workspace may not be saved if trace or edit windows are open.

An existing stored workspace with the same name will be replaced. The active workspace may be renamed by the system command)WSID or the system function ⎕WSID.

After a successful save, the system reports the workspace name, "`saved`", followed by the time and date.

Example

```
      )SAVE MYWORK
./MYWORK saved Thu Sep 17 10:32:20 1998
```

Execute (UNIX) Command:)SH {cmd}

This command allows WINDOWS or UNIX shell commands to be given from APL. `)SH` is a synonym of `)CMD`. Either command may be given in either environment (WINDOWS or UNIX) with exactly the same effect. `)SH` is probably more natural for the UNIX user. This section describes the behaviour of `)SH` and `)CMD` under UNIX. See `)CMD` for a discussion of their behaviour under WINDOWS.

The system commands `⎕SH` and `⎕CMD` provide similar facilities but may be executed from within APL code.

`)SH` allows UNIX shell commands to be given from APL. The argument must be entered in the appropriate case (usually lower-case). The result of the command, if any, is displayed.

In Dyalog APL/X, `)SH` without an argument opens a new window. In non-X versions, `)SH` without an argument causes APL to fork a (Bourne) shell and wait. When the shell is closed ("exit" or Ctrl-d), control returns to APL. See *User Guide* for further information.

Example

```
      )SH ls
EXT
FILES
```

State Indicator:)SI

This command displays the contents of the state indicator in the active workspace. The state indicator identifies those operations which are suspended or pendent for each suspension.

The list consists of a line for each suspended or pendent operation beginning with the most recently suspended function or operator. Each line may be:

1. The name of a defined function or operator, followed by the line number at which the operation is halted, and followed by the `*` symbol if the operation is suspended. The name of a function or operator defined in a namespace other than the root is displayed relative to the root. For example, `UTIL.PRINT`. In addition, the display of a function or operator which has dynamically changed space away from its origin, is prefixed with its current space. For example, `[⎕SE] TRAV`.
2. A primitive operator symbol.
3. The Execute function symbol (`⍎`).
4. The Evaluated Input symbol (`⎕`).
5. The System Function `⎕DQ` or `⎕SR` (occurs when executing a callback function).

Examples

```
      )SI
PLUS[2]*
.
MATDIV[4]
FOO[1]*
⍎
```

This example indicates that at some point function `FOO` was executed and suspended on line 1. Subsequently, function `MATDIV` was invoked, with a function derived from the Inner Product or Outer Product operator (`.`) having defined function `PLUS` as an operand.

In the following, function `foo` in namespace `x` has called `goo` in namespace `y`. Function `goo` has then changed space (`⎕CS`) to namespace `z` where it has been suspended:

```
      )si
[z] y.goo[2]*
x.foo[1]
```

Threads

In a multithreading application, where parent threads spawn child threads, the state indicator assumes the structure of a branching tree. Branches of the tree are represented by indenting lines belonging to child threads. For example:

```
      )SI
·    Calc[1]
&5
·    ·    DivSub[1]
·    &7
·    ·    DivSub[1]
·    &6
·    Div[2]*
&4
Sub[3]
Main[4]
```

Here, `Main` has called `Sub`, which has spawned threads `4` and `5` with functions: `Div` and `Calc`. Function `Div`, after spawning `DivSub` in each of threads `6` and `7`, has been suspended at line [2].

State Indicator & Name List: `)SINL`

This command displays the contents of the state indicator together with local names. The display is the same as for `)SI` (see above) except that a list of local names is appended to each defined function or operator line.

Example

```
      )SINL
PLUS[2]*        B       A       R       DYADIC  END
⋆
MATDIV[4]       R       END     I       J       ⎕TRAP
FOO[1]* R
⍎
```

Thread Identity:)TID {tid}

`)TID` associates the Session window with the specified thread so that expressions that you subsequently execute in the Session are executed in the context of that thread.

If you attempt to `)TID` to a thread that is paused or running, that thread will, if possible, be interrupted by a strong interrupt. If the thread is in a state which it would be inappropriate to interrupt (for example, if the thread is executing an external function), the system reports:

```
      Can't switch, this thread is n
```

If no thread number is given, `)TID` reports the number of the current thread.

Examples

```
      ⍝State indicator
      )si
·   print[1]
&3
·   ·   sub_calc[2]*
·   &2
·   calc[1]
&1

      ⍝Current thread
      )tid
is 2

      ⍝Switch suspension to thread 3
      )tid 3
was 2

      ⍝State indicator
      )si
·   print[1]*
&3
·   ·   sub_calc[2]
·   &2
·   calc[1]
&1

      ⍝Attempt to switch to pendent thread 1
      )tid 1
Can't switch, this thread is 3
```

List Global Defined Variables:)VARS {nm}

This command displays the names of global defined variables in the active workspace or current namespace. Names are displayed in ⎕AV collation order. If a name is included after the command, only those names starting at or after the given name in collation order are displayed.

Examples

```
      )VARS
A   B   F   TEMP VAR

      )VARS F
F  TEMP  VAR
```

Workspace Identification:)WSID {ws}

This command displays or sets the name of the active workspace.

If a workspace name is not specified, `)WSID` reports the name of the current active workspace. The name reported is the full path name, including directory references.

If a workspace name is given, the current active workspace is renamed accordingly. The previous name of the active workspace (excluding directory references) is reported. See page 1 for the rules for specifying a workspace name.

Examples

```
      )LOAD WS/TEMP
WS/TEMP saved Thu Sep 17 10:32:19 1998

      )WSID
is WS/TEMP

      )WSID WS/KEEP
was WS/TEMP

      )WSID
WS/KEEP
```

Load without Latent Expression:)XLOAD {ws}

This command causes the named stored workspace to be loaded. The current active workspace is lost.

`)XLOAD` is identical in effect to `)LOAD` except that `)XLOAD` does **not** cause the expression defined by the latent expression `⎕LX` in the saved workspace to be executed.

CHAPTER 8

Error Messages

The error messages reported by APL are described in this chapter. Standard APL messages that provide information or report error conditions are summarised in Figure 8(i) and described in alphabetical order.

APL also reports messages originating from the Operating System (WINDOWS or UNIX) which are summarised in Figure 8(ii). Only those Operating System error messages that might occur through normal usage of APL operations are described in this manual. Other messages could occur as a direct or indirect consequence of using the Operating System interface functions `⎕CMD` and `⎕SH` or system commands `)CMD` and `)SH`, or when a non-standard device is specified for the system functions `⎕ARBIN` or `⎕ARBOUT`. Refer to the WINDOWS or UNIX reference manual for further information about these messages.

Most errors may be trapped using the system variable `⎕TRAP`, thereby retaining control and inhibiting the standard system action and error report. For convenience, Figures 7(i) and 7(ii) identify the error code for trappable errors. The error code is also identified in the heading block for each error message when applicable.

See *User Guide* for a full description of the Error Handling facilities in Dyalog APL.

Standard Error Action

The standard system action in the event of an error or interrupt whilst executing an expression is to suspend execution and display an error report. If necessary, the state indicator is cut back to a statement such that there is no halted locked function visible in the state indicator.

The error report consists of up to three lines

1. The error message, preceded by the symbol ⍎ if the error occurred while evaluating the Execute function.

2. The statement in which the error occurred (or expression being evaluated by the Execute function), preceded by the name of the function and line number where execution is suspended unless the state indicator has been cut back to immediate execution mode. If the state indicator has been cut back because of a locked function in execution, the displayed statement is that from which the locked function was invoked.

3. The symbol ^ under the last referenced symbol or name when the error occurred. All code to the right of the ^ symbol in the expression will have been evaluated.

Examples

```
      X PLUS U
VALUE ERROR
      X PLUS U
             ^
      FOO
INDEX ERROR
FOO[2] X←X+A[I]
           ^

      CALC
⍎DOMAIN ERROR
CALC[5] ÷0
        ^
```

Error Code	Report
	bad ws
	cannot create <u>name</u>
	clear ws
	copy incomplete
1008	DEADLOCK
	defn error
11	DOMAIN ERROR
1005	EOF INTERRUPT
90	EXCEPTION
52	FIELD CONTENTS RANK ERROR
53	FIELD CONTENTS TOO MANY COLUMNS
54	FIELD POSITION ERROR
55	FIELD SIZE ERROR
56	FIELD CONTENTS/TYPE MISMATCH
57	FIELD TYPE/BEHAVIOUR UNRECOGNISED
58	FIELD ATTRIBUTES RANK ERROR
59	FIELD ATTRIBUTES LENGTH ERROR
60	FULL-SCREEN ERROR
61	KEY CODE UNRECOGNISED
62	KEY CODE RANK ERROR
63	KEY CODE TYPE ERROR
70	FORMAT FILE ACCESS ERROR
71	FORMAT FILE ERROR
19	FILE ACCESS ERROR
35	FILE ACCESS ERROR - CONVERTING FILE
38	FILE COMPONENT DAMAGED
23	FILE DAMAGED
21	FILE FULL
20	FILE INDEX ERROR
22	FILE NAME ERROR
32	FILE NAME QUOTA USED UP
26	FILE SYSTEM ERROR
34	FILE SYSTEM NO SPACE
28	FILE SYSTEM NOT AVAILABLE
30	FILE SYSTEM TIES USED UP
18	FILE TIE ERROR
24	FILE TIED
25	FILE TIED REMOTELY
31	FILE TIE QUOTA USED UP

Figure 8(i) : APL Error Messages

Error Code	Report
7	FORMAT ERROR
	incorrect command
12	HOLD ERROR
3	INDEX ERROR
	insufficient resources
1003	INTERRUPT
	is name
5	LENGTH ERROR
10	LIMIT ERROR
16	NONCE ERROR
72	NO PIPES
	name is not a ws
	Name already exists
	Namespace does not exist
	not copied name
	not found name
	not saved this ws is name
76	PROCESSOR TABLE FULL
4	RANK ERROR
1007	RESIZE
	name saved date/time
2	SYNTAX ERROR
	sys error number
1006	TIMEOUT
	too many names
92	TRANSLATION ERROR
84	TRAP ERROR
6	VALUE ERROR
	warning duplicate label
	warning duplicate name
	warning label name present in line 0
	warning pendent operation
	warning unmatched brackets
	warning unmatched parentheses
	was name
1	WS FULL
	ws not found
	ws too large

Figure 8(i) : APL Error Messages (Continued)

Error Code	Report
101	FILE ERROR 1 Not owner
102	FILE ERROR 2 No such file or directory
103	FILE ERROR 3 No such process
104	FILE ERROR 4 Interrupted system call
105	FILE ERROR 5 I/O error
106	FILE ERROR 6 No such device or address
107	FILE ERROR 7 Arg list too long
108	FILE ERROR 8 Exec format error
109	FILE ERROR 9 Bad file number
110	FILE ERROR 10 No children
111	FILE ERROR 11 No more processes
112	FILE ERROR 12 Not enough code
113	FILE ERROR 13 Permission denied
114	FILE ERROR 14 Bad address
115	FILE ERROR 15 Block device required
116	FILE ERROR 16 Mount device busy
117	FILE ERROR 17 File exists
118	FILE ERROR 18 Cross-device link
119	FILE ERROR 19 No such device
120	FILE ERROR 20 Not a directory
121	FILE ERROR 21 Is a directory
122	FILE ERROR 22 Invalid argument
123	FILE ERROR 23 File table overflow
124	FILE ERROR 24 Too many open files
125	FILE ERROR 25 Not a typewriter
126	FILE ERROR 26 Text file busy
127	FILE ERROR 27 File too large
128	FILE ERROR 28 No space left on device
129	FILE ERROR 29 Illegal seek
130	FILE ERROR 30 Read-only file system
131	FILE ERROR 31 Too many links
132	FILE ERROR 32 Broken pipe
133	FILE ERROR 33 Math argument
134	FILE ERROR 34 Result too large

Figure 8(ii) : Typical Operating System Error Messages

Note that operating system error messages **vary** between different operating systems. Dyalog APL reports the error code associated with the operating system error message that it received. Figure 8(iii) gives the messages recieved under Windows 9x.

Error Code	Report
101	FILE ERROR 1 No such file or directory
102	FILE ERROR 2 No such file or directory
103	FILE ERROR 3 Exec format error
105	FILE ERROR 5 Not enough memory
106	FILE ERROR 6 Permission denied
107	FILE ERROR 7 Argument list too big
108	FILE ERROR 8 Exec format error
109	FILE ERROR 9 Bad file number
111	FILE ERROR 11 Too many open files
112	FILE ERROR 12 Not enough memory
113	FILE ERROR 13 Permission denied
114	FILE ERROR 14 Result too large
115	FILE ERROR 15 Resource deadlock would occur
117	FILE ERROR 17 File exists
118	FILE ERROR 18 Cross-device link
122	FILE ERROR 22 Invalid argument
123	FILE ERROR 23 File table overflow
124	FILE ERROR 24 Too many open files
133	FILE ERROR 33 Argument too large
134	FILE ERROR 34 Result too large
145	FILE ERROR 45 Resource deadlock would occur

Figure 8(iii) : Windows Operating System Error Messages

APL Error Messages

There follows an alphabetical list of error messages reported from within Dyalog APL.

bad ws

This report is given when an attempt is made to `)COPY` or `)PCOPY` from a file that is not a valid workspace file. Invalid files include workspaces that were created by a version of Dyalog APL later than the version currently being used.

cannot create name

This report is given when an attempt is made to `)SAVE` a workspace with a name that is either the name of an existing, non-workspace file, or the name of a workspace that the user does not have permission to overwrite or create.

clear ws

This message is displayed when the system command `)CLEAR` is issued.

Example

```
      )CLEAR
clear ws
```

copy incomplete

This report is given when an attempted `)COPY` or `)PCOPY` fails to complete. Reasons include:

- Failure to identify the incoming file as a workspace.
- Not enough active workspace to accommodate the copy.

DEADLOCK 1008

If two threads succeed in acquiring a hold of two different tokens, and then each asks to hold the other token, they will both stop and wait for the other to release its token. The interpreter detects such cases and issues an error (1008) `DEADLOCK`.

This condition can also occur if calls to `⎕TSYNC` cause a cyclic dependency.

defn error

This report is given when either:

- The system editor is invoked in order to edit a function that does not exist, or the named function is pendent or locked, or the given name is an object other than a function.
- The system editor is invoked to define a new function whose name is already active.
- The header line of a function is replaced or edited in definition mode with a line whose syntax is incompatible with that of a header line. The original header line is re-displayed by the system editor with the cursor placed at the end of the line. Back-spacing to the beginning of the line followed by line-feed restores the original header line.

Examples

```
      X←1
      ∇X
defn error

      ∇FOO[0⎕]
[0]   R←FOO
[0]   R←FOO:X
defn error
[0]   R←FOO:X

      ⎕LOCK'FOO'
      ∇FOO[⎕]
defn error
```

DOMAIN ERROR 11

This report is given when either:

- An argument of a function is not of the correct type or its numeric value is outside the range of permitted values or its character value does not constitute valid name(s) in the context.
- An array operand of an operator is not an array, or it is not of the correct type, or its numeric value is outside the range of permitted values. A function operand of an operator is not one of a prescribed set of functions.
- A value assigned to a system variable is not of the correct type, or its numeric value is outside the range of permitted values
- The result produced by a function includes numeric elements which cannot be fully represented.

Examples

```
      1÷0
DOMAIN ERROR
      1÷0
      ^

      (×∘'CAT')2 4 6
DOMAIN ERROR
      (×∘'CAT')2 4 6
      ^

      ⎕IO←5
DOMAIN ERROR
      ⎕IO←5
      ^
```

EOF INTERRUPT 1005

This report is given on encountering the end-of-file when reading input from a file. This condition could occur when an input to APL is from a file.

EXCEPTION 90

This report is given when a Micorosft .Net object throws an exception. Details are available in ⎕EXCEPTION.

FIELD CONTENTS RANK ERROR 52

This report is given if a field content of rank greater than 2 is assigned to ⎕SM.

FIELD CONTENTS TOO MANY COLUMNS 53

This report is given if the content of a numeric or date field assigned to ⎕SM has more than one column.

FIELD POSITION ERROR 54

This report is given if the location of the field assigned to ⎕SM is outside the screen.

FIELD CONTENTS/TYPE MISMATCH 56

This report is given if the field contents assigned to ⎕SM does not conform with the given field type eg. character content with numeric type.

FIELD TYPE/BEHAVIOUR UNRECOGNISED 57

This report is given if the field type or behaviour code assigned to ⎕SM is invalid.

FIELD ATTRIBUTES RANK ERROR 58

This report is given if the current video attribute assigned to ⎕SM is non-scalar but its rank does not match that of the field contents.

FIELD ATTRIBUTES LENGTH ERROR 59

This report is given if the current video attribute assigned to ⎕SM is non-scalar but its dimensions do not match those of the field contents.

FULL-SCREEN ERROR 60

This report is given if the required full screen capabilities are not available to ⎕SM. This report is only generated in UNIX environments.

KEY CODE UNRECOGNISED 61

This report is given if a key code supplied to ⎕SR or ⎕PFKEY is not recognised as a valid code.

KEY CODE RANK ERROR 62

This report is given if a key code supplied to ⎕SR or ⎕PFKEY is not a scalar or a vector.

KEY CODE TYPE ERROR 63

This report is given if a key code supplied to ⎕SR or ⎕PFKEY is numeric or nested; ie. is not a valid key code.

FORMAT FILE ACCESS ERROR 70

This report is given if the date format file to be used by ⎕SM does not exist or cannot be accessed.

FORMAT FILE ERROR 71

This report is given if the date format file to be used by ⎕SM is ill-formed.

FILE ACCESS ERROR 19

This report is given when the user attempts to execute a file system function for which the user is not authorised, or has supplied the wrong passnumber. It also occurs if the file specified as the argument to ⎕FERASE or ⎕FRENAME is not exclusively tied.

Examples

```
      'SALES' ⎕FSTIE 1

      ⎕FRDAC 1
0 4121 0
0 4137 99

      X ⎕FREPLACE 1
FILE ACCESS ERROR
      X ⎕FREPLACE 1
      ^

      'SALES' ⎕FERASE 1
FILE ACCESS ERROR
      'SALES' ⎕FERASE 1
      ^
```

FILE ACCESS ERROR - CONVERTING FILE 35

When a new version of Dyalog APL is used, it may be that improvements to the component file system demand that the internal structure of component files must alter. This alteration is performed by the interpreter on the first occasion that the file is accessed. If the operating system file permissions deny the ability to perform such a restructure, this report is given.

FILE COMPONENT DAMAGED 38

This report is given if an attempt is made to access a component that is not a valid APL object. This will rarely occur, but may happen as a result of a previous computer system failure. Components files may be checked using **`qfsck`**. (See *User Guide.*)

FILE DAMAGED 23

This report is given if a component file becomes damaged. This rarely occurs but may result from a computer system failure. Components files may be checked using **`qfsck`**. (See *User Guide.*)

FILE FULL 21

This report is given if the file operation would cause the file to exceed its file size limit.

FILE INDEX ERROR 20

This report is given when an attempt is made to reference a non-existent component.

Example

```
      ⎕FSIZE 1
1 21 16578 4294967295

      ⎕FREAD 1 34
FILE INDEX ERROR
      ⎕FREAD 1 34
      ^
      ⎕FDROP 1 50
FILE INDEX ERROR
      ⎕FDROP 1 50
      ^
```

FILE NAME ERROR 22

This report is given if:

- the user attempts to `⎕FCREATE` using the name of an existing file.
- the user attempts to `⎕FTIE` or `⎕FSTIE` a non-existent file, or a file that is not a component file.
- the user attempts to `⎕FERASE` a component file with a name other than the EXACT name that was used when the file was tied.

FILE NAME QUOTA USED UP 32

This report is given when the user attempts to execute a file system command that would result in the User's File Name Quota (see *User Guide*) being exceeded.

This can occur with `⎕FCREATE`, `⎕FTIE`, `⎕FSTIE` or `⎕FRENAME` .

FILE SYSTEM ERROR 26

This report is given if the File System Control Block (FSCB) is removed or altered while files are tied.

Contact the System Administrator. If this occurs when a file is share-tied, the file may be damaged. It is therefore advisable to check the integrity of all such files using **`qfsck`**.

See *User Guide* for details.

FILE SYSTEM NO SPACE 34

This report is given if the user attempts a file operation that cannot be completed because there is insufficient disk space.

FILE SYSTEM NOT AVAILABLE 28

This report is given if the File System Control Block (FSCB) is missing or inaccessible. See *User Guide* for details.

FILE SYSTEM TIES USED UP 30

This report is given if the File System Control Block (FSCB) is full. See *User Guide* for details.

FILE TIE ERROR 18

This report is given when the argument to a file system function contains a file tie number used as if it were tied when it is not or as if it were available when it is already tied. It also occurs if the argument to `⎕FHOLD` contains the names of non-existent external variables.

Examples

```
      ⎕FNAMES,⎕FNUMS
SALES  1
COSTS  2
PROFIT 3

      X ⎕FAPPEND 4
FILE TIE ERROR
      X ⎕FAPPEND 4
      ^
      'NEWSALES' ⎕FCREATE 2
FILE TIE ERROR
      'NEWSALES' ⎕FCREATE 2
      ^

      'EXTVFILE' ⎕XT'BIGMAT'
      ⎕FHOLD 'BIGMAT'
FILE TIE ERROR
      ⎕FHOLD 'BIGMAT'
      ^
      ⎕FHOLD⊂'BIGMAT'
```

FILE TIED 24

This report is given if the user attempts to tie a file that is exclusively tied by another task, or attempts to exclusively tie a file that is already share-tied by another task.

FILE TIED REMOTELY 25

This report is given if the user attempts to tie a file that is exclusively tied by another task, or attempts to exclusively tie a file that is already share-tied by another task; and that task is running on other than the user's processor.

FILE TIE QUOTA USED UP 31

This report is given if an attempt is made to ⎕FTIE, ⎕FSTIE or ⎕FCREATE a file when the user already has the maximum number of files tied. (See File Tie Quota , *User Guide*)

FORMAT ERROR 7

This report is given when the format specification in the left argument of system function ⎕FMT is ill-formed.

Example

```
      'A1,1X,I5'⎕FMT CODE NUMBER
FORMAT ERROR
      'A1,1X,I5'⎕FMT CODE NUMBER
       ^
```

(The correct specification should be `'A1,X1,I5'` .)

HOLD ERROR 12

This report is given when an attempt is made to save a workspace using the system function ⎕SAVE if any external arrays or component files are currently held (as a result of a prior use of the system function ⎕FHOLD).

Example

```
      ∇HOLD∆SAVE
[1]    ⎕FHOLD 1
[2]    ⎕SAVE 'TEST'
      ∇

      'FILE' ⎕FSTIE 1

      HOLD∆SAVE
HOLD ERROR
HOLD∆SAVE[2] ⎕SAVE'TEST'
             ^
```

incorrect command

This report is given when an unrecognised system command is entered.

Example

```
      )CLERA
incorrect command
```

INDEX ERROR 3

This report is given when either:

- The value of an index, whilst being within comparison tolerance of an integer, is outside the range of values defined by the index vector along an axis of the array being indexed. The permitted range is dependent on the value of ⎕IO.
- The value specified for an axis, whilst being within comparison tolerance of an integer for a derived function requiring an integer axis value or a non-integer for a derived function requiring a non-integer, is outside the range of values compatible with the rank(s) of the array argument(s) of the derived function. Axis is dependent on the value of ⎕IO.

Examples

```
      A
1 2 3
4 5 6

      A[1;4]
INDEX ERROR
      A[1;4]
      ^

      ↑ [2]'ABC' 'DEF'
INDEX ERROR
      ↑ [2]'ABC' 'DEF'
      ^
```

insufficient resources

This report only occurs in Version 7 and is generated by)COPY or)LOAD. It occurs when the operating system runs out of GUI memory while attempting to recreate GUI objects which have been saved in the workspace. Close some applications and try again.

INTERRUPT 1003

This report is given when execution is suspended by entering a hard interrupt. A hard interrupt causes execution to suspend as soon as possible without leaving the environment in a damaged state.

Example

```
      1 1 2 ⍉(2 100⍴⍳200)∘.|?1000⍴200
(Hard interrupt)
INTERRUPT
      1 1 2 ⍉(2 100⍴⍳200)∘.|?1000⍴200
      ^
```

is name

This report is given in response to the system command `)WSID` when used without a parameter. `name` is the name of the active workspace including directory references given when loaded or named. If the workspace has not been named, the system reports is `CLEAR WS`.

Example

```
      )WSID
is WS/UTILITY
```

LENGTH ERROR 5

This report is given when the shape of the arguments of a function do not conform, but the ranks do conform.

Example

```
      2 3+4 5 6
LENGTH ERROR
      2 3+4 5 6
      ^
```

LIMIT ERROR 10

This report is given when a system limit is exceeded. System limits are installation dependent.

Example

```
      (16⍴1)⍴1
LIMIT ERROR
      (16⍴1)⍴1
      ^
```

NONCE ERROR 16

This report is given when a system function or piece of syntax is not currently implemented but is reserved for future use.

NO PIPES 72

This message applies to the UNIX environment ONLY.

This message is given when the limit on the number of pipes communicating between tasks is exceeded. An installation-set quota is assigned for each task. An associated task may require more than one pipe. The message occurs on attempting to exceed the account's quota when either:

- An APL session is started
- A non-APL task is started by the system function `⎕SH`
- An external variable is used.

It is necessary to release pipes by terminating sufficient tasks before proceeding with the required activity. In practice, the error is most likely to occur when using the system function `⎕SH`.

Examples

```
      'via' ⎕SH 'via'
NO PIPES
      'via' ⎕SH 'via'
      ^

      'EXT/ARRAY' ⎕XT 'EXVAR'
NO PIPES
      'EXT/ARRAY' ⎕XT 'EXVAR'
                  ^
```

name is not a ws

This report is given when the name specified as the parameter of the system commands `)LOAD`, `)COPY` or `)PCOPY` is a reference to an existing file or directory that is not identified as a workspace.

This will also occur if an attempt is made to `)LOAD` a workspace that was `)SAVE`'d using a later version of Dyalog APL.

Example

```
      )LOAD EXT\ARRAY
EXT\ARRAY is not a ws
```

Name already exists

This report is given when an `)NS` command is issued with a name which is already in use for a workspace object other than a namespace.

Namespace does not exist

This report is given when a `)CS` command is issued with a name which is not the name of a global namespace.

not copied name

This report is given for each object named or implied in the parameter list of the system command `)PCOPY` which was not copied because of an existing global referent to that name in the active workspace.

Example

```
      )PCOPY WS/UTILITY A FOO Z
WS/UTILITY saved Mon Nov 1 13:11:19 1993
not copied Z
```

not found name

This report is given when either:

- An object named in the parameter list of the system command `)ERASE` is not erased because it was not found or it is not eligible to be erased.

- An object named in the parameter list (or implied list) of names to be copied from a saved workspace for the system commands `)COPY` or `)PCOPY` is not copied because it was not found in the saved workspace.

Examples

```
      )ERASE ⎕IO
not found ⎕IO

      )COPY WS/UTILITY UND
WS/UTILITY saved Mon Nov 1 13:11:19 1993
not found UND
```

not saved this ws is name

This report is given in the following situations:

- When the system command)SAVE is used without a name, and the workspace is not named. In this case the system reports not saved this ws is CLEAR WS.
- When the system command)SAVE is used with a name, and that name is not the current name of the workspace, but is the name of an existing file.

In neither case is the workspace renamed.

Examples

```
      )CLEAR
      )SAVE
not saved this ws is CLEAR WS

      )WSID JOHND
      )SAVE
      )WSID ANDYS
      )SAVE JOHND
not saved this ws is ANDYS
```

PROCESSOR TABLE FULL 76

This report can only occur in a UNIX environment.

This report is given when the limit on the number of processes (tasks) that the computer system can support would be exceeded. The limit is installation dependent. The report is given when an attempt is made to initiate a further process, occurring when an APL session is started.

It is necessary to wait until active processes are completed before the required task may proceed. If the condition should occur frequently, the solution is to increase the limit on the number of processes for the computer system.

Example

```
      'prefect' ⎕SH 'prefect'
PROCESSOR TABLE FULL
      'prefect' ⎕SH 'prefect'
      ^
```

RANK ERROR 4

This report is given when the rank of an argument or operand does not conform to the requirements of the function or operator, or the ranks of the arguments of a function do not conform.

Example

```
      2 3 + 2 2⍴10 11 12 13
RANK ERROR
      2 3 + 2 2⍴10 11 12 13
      ^
```

RESIZE 1007

This report is given when the user resizes the ⎕SM window. It is only applicable to Dyalog APL/X and Dyalog APL/W.

name saved date/time

This report is given when a workspace is saved, loaded or copied.

date/time is the date and time at which the workspace was most recently saved.

Examples

```
      )LOAD WS/UTILITY
WS/UTILITY saved Fri Sep 11 10:34:35 1998

      )COPY SPACES GEOFF JOHND VINCE
./SPACES saved Wed Sep 30 16:12:56 1998
```

SYNTAX ERROR 2

This report is given when a line of characters does not constitute a meaningful statement. This condition occurs when either:

- An illegal symbol is found in an expression.
- Brackets, parentheses or quotes in an expression are not matched.
- Parentheses in an expression are not matched.
- Quotes in an expression are not matched.
- A value is assigned to a function, label, constant or system constant.
- A strictly dyadic function (or derived function) is used monadically.
- A monadic function (or derived function) is used dyadically.
- A monadic or dyadic function (or derived function) is used without any arguments.
- The operand of an operator is not an array when an array is required.
- The operand of an operator is not a function (or derived function) when a function is required.
- The operand of an operator is a function (or derived function) with incorrect valency.
- A dyadic operator is used with only a single operand.
- An operator is used without any operands.

Examples

```
      A>10)/A
SYNTAX ERROR
      A>10)/A
      ^

      ⊤2 4 8
SYNTAX ERROR
      ⊤2 4 8
      ^

      A.+1 2 3
SYNTAX ERROR
      A.+1 2 3
      ^
```

sys error number

This report is given when an internal error occurs in Dyalog APL.

Under UNIX it may be necessary to enter a hard interrupt to obtain the UNIX command prompt, or even to **kill** your processes from another screen. Under WINDOWS it may be necessary to reboot your PC.

If this error occurs, please submit a fault report to your Dyalog APL distributor.

TIMEOUT 1006

This report is given when the time limit specified by the system variable ⎕RTL is exceeded while awaiting input through character input (⍞) or ⎕SR.

It is usual for this error to be trapped.

Example

```
      ⎕RTL←5 ⋄ ⍞←'RESPOND WITHIN 5 SECONDS: ' ⋄ R←⍞
RESPOND WITHIN 5 SECONDS:
TIMEOUT
      ⎕RTL←5 ⋄ ⍞←'RESPOND WITHIN 5 SECONDS: ' ⋄ R←⍞
                                               ^
```

TRANSLATION ERROR 92

This report is given when the system cannot convert a character from Unicode to an Atomic Vector index or vice versa. Conversion is controlled by the value of `⎕AVU`. Note that this error can occur when you **reference a variable** whose value has been obtained by reading data from a TCPSocket or by calling an external function. This is because in these cases the conversion to/from `⎕AV` is deferred until the value is used.

TRAP ERROR 84

This report is given when a workspace full condition occurs whilst searching for a definition set for the system variable `⎕TRAP` after a trappable error has occurred. It does not occur when an expression in a `⎕TRAP` definition is being executed.

too many names

This report is given by the function editor when the number of distinct names (other than distinguished names beginning with the symbol `⎕`) referenced in a defined function exceeds the system limit of 4096.

VALUE ERROR 6

This report is given when either:

- There is no active definition for a name encountered in an expression.
- A function does not return a result in a context where a result is required.

Examples

```
      X
VALUE ERROR
      X
      ^

      ∇ HELLO
[1]     'HI THERE'
[2]   ∇

      2+HELLO
HI THERE
VALUE ERROR
      2+HELLO
        ^
```

warning duplicate label

This warning message is reported on closing definition mode when one or more labels are duplicated in the body of the defined function. This does not prevent the definition of the function in the active workspace. The value of a duplicated label is the lowest of the line-numbers in which the labels occur.

warning duplicate name

This warning message is reported on closing definition mode when one or more names are duplicated in the header line of the function. This may be perfectly valid. Definition of the function in the active workspace is not prevented. The order in which values are associated with names in the header line is described in *Chapter 2*.

warning pendent operation

This report is given on opening and closing definition mode when attempting to edit a pendant function or operator.

Example

```
[0]   ∇FOO
[1]    GOO
[2]   ∇

[0]   ∇GOO
[1]    ∘
[2]   ∇

      FOO
SYNTAX ERROR
GOO[1] ∘
       ^

      ∇FOO
warning pendent operation
[0]   ∇FOO
[1]    GOO
[2]   ∇
warning pendent operation
```

warning label name present in line 0

This warning message is reported on closing definition mode when one or more label names also occur in the header line of the function. This does not prevent definition of the function in the active workspace. The order in which values are associated with names is described in *Chapter 2*.

warning unmatched brackets

This report is given after adding or editing a function line in definition mode when it is found that there is not an opening bracket to match a closing bracket, or vice versa, in an expression. This is a warning message only. The function line will be accepted even though syntactically incorrect.

Example

```
[3]   A[;B[;2]←0
warning unmatched brackets
[4]
```

warning unmatched parentheses

This report is given after adding or editing a function line in definition mode when it is found that there is not an opening parenthesis to match a closing parenthesis, or vice versa, in an expression. This is a warning message only. The function line will be accepted even though syntactically incorrect.

Example

```
[4]   X←(E>2)^E<10)≠A
warning unmatched parentheses
[5]
```

was name

This report is given when the system command `)WSID` is used with a parameter specifying the name of a workspace. The message identifies the former `name` of the workspace. If the workspace was not named, the given report is was `CLEAR WS`.

Example

```
      )WSID TEMP
was UTILITY
```

WS FULL 1

This report is given when there is insufficient workspace in which to perform an operation. Workspace available is identified by the system constant `⎕WA`.

The maximum workspace size allowed is defined by the environment variable `MAXWS`. See *User Guide* for details.

Example

```
      ⎕WA⍴1.2
WS FULL
      ⎕WA⍴1.2
      ^
```

ws not found

This report is given when a workspace named by the system commands `)LOAD`, `)COPY` or `)PCOPY` does not exist as a file, or when the user does not have read access authorisation for the file.

Examples

```
      )LOAD NOWS
ws not found

      )COPY NOWS A FOO X
ws not found
```

ws too large

This report is given when:

- the user attempts to)LOAD a workspace that needs a greater work area than the maximum that the user is currently permitted.

- the user attempts to)COPY or)PCOPY from a workspace that would require a greater work area than the user is currently permitted if the workspace were to be loaded.

The maximum work area permitted is set using the environment variable MAXWS.

Operating System Error Messages

There follows a numerically sorted list of error messages emanating from a typical operating system and reported through Dyalog APL.

FILE ERROR 1 Not owner 101

This report is given when an attempt is made to modify a file in a way which is forbidden except to the owner or super-user, or in some instances only to a super-user.

FILE ERROR 2 No such file or directory 102

This report is given when a file (which should exist) does not exist, or when a directory in a path name does not exist.

FILE ERROR 5 I/O error 105

This report is given when a physical I/O error occurred whilst reading from or writing to a device, indicating a hardware fault on the device being accessed.

FILE ERROR 6 No such device or address 106

This report is given when a device does not exist or the device is addressed beyond its limits. Examples are a tape which has not been mounted or a tape which is being accessed beyond the end of the tape.

FILE ERROR 13 Permission denied 113

This report is given when an attempt is made to access a file in a way forbidden to the account.

`FILE ERROR 20 Not a directory 120`

This report is given when the request assumes that a directory name is required but the name specifies a file or is not a legal name.

`FILE ERROR 21 Is a directory 121`

This report is given when an attempt is made to write into a directory.

`FILE ERROR 23 File table overflow 123`

This report is given when the system limit on the number of open files is full and a request is made to open another file. It is necessary to wait until the number of open files is reduced. If this error occurs frequently, the system limit should be increased.

`FILE ERROR 24 Too many open files 124`

This report is given when the task limit on the number of open files is exceeded. It may occur when an APL session is started or when a shell command is issued to start an external process through the system command `⎕SH`. It is necessary to reduce the number of open files. It may be necessary to increase the limit on the number of open files to overcome the problem.

`FILE ERROR 26 Text file busy 126`

This report is given when an attempt is made to write a file which is a load module currently in use. This situation could occur on assigning a value to an external variable whose associated external file name conflicts with an existing load module's name.

FILE ERROR 27 File too large 127

This report is given when a write to a file would cause the system limit on file size to be exceeded.

FILE ERROR 28 No space left on device 128

This report is given when a write to a file would exceed the capacity of the device containing the file.

FILE ERROR 30 Read-only file system 130

This report is given when an attempt is made to write to a device which can only be read from. This would occur with a write-protected tape.

Symbolic Index

Alphabetic Index

Dyalog Ltd
South Barn
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom
www.dyalog.com