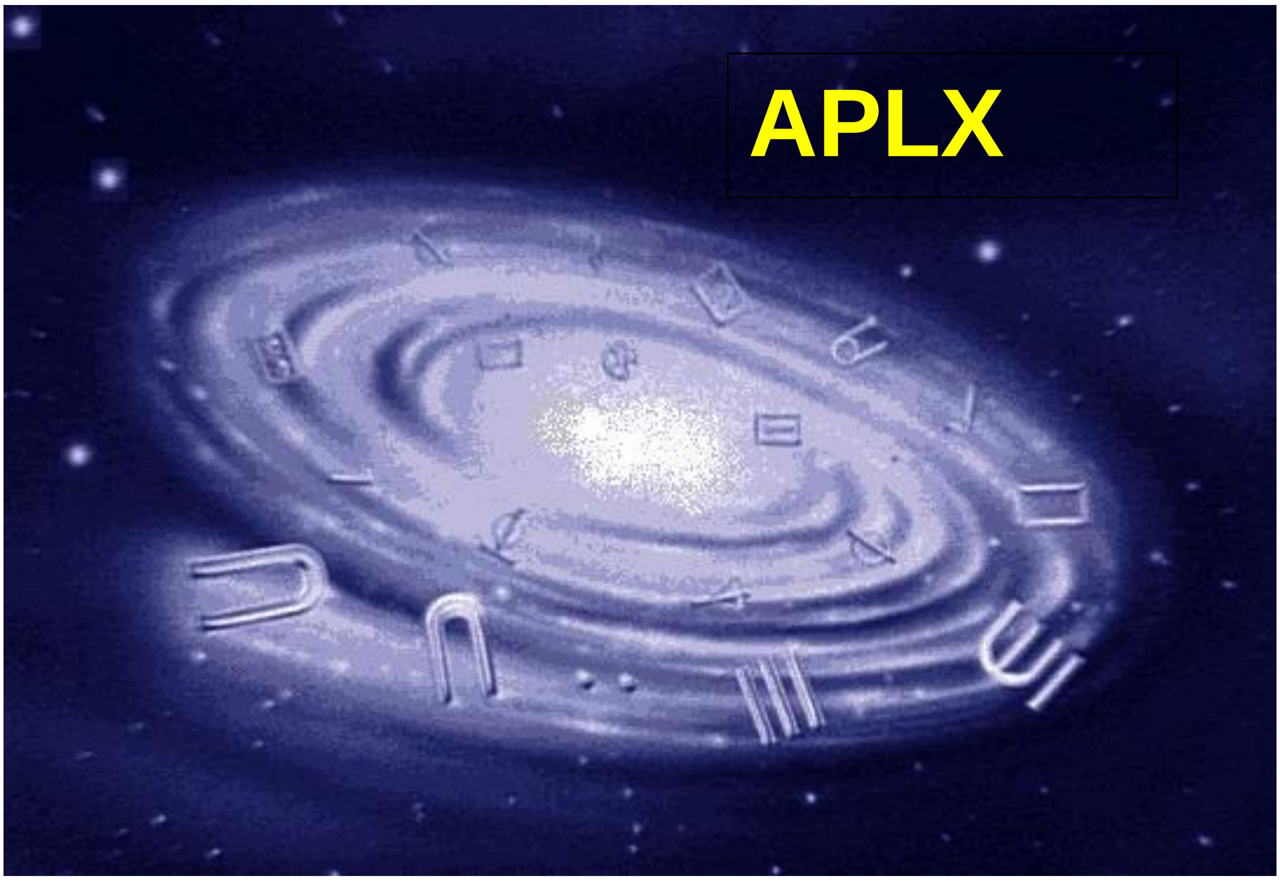


APLX Language Manual

Version 5.0

MICRO APL

Version 5.0 June 2009

Contents

Section 1: APL Fundamentals

The Workspace

The *workspace* is a fundamental concept in APL. It enables you to develop a project as a series of small pieces of program logic. These are organized into *functions, operators* and *classes*, as described below. (For brevity, we sometimes use the term 'function' in this discussion to refer to all three of these). All of these co-exist in the workspace and are instantly available for inspection, amendment, and execution - or for use on another project.

Data of all shapes and sizes (stored in *variables*) can inhabit the same workspace as the functions, and is also instantly available, which greatly facilitates testing. And, of course, the entire collection can be saved on to disk by a single command or menu option.

Functions, operators, and classes can quickly be constructed, tested, strung together in various combinations, and amended or discarded. Most importantly, it is very easy in APL to create test data (including large arrays), for trying out your functions as you develop them. Unlike many traditional programming environments, you do not need to compile and run an entire application just to test a small change you have made - you can test and experiment with individual functions in your workspace. This makes the workspace an ideal prototyping area for 'agile development', and helps explain why APL is sometimes referred to as a 'tool of thought'.

Functions, Operators, Classes

In APL, the term *function* is used for a basic program module. Functions can either be built-in to the APL interpreter (for example, the + function which does addition), or defined by the user as a series of lines of APL code. Functions can take 0, 1 or 2 arguments. For example, when used for addition + takes two arguments (a left argument and a right argument). The arguments to functions are always data (APL arrays). Functions usually act on whole arrays without need for explicit program loops.

An *operator* is like a function in that it takes data arguments, but it also takes either one or two *operands* which can themselves be functions. One of the commonly-used built-in operators is Each (¨). This takes any function as an operand, and applies it to each element of the supplied data arguments. Just as you can define your own functions as a series of lines of APL code, you can also define your own operators.

A *class* is a collection of functions and possibly operators (together known as *methods*), together with data (placed in named *properties* of the class). A class acts as a template from which you can create *objects* (instances of classes), each of which can have its own copy of the class data, but which shares the methods with all other instances of the class. A class can be used to encapsulate the behavior of a specific part of your application.

Workspace size

The workspace size is stated on the screen when you start an APL session. Depending on the workspace size, it is either expressed in 'KB' 'MB' or 'GB', where:

- One 'GB' represents a Gigabyte, approximately a thousand million bytes
- One 'MB' represents a Megabyte, approximately a million bytes
- One 'KB' represents a Kilobyte, approximately a thousand bytes, and
- One byte is (again approximately) the amount of computer memory used to store a single character.

During the session you can find out how much space is free by using the system function `⎕WA`, which stands for Workspace Available.

The maximum size of the workspace depends on how much memory (RAM) you have on your system, and the amount of disk space reserved for virtual memory.

Managing the workspace

There are *system commands* for enquiring about the workspace and doing operations that affect it internally. The most useful of these are mentioned below under the heading 'Internal workspace commands'. (Note that, to distinguish them from names in your program, the names of system commands start with a right parenthesis.)

There are also system commands for copying the current workspace to disk, reloading it into memory and doing other similar operations. These are mentioned below under the heading 'External workspace commands'. You can either type these commands directly, or (on most versions of APLX) use the File menu to load and save workspaces.

Internal workspace commands

At the start of a session, you're given an empty workspace which has the name `CLEAR WS`. At any time you can return to this state by issuing the system command `)CLEAR`. Any variables or functions you have set up in the workspace are wiped out by this command, so if you want to keep them, you should first save the workspace on to a disk.

You can get a list of the variable names in the workspace by using the `)VARS` command. The command `)FNS` produces the equivalent list of user-defined functions, and the command `)OPS` gives the list of user-defined operators. The command `)CLASSES` lists the classes you have defined.

If you don't want to clear the entire workspace, you can get rid of individual items by using the command `)ERASE` followed by the name(s) of the items(s) you want to remove.

External workspace commands

Note: *In practice, you will often use menus to load and save workspaces, rather than typing the system commands described below. For example, rather than typing* `)LOAD`*, you can use the File menu to open a dialog which allows you to select the workspace you want to load.*

A collection of workspaces on a disk, or other storage medium, is a *library*. (It corresponds to a directory or folder in the host operating system). Unless you change the library number associated with each device, the device listed first when you type `⎕MOUNT ''` (see below under 'System Functions') is Library 0, the next one is Library 1, and so on up to Library 9. (In most versions of APLX, you can set up these libraries using the Preferences item of the Tools or APLX menu). Most of the commands in this section can include the number (0, 1 or whatever) to indicate which library the command applies to. If no library number is given, APL assumes that library 0 is intended.

Library 10 is a special case. It contains the utility workspaces and examples supplied as part of the APLX installation.

The use of library numbers is a convenience which helps you organize your workspaces on disk, and saves you from having to enter long path names when referring to them. But if you prefer, you can enter the full path name to a workspace when you load and save it (or use the File menu).

To find out the names of the workspaces which you have already stored in library 0, use the command `)LIB`. To list the workspaces supplied with APLX, use: `)LIB 10`.

You can save the current workspace by simply issuing the command: `)SAVE`. Everything in the workspace is copied on to the disk and the saved workspace is given the same name as the workspace in memory. If you want the saved version to have a different name, you specify the (new) name immediately after the `)SAVE` (e.g. `)SAVE NEWNAME`).

The `)LOAD` command followed by the name of a workspace brings the named workspace back into memory. The workspace already in memory is overwritten.

If you want to bring specific functions or variables into memory, but don't want to overwrite the workspace already there, you can use the `)COPY` command.

You can get rid of a workspace on a disk by using the `)DROP` command.

System variables

What goes on in the workspace is conditioned to some extent by the current settings of system variables. These are built-in variables, whose names begin with '`⎕`'.

Some system variables you may occasionally want to enquire about or (in some cases) alter are:

- `⎕WA` Workspace available: the number of bytes available for use in the workspace.

- ⎕PP Print precision: the number of digits displayed in numeric output. The default setting is 10.
- ⎕PW Print width: the number of characters to the line. On most systems, the default setting is 80 (or the size of the visible window).
- ⎕LX Latent Expression: the expression or user-defined function in this variable is executed when the workspace is loaded. You might, for example, write a function which set things up for you when you started a session and assign its name to ⎕LX. Unless you assign a value to ⎕LX, it's empty.

You can find out the value of a system variable by typing its name. For example, to see the setting of ⎕PP, the variable which determines how many digits are displayed in numeric output, you would type:

```
      ⎕PP
10
```

You can reset the value of most system variables by using the symbol ← . For example, to change ⎕PP from its normal value of 10, to a value of 6, you would type:

```
      ⎕PP ← 6
```

System functions

We've been discussing system variables. System functions can also affect your working environment. The system function ⎕MOUNT, for example, is used to associate operating-system directories with the library identifiers you use in your programs.

Other system functions duplicate tasks performed by system commands. For example, the system function ⎕NL which stands for **name list**, can be used to produce a list of variables, functions, operators, or classes, and the system function ⎕EX can be used to **expunge** individual APL objects. Similar jobs are done by the system commands)VARS)FNS)OPS)CLASSES, and)ERASE.

The difference between system functions and system commands is that system functions are designed for use in user-defined functions, and behave like other functions in that they return results which can be used in APL statements. System commands, on the other hand, are primarily designed for direct execution and can only be included in a user-defined function if quoted as the text argument to the function ⍎ (execute - a function which causes the expression quoted to be executed.)

There are many System Functions and Variables available in APLX. They have other purposes besides control of the workspace; for example they are used for reading and writing files, and for accessing databases, and for doing string-searches using regular expressions.

Data

A data item is composed of numbers, characters, or references to objects. It can be a constant or a variable:

```
231           (constant)
NUM           (variable)
```

System variables are a special class of variable. Their names start with a `⎕`. Normally their initial values are set by the system, eg:

```
⎕PP
```

Data in APL is arranged in *arrays*. An array is a collection of data with a number of dimensions (*rank*) and a number of elements in each dimension (*shape*). Some or all of the elements may themselves be arrays, making the array a *nested* array with a third property, *depth*.

The commonest ranks of array are given special names:

Rank	Name	Dimensions	
0	Scalar	None	(one element only)
1	Vector	1	(elements)
2	Matrix	2	(rows and columns)
3		3	(planes, rows and columns)
4		4	(blocks,planes,rows and columns)

Arrays of up to 63 dimensions are allowed in APLX.

The 'depth' of an array is a measure of the degree of nesting in the array. A simple (non-nested) scalar will have a depth of 0 and an array whose elements are all scalars (character or numeric) is known as a 'simple' array and has a depth of 1. In a nested array, the depth of the array is defined as the depth of the deepest element. The following table shows the way in which the depth of an array may be calculated:

Depth	Description
0	Simple scalar
1	Simple array
2	Deepest element in the array is of depth 1
3	Deepest element in the array is of depth 2
.	
n	Deepest element in the array is of depth n-1

Character Data

Anything enclosed between either single or double quotes is treated as character data (you must use the same type of quote mark to end the string as you use to begin it). This includes the digits, 0 to 9, and any of the symbols on the keyboard. It also includes the invisible character, space. If you have used single quotes to delimit the string, then to include a single quote itself in the character data, type it where it is required, followed immediately by another single quote. (Similarly for double quotes). Alternatively, if you use single-quotes to delimit a string, you can place double-quotes directly in the string without doubling them up, and vice-versa. The single- or double-quote characters used to surround character data are not displayed by APLX.

```
      ALF←'ABC -+= 123'   (The characters in quotes are put in ALF)
      ALF                 (When displayed, the quotes are dropped)
ABC -+= 123

      ⍴ALF                (⍴ is used to ask the size of ALF.
11                         It contains 11 characters including spaces)

      A←'DON''T WALK'     (' entered as ''
      A
DON'T WALK                 Only one is displayed)

      A←"DON'T WALK"      (Alternative using double-quotes.
      A
DON'T WALK                 The result is the same)

      B← '''TIS TRUE '    (' as the first character of a text string)
      B
'TIS TRUE

      B← "'TIS TRUE "     (Alternative using double-quotes)
      B
'TIS TRUE

      NUM←'501'           (Digits included in character data are
      NUM+10               characters rather than numbers and
DOMAIN ERROR               can't be used in arithmetic)

      TABLE←3 6⍴"YES NO"
YES NO                    (Character data can be formed into
YES NO                     matrices. The six characters YES NO
YES NO                     are formed into a matrix of 3 rows and
                           6 columns)
```

Array type & prototype

Any array has a 'type' which is zero for numeric elements and the blank character for character elements. The type of the first element in an array is known as the 'prototype' of the array. The prototype of an array is used in two important areas. Firstly, the prototype of an array is used to determine the structure of an empty array formed from that array. Secondly, the prototype of an array is used as a 'fill' element by those APL functions that can generate extra elements (for example `↑` 'take').

You can see the structure of an array, including its type and prototype, using the `⎕DISPLAY` system function (or the `)DISPLAY` system command). In desktop editions of APLX, you can also invoke a Display Window to show array structure, using the pop-up menu which appears when you right-click (or, under MacOS, click-and-hold) over a variable name. See the description of the `⎕DISPLAY` system function for details of how the structure is shown.

Empty arrays

An empty array is one in which at least one of the dimensions is zero. For example, an empty vector is a vector of length 0. An array with four rows and zero columns is an empty matrix; it has shape (4 by 0), but does not contain any data elements.

A empty vector can most simply be made by one of the expressions shown below. Note that the 'type' of the resultant empty vector can be be numeric or character.

```
      ⎕DISPLAY ⍳0                  ⎕DISPLAY ''
.⊖.                          .⊖.
|0|                          | |
'~'                          '-'
empty numeric vector         empty character vector
```

The alternative expression `0⍴` is often used, and may again create either a numeric or character empty vector when used with the appropriate argument. (Higher dimensional empty arrays may be made by expressions of the form 0 `2⍴..` and so on - see the entry for `⍴` 'reshape').

```
      ⎕DISPLAY 0⍴676               ⎕DISPLAY 0⍴'PETER'
.⊖.                          .⊖.
|0|                          | |
'~'                          '-'
empty numeric vector         empty character vector
```

An empty numeric vector can also be created using `⍬` (Zilde), a primitive constant, which is equivalent to `⍳0` or `0⍴0`.

```
      X←⍬
      ⍴X
0
      X≡0⍴0
1
```

```
      ⎕DISPLAY ⍬      ⍝ Empty numeric vector
┌⊖┐
│0│
└~┘
```

Prototypes of nested arrays

More complex empty arrays result when a nested array is used to generate the empty array. In each case, it is the 'prototype' (derived from the first element of the original array) that dictates the type and structure of the resultant empty array.

```
      ⎕DISPLAY 'ABC' (⍳3)          ⎕DISPLAY 0⍴'ABC' (⍳3)
.→---------------.            .⊖------.
| .→--.   .→----. |           | .→--. |
| |ABC|   |1 2 3| |           | |   | |
| '---'   '~----' |           | '---' |
'∊---------------'            '∊------'
```

The original array (on the left) is a two element nested array whose first element is itself a character vector. The resultant empty vector (the expression on the right) is an empty vector which is itself a nested array with the prototype a length 3 character vector.

```
      ⎕DISPLAY (2 2⍴⍳4) 'ABC'       ⎕DISPLAY 0⍴(2 2⍴⍳4) 'ABC'
.→--------------.             .⊖-------.
| .→---.  .→--. |             | .→---. |
| ↓ 1 2|  |ABC| |             | ↓ 0 0| |
| | 3 4|  '---' |             | | 0 0| |
| '~---'        |             | '~---' |
'∊--------------'             '∊-------'
```

Similar considerations apply above, except that the type of the first element of the original array is numeric, or even, as below, when the first element is mixed.

```
      ⎕DISPLAY (2 2⍴ 1 'K' 2 'J' ) (⍳4)      ⎕DISPLAY 0⍴(2 2⍴ 1 'K' 2 'J') (⍳4)
.→--------------------.                 .⊖--------.
| .→----.  .→-------. |                 | .→----. |
| ↓ 1 K |  |1 2 3 4| |                 | ↓ 0   | |
| | 2 J |  '~-------' |                 | | 0   | |
| '+----'             |                 | '+----' |
'∊--------------------'                 '∊--------'
```

The prototype concept can be used to display the 'type' of an array. In the example below, the array is first enclosed `(⊂)` to form a scalar and an empty vector made from the scalar `(0⍴)`. Finally the `↑` ('first') function removes the additional level of nesting introduced.

```
      ⎕DISPLAY VAR
.→------------------------------------.
| .→-------------------------.   .→--. |
| | .→----.   .→-----------. |   |ABC| |
| | ↓ 1 A |   | .→--.      | |   '---' |
| | | B 2 |   | |1 2|    7 | |         |
| | '+----'   | '~--'      | |         |
| |           '∊-----------' |         |
| '∊-------------------------'         |
'∊------------------------------------'
```

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

The prototype as a fill element

Certain functions require the addition of 'fill' elements to arrays, for example the functions `↑` ('take'), `\` ('expand') and `/` ('replicate'). These function can add extra elements to an existing array; the prototype is used to determine the type and shape of the extra elements.

The fill element depends on the type of the array being extended, as follows:

Type of array	*Fill Element*
Numeric	Zero
Character	Space
Nested or mixed	Prototype or first element, with numbers replaced by zeroes and characters by spaces
Object or Class Reference	The NULL object

```
      5↑1 2 3                 (fill element for simple numeric is 0)
1 2 3 0 0
      ⎕DISPLAY 5↑'ABC'        (fill element for simple character array
                               is blank)
.→----.
|ABC  |
'-----'
```

```
      ⎕DISPLAY VAR            (nested array - vector of length 2 )
.→-----------------------------------.
| .→------------------------.  .→--. |
| | .→----.  .→-----------. |  |ABC| |
| | ↓ 1 A |  | .→--.      | |  '---' |
| | | B 2 |  | |1 2|   7  | |        |
| | '+----'  | '~--'      | |        |
| |          '∊-----------' |        |
| '∊------------------------'        |
'∊-----------------------------------'
```

```
      ⎕DISPLAY 3↑VAR          (prototype used as trailing fill element)
.→----------------------------------------------------------------.
| .→------------------------.  .→--.  .→------------------------. |
| | .→----.  .→-----------. |  |ABC|  | .→----.  .→-----------. | |
| | ↓ 1 A |  | .→--.      | |  '---'  | ↓ 0   |  | .→--.      | | |
| | | B 2 |  | |1 2|   7  | |         | |   0 |  | |0 0|   0  | | |
| | '+----'  | '~--'      | |         | '+----'  | '~--'      | | |
| |          '∊-----------' |         |          '∊-----------' | |
| '∊------------------------'         '∊------------------------' |
'∊----------------------------------------------------------------'
```

```
      ⎕DISPLAY ¯3↑VAR         (prototype used as leading fill element)
.→----------------------------------------------------------------.
| .→------------------------.  .→------------------------.  .→--. |
| | .→----.  .→-----------. |  | .→----.  .→-----------. |  |ABC| |
| | ↓ 0   |  | .→--.      | |  | ↓ 1 A |  | .→--.      | |  '---' |
| | |   0 |  | |0 0|   0  | |  | | B 2 |  | |1 2|   7  | |        |
| | '+----'  | '~--'      | |  | '+----'  | '~--'      | |        |
| |          '∊-----------' |  |          '∊-----------' |        |
| '∊------------------------'  '∊------------------------'        |
'∊----------------------------------------------------------------'
```

Display of arrays

Display of simple, numeric arrays

By default, numeric data is displayed with a space between each successive element of a vector (or dimension in an array). Arrays are displayed in such a way that their structure should be apparent. Vectors are displayed as a line of data. Matrices are displayed with each row of the matrix on successive lines of the screen or printer. For arrays of higher rank, the display is by means of successive matrices. Successive planes are separated by one blank line, successive blocks by two blank lines and so on.

An empty vector displays as one blank line. Empty arrays of rank 2 or more are not displayed.

Display of simple, character arrays

These are displayed without spaces between elements on the row; other rules are the same as those for simple numeric arrays.

Display of simple, mixed arrays

The rules described above apply to simple arrays which are all character or all numeric. Simple, mixed arrays make use of the rule that a column which contains a number is always separated from adjacent columns by at least one blank.

```
      MAT                     (Simple, mixed array, note columns are
 A B   45 C                    separated by spaces)
 D 999 F  1000
      2 4⍴'ABCDEFGH'          (Simple character array - columns not
ABCD                           separated)
EFGH
```

Display of a class or object reference

By default, APLX displays an object reference as the unqualified class name contained in square brackets. Class references are displayed as the class name in curly braces:

```
      )CLASSES
Queue                      ⍝ User-defined APL class
      Queue
{Queue}                    ⍝ Default display of class reference
      QUEUE23←⎕NEW Queue
      QUEUE23              ⍝ Default display of APL object reference
[Queue]
```

You can change the default display for an object by using the `⎕DF` system method.

Display of nested arrays

The display of any nested array is preceded by a leading blank so that nested arrays will be indented one space. It is also followed by a trailing blank.

```
      ⍳5                        (Simple vector)
1 2 3 4 5
      (⍳3) (⍳2)                 (Nested vector indented by one space)
 1 2 3  1 2
```

The other rules for the display of nested arrays are:

- At least one blank between columns with numbers
- No separation between columns with scalar characters
- Numbers right justified on the decimal point
- Character vectors (containing only scalars) left justified
- Columns with text and numbers right justified
- Other nested items displayed with leading and trailing blank for each level of nesting.

For example:

```
      2 3⍴'ABC' 100027 'NAME' 3 'DAT' 27
 ABC 100027 NAME
   3    DAT   27
      3 2⍴'JOHN' 'SMITH' 'ARTHUR' 'JONES' 'WILFRED' 'HART'
 JOHN    SMITH
 ARTHUR  JONES
 WILFRED HART
```

Vector Notation

If an expression contains one or more arrays, then the resultant vector will contain elements which are those arrays. The way in which this type of expression is constructed is known as 'vector notation'. Parentheses or quote characters are used to delimit arrays in vector notation. Alternatively, the expression may contain variable names.

```
      'ABC' 'DEF'                (Two three element character vectors make
 ABC DEF                          a two element nested vector)

      (1 2 3) 'DEF'              (Three element numeric and three element
 1 2 3  DEF                       character vector make a two element nested
                                  vector)
      ⍴1 2 3 'DEF'               (Three numeric scalars and a three element
4                                 character vector make a four element vector)
      ⍴(1 2 3) 'DEF'             (The parentheses force the three numbers
2                                 to be treated as the first element of the
                                  two element result)
      ⍴1 2 3 'D' 'E' 'F'         (Each character is now treated as a scalar
6                                 giving a 6 element mixed result)
      ((1 2) (3 4)) 2 3          (First element of the vector is itself
 1 2  3 4  2 3                    a nested vector - two two element numeric
                                  vectors. The ⎕DISPLAY function clarifies:)
      ⎕DISPLAY ((1 2)(3 4))2 3
.→---------------------------.
| .→-------------.           |
| | .→--.  .→--. |   2    3  |
| | |1 2|  |3 4| |           |
| | '~--'  '~--' |           |
| '∊-------------'           |
'∊---------------------------'

      X←2 2⍴⍳4                   (Two row, two column numeric matrix)
      Y←'HELLO'                  (Five element character vector)
      X Y                        (Variables entered in vector form)
 1 2   HELLO
 3 4
      ⍴X Y                       (Shape 2)
2
```

Primitive Functions

Built-in APL functions (or 'primitive' functions) are denoted by symbols (such as + - ÷ ×).

Primitive functions can be either *monadic* (which means they take a single right argument), or *dyadic* (in which case they take an argument on the left and an argument on the right). The same symbol may have both monadic and dyadic forms.

Execution order

A line of APL may consist of several functions, and arguments. All primitive and user-defined functions have the same precedence, and simply act on the data on the right. Thus, expressions are evaluated from right to left, and the result of one function becomes the (right) argument of the next function. See the section on Binding strengths for more details.

Scalar and Mixed functions

APL's primitive (i.e. built-in) functions fall into two classes, *Scalar* and *Mixed* functions. Scalar functions are defined on scalar arguments, and extend to arrays of any rank on an element-by-element basis. Mixed functions are defined on arrays, and may yield results which are different in shape or rank from their arguments. Most of the arithmetic primitives (such as addition, multiplication, logarithm) are scalar functions.

If one of the arguments to a dyadic function is a scalar, the scalar is applied to each element of the other argument (a property known as *scalar extension*). The other important property of scalar functions is that they are pervasive, that is they apply at all levels of nesting. Monadic scalar functions are applied independently to every simple scalar in their argument, and the result retains the structure of the argument. Dyadic scalar functions are applied independently to corresponding pairs of simple scalars in the other argument. If one of the arguments is a scalar, it will be applied to all simple elements of the other argument. For example:

```
      2 3 4 5 + 7 8 9 10
9 11 13 15
      23 + 7 8 9 10
30 31 32 33
      (1 2 3) (2 2⍴4 5 6 7) (7 8) + (10 11 12) (2 2⍴11 22 33 44) (60 70)
 11 13 15   15 27   67 78
            39 51
      (1 2 3) (2 2⍴4 5 6 7) (7 8) + 1
 2 3 4   5 6   8 9
         7 8
```

Note that you can use the Each operator (¨) to apply a non-scalar function to each element of an array.

Numbers or text

Some functions work on numbers only. The arithmetic functions are in this category. You will get a message saying you've made a `DOMAIN ERROR` if you try to use any of the arithmetic operators on text data.

Some functions work on either. The `⍴` function, for example, can be used (with one argument) to find how many characters are in a text item, or how many numbers are in a numeric item. Its two-argument form (which you've seen used to shape data into a specified number of rows and columns) also works on either numbers or characters.

The logical functions (logical `∧`, `∨` and the rest of that family) work on a subset of the number domain. They recognise two states only, true or false as represented by the numbers `1` and `0`. If any other numbers or characters are submitted to them, a `DOMAIN ERROR` results.

Arithmetic functions

Function	***Monadic form***	***Dyadic form***
+	Identity (Conjugate) *(Scalar function)*	Add *(Scalar function)*
-	Negate *(Scalar function)*	Subtract *(Scalar function)*
×	Sign of *(Scalar function)*	Multiply *(Scalar function)*
÷	Reciprocal *(Scalar function)*	Divide *(Scalar function)*
⌈	Ceiling *(Scalar function)*	Greater of *(Scalar function)*
⌊	Floor *(Scalar function)*	Lesser of *(Scalar function)*
|	Absolute value *(Scalar function)*	Residue (remainder) of division *(Scalar function)*

(Note: the `-` minus sign represents the negate and subtract functions, the `¯` sign is used to identify negative numbers.)

Examples of arithmetic functions

A vector of numbers is multiplied by a single number.

```
      2 6 3 19 × 0.5
1 3 1.5 9.5
```

A vector of numbers is divided by a single number:

```
      3 7 8 11 ÷ 3
1 2.333333333 2.666666667 3.666666667
```

A vector of numbers is divided by a single number. The results are rounded up to the next whole number and are then displayed:

```
      ⌈ 3 7 8 11 ÷3
1 3 3 4
```

The same operation as the last example, except that `0.5` is subtracted from each number before it's rounded up in order to give 'true' rounding:

```
      ⌈ ¯0.5 + 3 7 8 11 ÷3
1 2 3 4
```

Two vectors containing some negative values are added. `×` is applied to the resulting vector to establish the sign of each number. The final result is a vector in which each positive number is represented by a `1`, each negative number by a `¯1` and each zero by a `0`.

```
      ×12 ¯1 3¯5 + 2 ¯6 ¯4 5
1 ¯1 ¯1 0
```

The remainder of dividing 17 into 23 is displayed:

```
      17 | 23
6
```

The remainders of two division operations are compared and the smaller of the two is displayed as final result:

```
      (3 |7 ) ⌊ 4 | 11
1
```

Algebraic functions

Function	***Monadic form***	***Dyadic form***
`⍳`	Index generator	*(see Comparative functions)*
`?`	Roll (Random number) *(Scalar function)*	Random deal
`*`	'e' to power *(Scalar function)*	Power *(Scalar function)*
`⍟`	Natural Logarithm *(Scalar function)*	Log to the base *(Scalar function)*
`○`	pi times *(Scalar function)*	Circular & Hyperbolic functions (Sine, cosine, etc) *(Scalar function)*
`!`	Factorial or Gamma function *(Scalar function)*	Binomial *(Scalar function)*
`⌹`	Matrix inversion	Matrix division

<u>Examples of algebraic functions</u>

The numbers 1 to 10 are put in a variable called `X`.

```
      X ← ⍳10
1 2 3 4 5 6 7 8 9 10
```

3 random numbers between 1 and 10, with no repetitions.

```
      3?10
2 8 3
```

The logarithm to the base 2 of 2 4 8.

```
      2 ⍟ 2 4 8
1 2 3
```

The number of combinations of 2 items which can be made from a population of 4 items.

```
      2 ! 4
6
```

Comparative functions

Function	***Dyadic form only***
<	Less than *(Scalar function)*
≤	Less than or equal *(Scalar function)*
=	Equal *(Scalar function)*
≥	Greater than or equal *(Scalar function)*
>	Greater than *(Scalar function)*
≠	Not equal *(Scalar function)*
≡	Match
≢	Not Match
∊	Membership
⍳	Index of
⍷	Find

Examples of comparative functions

Are two given numbers equal? (1 = yes 0 = no)

```
      10 = 5
0
      12 = 12
1
```

Are the corresponding characters in two strings equal?

```
      'ABC' = 'CBA'
0 1 0
```

Is the first number greater than the second?

```
      10 > 5
1
```

Is each number in the first vector less than the corresponding number in the second vector?

```
      3 9 6 < 9 9 9
1 0 1
```

Is the number on the left in the vector on the right?

```
      12 ∊ 6 12 24
1
```

Is the character on the left in the string on the right?

```
      'B'  ∊  'ABCDE'
1
```

Which numbers in a matrix are negative? (The contents of `TABLE` are shown first so that you can see what's going on.)

```
      TABLE
12 54  1
¯3 90 23
16 ¯9  2
      TABLE < 0
0 0 0
1 0 0
0 1 0
```

Find the number on the right in the vector on the left and show its position.

```
      13 7 9 0⍳9
3
```

Are two matrices exact matches?

```
      (2 2⍴⍳4) ≡  (2 2⍴⍳4)
1
```

Find the pattern `'CAT'` within the characters `'THATCAT'`

```
      'CAT ' ⍷ 'THATCAT '
0 0 0 0 1 0 0
```

Logical functions

Function	***Monadic form***	***Dyadic form***
~	Not *(Scalar function)*	*See Selection functions*
∨		Or *(Scalar function)*
∧		And *(Scalar function)*
⍱		Nor *(Scalar function)*
⍲		Nand *(Scalar function)*

<u>Examples of logical functions</u>

Logical NOT:

```
      ~1 1 1 0 0 0 1
0 0 0 1 1 1 0
```

The same data submitted to various logical functions:

```
      1 ∨ 0
1
      1 ∧ 0
0
      1 ⍱ 0
0
      1 ⍲ 0
1
```

Each element in one vector is compared (∧) with the corresponding element in another.

```
      1 0 1 ∧ 0 0 1
0 0 1
```

Two expressions are evaluated. If both are true (i.e. both return a value of 1) then the whole statement is true (i.e. returns a value of 1):

```
      (5 > 4) ∧ 1 < 3
1
```

Manipulative and selection functions

Function	***Monadic form***	***Dyadic form***
⍴	Shape of	Reshape
≡	Depth of an array	*(see comparative functions)*
,	Ravel (Convert array to vector)	Catenate (join) data items
∊	Enlist (Make into simple vector)	*(see comparative functions)*
~	*See logical functions*	Without (Removes elements from a vector)
∪	Unique	Union
ö		Intersection
⌽	Reverse elements	Rotate elements
⍉	Transpose	Transpose as specified
↑	First	Take from an array
↓		Drop from an array
⊂	Enclose an array	Partition (Creates an array of vectors)
⊃	Disclose an array	Pick items from an array
⌷		Index an array
⊣	Stop (replace argument with empty)	Left (pass left argument)
⊢	Pass (argument unchanged)	Right (pass right argument)

Examples of manipulative functions

An enquiry about the size of a character string:

```
      ⍴ 'ARLINGTON A.J, 22 BOND RD SPE 32E'
```

```
33
```

A three-row four-column matrix is formed from the numbers 1 to 12 and is assigned to `DOZEN`:

```
      DOZEN ← 3 4 ρ ι 12
      DOZEN
1  2  3  4
5  6  7  8
9 10 11 12
```

The matrix `DOZEN` is ravelled into a vector:

```
      ,DOZEN
1 2 3 4 5 6 7 8 9 10 11 12
```

The matrix `DOZEN` is first converted to vector form and is then catenated (joined) with the vector `13 14 15`):

```
      (,DOZEN), 13 14 15
      DOZEN
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

The matrix `DOZEN` is re-formed from the original data in reverse order:

```
      +DOZEN ← 3 4ρ⌽,DOZEN
12 11 10 9
 8  7  6 5
 4  3  2 1
```

Numbers are removed from a vector:

```
      1 2 3 4 5 6 ~ 2 4 6
1 3 5
```

First 3 characters are selected from a vector:

```
      3 ↑'AWFULLY'
AWF
```

Data array enclosed into a nested scalar, with an empty shape:

```
      ⊂999 34
 999 34
      ρ⊂999 34
empty
```

Index the third item from a vector:

```
      3 ⌷ 1 2 3 4 5
3
```

Sorting and coding functions

Function	*Monadic form*	*Dyadic form*
⍋	Ascending sorted indices, default sort order	Ascending sorted indices, specified sort order
⍒	Descending sorted indices, default sort order	Descending sorted indices, specified sort order
⊤		Encode (Convert to a new number system)
⊥		Decode (Convert back to units)

Examples of sorting and coding functions

To put a vector of numbers into ascending order:

```
      LIST ← 200 54 13 9 55 100 14 82
      ⍋LIST
4 3 7 2 5 8 6 1
      LIST[4 3 7 2 5 8 6 1)
9 13 14 54 55 82 100 200
```

To sort the same vector as in example 1 with less typing:

```
      LIST[⍋LIST]
9 13 14 54 55 82 100 200
```

To find how certain symbols rank in the collating order (i.e. the order in which APL holds characters internally):

```
      SYMBS ← '⌹\≠(/'
      ORDER ← ⍋SYMBS
      SYMBS[ORDER]
⌹≠/(\
```

To convert the hex number 21 to its decimal equivalent:

```
      16 16 ⊥ 2 1
33
```

Formatting functions

Function	*Monadic form*	*Dyadic form*
⍕	Format	Format by specification, Format by example
⍺		Picture format

Examples of formatting functions

To display each number in a vector in a 6-character field with two decimal places:

```
      6 2 ⍕ 60.333333 19 2 52.78
60.33 19.00 2.00 52.78
```

To display each number in a vector preceded by a dollar sign and with up to three leading zeroes suppressed:

```
      '$$Z,ZZ9' ⍺ 3899 66 2
$3,899    $66     $2
```

Miscellaneous functions and other symbols

Function	
⎕	Accept numbers from keyboard or Output with newline
⍞	Accept characters from keyboard or Bare output
⋄	Statement separator
⍝	Comment
⍎	Execute an APL expression
⍬	Empty numeric vector (Zilde)

Primitive Operators

An 'operator' modifies the behaviour of a primitive or user-defined function. It has an operand or operands that are primitive, derived or user-defined functions or data. The result of using an operator is known as a derived function which can then be applied monadically or dyadically to data or alternatively it may be, in turn, used as an argument to another operator. Operators can themselves be monadic or dyadic. Monadic operators will be placed to the right of their operands:

```
             +/                       (Monadic / operator)
             ρ¨                       (Monadic ¨ operator)
             +.×                      (Dyadic . operator)
```

Operators form a powerful extension to the repertoire of the language. They can be used to specify the **way** in which a function or functions are to be applied to data - they allow a function to be applied repeatedly and cumulatively over all the elements of a vector, matrix or multidimensional array.

The primitive operators available are:

Operator	*Name*
/	Reduce or Compress
\	Scan or Expand
.	Inner Product
∘.	Outer Product
¨	Each
[ ]	Axis

Reduce and scan

When used with **functions** as their operand, slash and backslash are known as reduce and scan. Reduce and scan apply a single function to all the elements of an argument. For example, to add up a vector of arguments, you can either type:

```
      22 + 93 + 4.6 + 10 + 3.3
132.9
```

or alternatively:

```
      +/22 93 4.6 10 3.3
132.9
```

The / operator in the last example had the effect of inserting a plus sign between all the elements in the vector to its right.

The \ operator is similar except that it works cumulatively on the data, and gives all the intermediate results. So:

```
      +\22 93 4.6 10 3.3
 22 115 119.6 129.6 132.9
```

from the results of:

```
      22 (22+93) (115+4.6) (119.6+10) (129.6+3.3)
```

Compress and Expand

When used with one or more **numbers** as their operand, slash and backslash carry out operations known as compress and expand.

Compress can be used to select all or part of an object, according to the value of the numbers forming its operand. For example, to select some characters from a vector:

```
      1 0 1 1 0 1 / 'ABCDEF'
ACDF
```

Conversely, expand will insert fill data into objects:

```
      TAB ← 2 3⍴⍳6
      TAB
1 2 3
4 5 6
      1 0 1 0 1 \[2]TAB
1 0 2 0 3
4 0 5 0 6
```

Columns are inserted in positions indicated by the `0`s. (Note also the use of the axis operator).

Outer and inner products

The product operators allow APL functions to be applied between all the elements in one argument and all the elements in another.

This is an important extension because previously functions have only applied to **corresponding** elements as in this example:

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

The outer product gives the result of applying the function to **all** combinations of the elements in the two arguments. For example, to find the outer product of the two arguments used in the last example:

```
      1 2 3 ∘.+ 4 5 6
5 6 7
6 7 8
7 8 9
```

The first row is the result of adding the first element on the left to every element on the right, the second row is the result of adding the second element in the left to every element on the right and so on till all combinations are exhausted.

This example works out a matrix of powers:

```
      1 2 3 4 ∘.*1 2 3 4
1  1  1   1
2  4  8  16
3  9 27  81
4 16 64 256
```

as can be seen more clearly if we lay it out like this:

```
 |_1___2___3____4
 |---------------
1|  1   2   3    4
2|  2   4   6    8
3|  3   9  27   81
4|  4  16  64  256
```

(Since the outer product involves operations between all elements, rather than just between corresponding elements, it's not necessary for the arguments to conform in shape or size.)

The inner product allows **two** functions to be applied to the arguments. The operations take place between the **last** dimension of the left argument and the **first** dimension of the right argument, hence 'inner' product since the two inner dimensions are used.

In the case of matrices, first each row of the left argument is applied to each column of the right argument using the rightmost function of the inner product, then the leftmost function is applied to the result, in a reduction (`/`) operation.

Given that you can use a combination of any two suitable functions, there are many possible inner products. These can perform a variety of useful operations. Some of the more common uses are:

- locating incidences of given character strings within textual data
- evaluation of polynomials
- matrix multiplication
- product of powers

Each

As its name implies, the each operator will apply a function to each element of an array.

So, to find the lengths of an array of vectors

```
      ⍴¨(1 2 3) (1 2) (1 2 3 4 5)
3 2 5
```

As with other operators, each can be used for user-defined functions. Here we use an 'average' function on an array of vectors.

```
      AVERAGE 1 2 3
2
      AVERAGE ¨ (1 2 3) (4 5 6) (10 100 1000)
2 5 370
```

Axis Operator

A number of primitive functions and operators can be applied to a particular axis (or dimension) of an array. The [ ] brackets are used to indicate the axis being specified.

The highest dimension of a data item is considered to be the first dimension and the lowest dimension the last . Thus the first dimension of a matrix is the rows and the last dimension is the columns. In the case of a three-dimensional object, the first dimension is the planes followed by the rows and columns.

Axis numbers are governed by the Index Origin, ⎕IO, and in Index Origin 1, (the default), the first dimension is represented by [1], the second by [2] and so on. In Index Origin 0 the first dimension would be [0], the second [1] and so on. The number used to represent the axis is always a whole number, except for the ravel and laminate functions.

The primitive functions and operators which will accept an axis operator include the dyadic forms of the primitive scalar functions :

```
     + - × ÷ | ⌈ ⌊ * ⍟ ○ ! ∧ ∨ ⍲ ⍱ < ≤ = ≥ > ≠
```

and some primitive mixed functions :

```
     , ⍪       Ravel/Catenate/Laminate       (note first axis variant)
     ⌽ ⊖       Reverse/Rotate                (note first axis variant)
     ⊂         Enclose/Partition
     ⊃         Disclose
     ↑         Take
     ↓         Drop
     ⌷         Index
```

as well as the operators:

```
     / ⌿       Compress/Replicate            (note first axis variant)
     / ⌿       Reduce                        (note first axis variant)
     \ ⍀       Scan                          (note first axis variant)
     \ ⍀       Expand                        (note first axis variant)
```

See the reference section entry for Axis ([]) for more details, as well as the reference entries for individual mixed functions and operators listed above.

Formatting

The default way in which APL displays results may not always suit your requirements. Obviously you can do a certain amount by using functions like size to reshape data, or catenate to join data items, but for many applications you may want much more sophisticated facilities. You may, for example, want to insert currency signs and spaces in numeric output, or produce a neatly formatted financial report, or specify precisely the format in which numbers are displayed.

APLX has a variety of functions for formatting data, providing flexibility as well as compatibility with a number of other APL interpreters.

Formatting functions

There are three functions in APLX which all convert the format of data from numbers to characters, and allow you to specify how the converted numeric data should be laid out.

The functions are:

- The ⍕ primitive (Format, Format by specification, Format by example)
- The ⍺ primitive (Picture format)
- The ⎕FMT system function.

Each function lets you specify how many character positions a number should occupy when it is displayed, and how many of these positions are available for decimal places. The number of characters and number of decimal places are specified in the left argument:

```
      6 2 ⍕ 1341.82921
341.83
```

(Note that since the number had to be truncated to fit the character positions allowed, it was first rounded to make the truncated representation as accurate as possible.)

Picture format (⍺) and Format by Specification (⍕ with a character left argument) allow you to use editing characters to define a 'picture' of how data should look when it is displayed. The picture is the left argument and the data the right.

The following example shows the values in a 4-row 2-column matrix called TAB. It then shows the ⍺ function applied to this matrix and its effect on TAB:

```
      TAB
1096.2   ¯416.556
 296.974 1085.238
¯811.188  844.074
¯745.416  153.468
```

```
      '$$Z,ZZ9.99 DR        ' ⍺ TAB
 $1,096.20               $416.56 DR
   $296.97             $1,085.24
   $811.19 DR            $844.07
   $745.42 DR            $153.47
```

⎕FMT takes the process a stage further, allowing a variety of picture phrases, qualifiers and decorators to be supplied as the format specification.

```
      'B K2 G< ZZ9 DOLLARS AND 99 CENTS>' ⎕FMT 8.23 12.86 0 2.52
   8 DOLLARS AND 23 CENTS
  12 DOLLARS AND 86 CENTS

   2 DOLLARS AND 52 CENTS
```

Names

The following rules apply to user-assigned *symbols*, i.e. the names of variables, functions, operators, classes and labels in APLX.

The first character of the name must be one of the alphabetic characters `A-Z` or `a-z`, or one of the characters `∆` or `⍙`. Subsequent characters can also include digits `0-9`, underbar `_` and high minus `¯`.

Names consist of up to 30 characters (longer names will be truncated).

The following are all valid names in APLX:

```
DATA X X1 FIRST_VALUE ∆ ∆L1 ErrorCode model mode¯restart a999 Item∆1
```

Case is significant in names, so `DATA` `Data` and `data` are three distinct names.

There are no reserved names in APL. System-assigned names are distinct from user-assigned names because they start with a Quad `⎕` symbol.

Specification (Assignment)

The symbol ← associates the data on its right with the name specified on its left. The named data is known as a *variable*. The name associated with it is the variable name. Subsequent references to the variable name automatically refer to the data associated with that name. This operation is known as *specification* or *assignment*.

```
            NUM← 24                     (scalar 24 is assigned to NUM)
            DESCRN←'ITEM 241A'          (simple character vector)
            PRICES← 2.34 9.30 12 60.38  (numeric vector is assigned to PRICES)
            DATA←1 'A' 2 'D'            (mixed vector assigned to DATA)
```

When entering character data, care must be exercised if the quote character is to be included in the data. As stated above, adjacent quote marks are evaluated as indicating the quote character in the data. A vector containing only characters can be entered in one of two ways - the characters can either be entered within one set of quote marks or the characters must be separated by spaces. See also the section on Vector Notation.

```
            ALF←'A' 'B' 'C' 'D'    (data is a set of characters)
            ALF
      ABCD
            ALF←'ABCD'             (alternative method of entry)
            ALF
      ABCD
            ALF←'A''B''C''D'       (no space between characters)
            ALF
      A'B'C'D
```

The right argument to ← can be any APL expression that generates a result:

```
            COST←110-67                 (The result of evaluating an
            COST                         expression is assigned to
      43                                 COST)
            PROFIT←(PRICE←COST×1.2)-COST←100
            PROFIT                      (right to left execution ensures
      20                                 that the value assigned to
                                         COST is used in the expression
                                         inside parentheses.)
```

Variables which are either scalars or vectors can be entered directly, as shown above. Matrices or higher dimensional arrays must be established or entered via functions (see for example ⍴, 'reshape').

```
            TAB←2 3⍴⍳6                  (The numbers 1 - 6 are arranged as 2
                                         rows of 3 columns and are assigned
                                         to TAB)
```

Multiple specification

It is possible to make multiple simultaneous assignments by enclosing a list of variable names in parentheses on the left of an assignment arrow.

```
      (A B C)← 1 2 3
      A
1
      B
2
      C
3
```

A scalar to the right of the assignment arrow will be assigned to every item on the left. This is known as 'scalar extension'.

```
      (A B C)←5
      A
5
      B
5
      C
5
      (A B C)← 'HI' 'THERE' 'FOLKS'
      ρA                       (A assigned 'HI' and so on)
2
      (A B C)←⊂'HI' 'THERE' 'FOLKS'
      ρA                       (A, B and C assigned the enclosed vector
3                               to the right of the assignment arrow)
```

Caution:

Do not omit the parentheses if you are trying to do multiple specification, as in:

```
      A B C←5
```

This expression will assign the value 5 to C and then attempt to evaluate the resultant expression. See also the discussion of binding strengths.

Selective specification

A number of APL functions can be used to select elements or portions of an array. These selection operations can also be used as specifications when enclosed in parentheses and used as the left argument to the assignment symbol. The array being selected must appear as the rightmost name within the parentheses. The following functions can be used to make the selection, either singly or in combinations.

Monadic ∊ ↑ , ⌽ ⍉ ⊖
Dyadic ↑ ↓ ⊃ ⍴ ⌽ ⍉ ⊖ ⌷

and the functions \ ('expand') and / ('compress', 'replicate').

Bracket indexing can also be used as the left argument to the assignment arrow although in this case it is not necessary to enclose the indexing expression in parentheses.

Some examples will illustrate.

First, bracket indexing:

```
      TAB←2 3⍴⍳6              (Simple matrix)
      TAB[2;1]←8              (Row 2 column 1 assigned the value 8)
```

Nearly all the selection functions listed above operate on the outermost structure of a nested array. The shape of the right argument to the assignment arrow must either match that of the selected elements or be a scalar in which case scalar extension applies.

```
      VEC←⍳5
      (3↑VEC)←'ABC'           (First three elements become 'ABC')
      VEC
ABC 4 5
      (3↑VEC)←'A'             (A scalar right argument is extended to
      VEC                      all items specified)
AAA 4 5
      MAT←3 4⍴'ABCDEFGHIJKL'   (Simple character matrix)
      (,MAT)←'NEW DATAHERE'   (Ravel used for selection, so vector
      MAT                      used as right argument to ←)
NEW
DATA
HERE
      (('A'=,MAT)/,MAT)←'*'   (Combination of compression and ravel
      MAT                      used for selection)
NEW
D*T*
HERE
      (,2 2↑MAT)←'□□□□'       (Combination of ↑ and , used for
      MAT                      selection)
□□W
□□T*
HERE
```

```
      TABLE←3 4⍴⍳12
      TABLE
1  2  3  4
5  6  7  8
9 10 11 12
      (1 0 1 0/TABLE)←3 2⍴100 (Compression used for selection)
      TABLE
100  2 100  4
100  6 100  8
100 10 100 12
      DATA←⍳13
      X←10 20 30               (Other APL functions may be used within
      ((⍴X)↑DATA)←X             the parentheses - here ⍴ is used to supply
      DATA                      the left argument to ↑)
10 20 30 4 5 6 7 8 9 10 11 12 13
      Y←⍳10
      X←3
      ((2+X)↑Y)←⌽⍳X+2          (Left argument to ↑ includes the + function)
      Y
5 4 3 2 1 6 7 8 9 10
```

The function enlist (∊) removes all nesting from an array. When used with selective specification, it can be used to replace elements at the deepest level of nesting, whilst retaining the array's structure .

```
      NEST←(2 2⍴⍳4) 'TEXT' (3 1⍴⍳3)
      NEST
 1 2   TEXT   1
 3 4          2
              3
      (∊NEST)←0                (Entire array set to 0, but original
      NEST                      structure retained)
 0 0   0 0 0 0   0
 0 0             0
                 0
   (6⌷∊NEST)←999               (Single element at bottom of nested array
   NEST                         array altered)
0 0   0 999 0 0   0
0 0               0
                  0
   (7⌷∊NEST)←⊂'TEXT'           (Further nesting introduced)
   NEST
0 0    0 999 TEXT 0    0
0 0                    0
                       0
```

The function first (↑), selects the whole array which is the first element in an array. If first is used purely to select the first array within a nested array, then the array which is the right argument to the assignment arrow will replace the selected array.

```
      (↑NEST)←'ABC'            (First element of NEST is a matrix of
      NEST                      shape 2 by 2. This is replaced by
 ABC   0 999 TEXT 0    0        a length 3 vector)
                       0
                       0
      (2⌷↑NEST)←'⎕'            (One element within the first element of
      NEST                      NEST is replaced)
 A⎕C   0 999 TEXT 0    0
                       0
                       0
```

Pick `(⊃)` will select an entire array at an arbitrary depth in a nested array, and will also replace that entire array by the right argument to the specification symbol.

```
      2 2⊃NEST
999
      (2 2⊃NEST)←⍳10          (New array placed in 2 2⊃ of NEST)
      NEST
 A⎕C  0  1 2 3 4 5 6 7 8 9 10  TEXT 0    0
                                         0
                                         0
      (2⊃NEST)←'DATA'         (Nested vector at element 2 replaced by
      NEST                     length 4 vector)
 A⎕C DATA   0
            0
            0
      (3 (2 1)⊃NEST)←1000     (Row 2 column 1 of element 3
      NEST                     specified)
 A⎕C DATA      0
            1000
               0
```

There are some exceptions and restrictions to the rules for selective specification:

- User-defined functions and operators cannot be used within selective specification

- Execute `(⍎)` is not allowed within selective specification

- System functions are allowed within selective specification with the exception of those which use execute (`⎕EA` and `⎕EC`)

- The selection expression must select elements from the variable and not insert fill items (as, for example, can be done by expand and replicate).

- No arithmetic operations can be carried out on the array being specified or on the elements selected

- Assignments are not allowed within the parentheses used for selective specification.

- Selective and multiple specification operations cannot be mixed.

Thus, if

```
      X←3 4 5
```

then the following expressions are not allowed:

```
      (AVERAGE X)←6           (Use of user-defined function)

      ((⍎'1+2')↑X)←'ABC'      (Use of execute)

      ((Y←2)↓X)←'A'           (Assignment within the selection expression)

      ((2↓X) Y Z)←'ABC'       (Mixture of multiple and selective
                               specifications)
```

```
            (10↑X)←⍳10                (Fill items inserted by the selection
                                       expression)
```

As stated above, arithmetic may not be carried out on the elements of an array that are selected:

```
            X←⍳5
            (2+1↑X)←5
      DOMAIN ERROR
      (2+1↑X)←5
           ^
```

but other expressions within the specification parentheses may use arithmetic operations, even on another instance of the name being specified:

```
            ((2+1↑X)↑X)←100
            X
      100 100 100 4 5
```

Binding strengths

In simple terms, APL evaluates expressions right-to-left, that is to say the result of the rightmost function is evaluated, and becomes the right argument of the next function. There are no 'precedence rules' to remember; all primitive and user-defined functions have the same precedence. For example:

```
      5⍴3.2×12÷4
9.6 9.6 9.6 9.6 9.6
```

In this example, the division `12÷4` is evaluated first. The result of this expression becomes the right argument of the multiply, which returns the scalar result `9.6`. This in turn becomes the right argument of the reshape (`⍴`) function.

The right to left function execution rule needs to be modified to cope with more complex expressions, for example nested vectors or certain expressions containing operators. The 'binding strength' defines how certain symbols 'bind' for evaluation. The order of binding strengths is shown below, in descending order. (Note that binding strengths can be altered by the `⎕CS` and `)CS` ('compatibility setting') commands.)

Binding	*Bound items*
Brackets `[]`	Brackets to object to the left
Specification ← left	← to object on its left
Right Operand	Dyadic operator to its right operand
Vector	Array to array
Left Operand	Operator to its left operand
Left Argument	Function to left argument
Right Argument	Function to right argument
Specification ← right	← to object on its right

Parentheses can override the binding strength hierarchy. Some examples include:

```
      A←'DEF'                (Set up variables A B)
      B←'XYZ'
      A B
DEF XYZ
      A B[2]                 ([] has higher binding than vector so the
DEF Y                         result includes the second element of B)
      (A B)[2]               (Parentheses force selection of B)
XYZ
      A B←3                  (← has stronger binding than vector)
DEF 3
      (A B)←3                (Parentheses alter binding)
      A
3
      B
3
```

```
      1 2 3 + 4 5 6          (Vector has stronger binding than function)
5 7 9
      1 2 (3+4) 5 6          (Parentheses alter binding)
1 2 7 5 6
      1 0 1/'ABC'            (Vector has stronger binding than left
AC                            operand, so left operand is 1 0 1)
      +/[2]2 2⍴⍳4            (Axis brackets have stronger binding than
3 7                           operator to operands, so /[2] operator
                              is formed
```

Finally, the relative binding strengths of left and right operands can be used to predict the result of expressions with multiple operators. `+.×.-` is evaluated as `(+.×).-` and not as `+.(×.-)` since the `×` binds first as right operand to the first . ('inner product') operator.

Bracket indexing

Bracket indexing can be used to select elements from an array, for example one or more elements from a vector, or individual rows or columns of a matrix.

The index or indices are enclosed in square brackets, each dimension being separated by a semicolon. If no number is used for a particular dimension, then all the elements in that dimension are selected. APL allows index references to start either at 0 or 1. The index origin (which is controlled by `⎕IO` or `)ORIGIN`) determines whether index positions start from 1 or 0. In the examples below, and generally throughout this manual, the default convention of index origin 1 is used.

```
      LIST←12 24 36 48
      LIST[2]                (Selects the second item in LIST)
24
      LIST[1]+LIST[4]        (Adds the first and fourth items in LIST)
60

      ALF←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      ALF[26 1 13 2 9 1]     (Selects the letters in ZAMBIA
ZAMBIA                        from the contents of ALF)

      TABLE                  (TABLE consists of 2 rows and 4 columns)
10 20 30 40
50 60 70 80
     TABLE[1;4]              (Selects the item in row 1, column 4)
40

      TABLE[1;1 2 3 4]+TABLE[2;1 2 3 4]
60 80 100 120                (Adds the 4 columns in row 1 to the
                              4 columns in row 2)

      TABLE[1;]+TABLE[2;]     (Shorthand way of doing the same
60 80 100 120                 operation as in the last example)
```

In general, the indices may be of any shape or rank, so long as each of their elements correspond to valid elements within the array being indexed. The shape of the result of an indexing operation is generated by the shape of the index arrays. Thus

```
      ⍴TABLE[A;B]            (Where A and B are arrays)
```

is identical to

```
      (⍴A),⍴B
```

This has the important consequence, that if all the indexing arrays are in fact scalars, the result is also a scalar. Similarly, any axis of an array indexed by a scalar generates a result in which that axis does not exist.

```
      ALF[2 2⍴⍳4]            (ALF indexed by a matrix)
AB                           (Result is a matrix)
CD
```

```
      ⍴TABLE[1;1 2 3 4]       (Rows indexed by a scalar, result is
4                              a vector)

      ⍴TABLE[,1;1 2 3 4]      (Rows indexed by a vector, result is
1 4                            a matrix)

      ⍴TABLE[1 1⍴1;1 2 3 4]   (Rows indexed by a matrix,
1 1 4                          result is a three dimensional array
```

The ⌷ ('index') function

An alternative to bracket indexing is the ⌷ ('index') function, which is discussed fully in the reference section. The index specification is given as the left argument to the ⌷ function and is equivalent to bracket indexing in that

```
      ROW COL ⌷ MATRIX
```

and

```
      MATRIX[ROW;COL]
```

are equivalent. Although arguably less readable than bracket indexing, the index function has the advantage that it is syntactically consistent with other APL primitive functions, and can thus be used with operators such as Each.

User-defined Functions

User-defined functions are the equivalent of subroutines or functions in other programming languages. They associate a series of lines of APL code with a name chosen by the programmer.

When a function is evaluated, it performs some action on data known as an 'argument'. Functions may have no arguments, one argument, or two arguments. These three types of functions are often referred to as follows:

```
0 arguments              Niladic
1 argument               Monadic       Argument on right
2 arguments              Dyadic        Arguments on left and right
```

If you defined a function called. say, `SD` which found the standard deviation of a set of numbers, you could write it so that it expected the data as its right-hand argument. You would then call `SD` in exactly the same way as a primitive function such as `⌈`:

```
      X ← SD 23 89 56 12 99 2 16 92
```

A function may or may not return a result.

You specify the number of arguments the function is to have, and the name of the result field (if there is one) when you define the function header of the function you are about to write.

Header line for user-defined functions

In addition to the names used for the left and right arguments and result (if applicable) which will all be 'local', the header line may also be used to localize other variables (and system variables), as well as function names. Whilst the function or operator is running, these local variables 'shadow' global variables of the same name, that is they will exclude from use a global object of the same name. System commands continue to reference the global objects. Local variables (and functions) are however themselves global to functions called within their function or operator.

The general format for a function header is:

```
             R← A FUNCTION B;VAR1;VAR2
or
                A FUNCTION B;VAR1;VAR2
```

depending on whether or not a result is returned. R, the result, A, the left argument, B, the right argument are all optional. Local names, if any, are listed after the function name and arguments, separated from them and each other by semi-colons (;), VAR1 and VAR2 above. Comments may also appear at the end of the header line, following a `⍝` ('comment') symbol.

Editing functions

In most versions of APLX, there are two ways to create or edit a function.

The most commonly used way is to use an **on-screen editor**, which allows you to edit the function text very easily in an editor window. The editor is either invoked through the application's Edit menu, or with the `)EDIT` system command (or the `⎕EDIT` system function), e.g.

```
      )EDIT FUNK
```

For backward compatibility with old APL systems, APLX also supports a primitive line-at-a-time editor called the Del (or Line) Editor. To enter definition mode and create a new function you type `∇` (Del) followed by the function name. If you type nothing else, you are defining a function that will take no arguments:

```
      ∇FUNK
```

For clarity, we will list functions here as though they were entered using the Del editor, where a `∇` character is used to mark the start and end of the function listing. If you are using the on-screen editor, you **do not type** the `∇` characters or the line numbers.

The function header

The first line of a function is called the function header. This example is the header for a function called `FUNK`:

```
      ∇FUNK
```

If you want the function you are defining to have arguments you must put them in the header by typing a suitable function header:

```
      ∇SD X
```

The above header specifies that `SD` will take one argument. Here is what `SD` might look like when you had defined it:

```
      ∇SD X
  [1] SUM ← +/X
  [2] AV ← SUM÷⍴X
  [3] DIFF ← AV-X
  [4] SQDIFF ← DIFF*2
  [5] SQAV ← (+/SQDIFF)÷⍴SQDIFF
  [6] RESULT ← SQAV*0.5
      ∇
```

It's quite unimportant what the statements in the function are doing. The point to notice is that they use the variable `X` named in the function header. When `SD` is run, the numbers typed as its right-hand argument will be put into `X` and will be the data to the statements that use `X` in the function. So if you type:

```
      SD 12 45 20 68 92 108
```

those numbers are put in `X`. Even if you type the name of a variable instead of the numbers themselves, the numbers in the variable will be put into `X`.

The function header for a dyadic (two-argument) function would be defined on the same lines:

```
      ∇X CALC Y
```

When you subsequently use `CALC` you can supply two arguments:

```
      1 4 7 CALC 0 92 3
```

When `CALC` is run the left argument will be put into `X` and the right argument into `Y`.

If you want the result of a function to be put into a specified variable, you can arrange that in the function header too:

```
      ∇Z ← X CALC Y
```

In practice, most APL functions return a result, which can then be used in expressions for further calculations, or stored in variables.

Defining `Z` to be the result of `X CALC Y` allows the outcome of `CALC` to be either assigned to a variable, or passed as a right argument to another (possibly user-defined) function, or simply displayed, by not making any assignment. The variable `Z` acts as a kind of surrogate for the final result during execution of `CALC`.

Local and global variables

Variable names quoted in the header of a function are **local**. They exist only while the function is running and it doesn't matter if they duplicate the names of other variables in the workspace.

The other variables - those used in the body of a function but not quoted in the header, or those created in calculator mode - are called **global** variables.

In the `SD` example above, `X` was named in the header so `X` is a local variable. If another `X` already exists in the workspace, there will be no problem. When `SD` is called, the `X` local to `SD` will be set up and will be the one used. The other `X` will take second place till the function has been executed - and of course, its value won't be affected by anything done to the local `X`. The process whereby a local name overrides a global name is known as 'shadowing'.

It is obviously convenient to use local variables in a function. It means that if you decide to make use of a function written some time before, you do not have to worry about the variable names it uses duplicating names already in the workspace.

But to go back to the `SD` example. Only `X` is quoted in the header, so only `X` is local. It uses a number of other variables, including one called `SUM`. If you already had a variable called `SUM` in the workspace, running `SD` would change its value.

You can 'localize' any variable used in a function by putting a semicolon at the end of the function header and typing the variable name after it:

```
      ∇SD X;SUM
```

You may wonder what happens if functions that call each other use duplicate local variable names. You can think of the functions as forming a stack with the one currently running at the top, the one that called it next down, and so on. A reference to a local variable name applies to the variable used by the function currently at the top of the stack.

Comments in functions

If you want to include comments in a function, simply enter them, preceded by a comment `⍝` symbol.

```
      ∇R ← AV X
  [1] ⍝ This function finds the average of some numbers
  [2] R ← (+/X)÷⍴X ⍝ The numbers are in X
      ∇
```

There are two comments in the example above. Note that the one on line 2 doesn't start at the beginning of a line.

Locked functions

It is possible to *lock* a function. A locked function can only be run. You can not edit it or list the statements it consists of. To lock a function, edit it in the Del editor but type a `⍫` rather than a `∇` to enter or leave function definition mode.

A locked function cannot be unlocked.

Localized functions

Local functions cannot be edited by the standard `∇` editor, and the `∇` editor will always refer to a global function of the same name (if it exists). `⎕CR` may be used to examine local functions.

Ambivalent functions

All dyadic functions may be used monadically. If used monadically, the left argument is undefined (i.e. has a Name Classification, `⎕NC` of 0). This type of function is known as an *ambivalent* or *nomadic* function, and will usually start by testing for the existence of the left argument.

```
         ∇R←A NOMADIC B
    [1] :If 0=⎕NC 'A'         ⍝ DOES A EXIST?
    [2]   A←5                 ⍝ NO, SO WE HAVE BEEN USED MONADICALLY
    [3] :EndIf
        etc.
```

User-defined Operators

An 'operator' modifies the behaviour of a primitive or user-defined function. It has an operand or operands that are primitive, derived or user-defined functions or data. The result of using an operator is known as a *derived function* which can then be applied monadically or dyadically to data or alternatively it may be, in turn, used as an argument to another operator.

As well as the primitive (built-in) operators, user-defined operators are permitted. These are created an edited in the same way as user-defined functions (using the `∇` editor, or `)EDIT`), but are distinguished from functions by the format of the header, line 0.

Header line for user-defined operators

The format for an operator header follows one of the following forms, where `LO`=Left operand, `RO`=Right operand, `A`=Left argument, `B`=Right argument:

```
             R←(LO OPERATOR) B          (Monadic operator with one argument)
or
             R←A (LO OPERATOR) B        (Monadic operator with two arguments)
or
             R←(LO OPERATOR RO) B       (Dyadic operator with one argument)
or
             R←A (LO OPERATOR RO) B     (Dyadic operator with two arguments)
```

User-defined operators need not return explicit results.

Example

This simple monadic operator with two arguments `COMMUTE` reverses the arguments of a function. In this example, `FN` represents the function (the left operand) which will be combined with the operator to make a *derived function*, `L` represents the left argument supplied to the derived function, and `R` represents the right argument supplied to the derived function:

```
      ∇Z←L (FN COMMUTE) R
[1]  ⍝ Operator which reverses the arguments to a dyadic function
[2]   Z←R FN L
      ∇

      100 ÷ 3
33.33333333
      100 ÷ COMMUTE 3       ⍝ Equivalent to 3 ÷ 100
0.03
      100 ⍴ COMMUTE 3       ⍝ Equivalent to 3 ⍴ 100
100 100 100
      100 ⎕DR COMMUTE 1     ⍝ Equivalent to 1 ⎕DR 100
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0
```

Using data as operands

The left and/or right operands to a user-defined operator do not have to be functions; they can alternatively be arrays. The effect is to substitute the supplied array for in expression which references the operand:

```
      'FRUIT' ('OLD' COMMUTE) 'HELLO'
HELLO OLD FRUIT
```

Classes and Objects

Overview of Classes and Objects

As well as traditional APL functions and operators, APLX adds object-oriented programming facilities to the core APL language. These facilities are broadly similar to those implemented in other object-oriented programming languages (such as C++, C#, Java, or Ruby), but with the difference that APL's array-programming approach applies to classes and objects in the same way as it applies to ordinary data.

The fundamental building block for object-oriented programming in APLX Version 4 is the *class*. For example, in a commercial invoicing application, a given class might represent the attributes and behavior of an Invoice, and another class might represent a CreditNote. In an application concerned with geometry, a class might represent a Sphere, or a Rectangle, or a Polygon. A class contains definitions both for program logic (functions and operators, known collectively as the *methods* of the class), and for data (named variables associated with the class, known as *properties*). The term *members* is used to describe both the properties and methods of a class.

In most cases, when you come to use a class, you need to create an *instance* of that class, also known as an *object*. Whereas the class represents an abstraction of (say) an Invoice, or a Sphere, or a Rectangle, an object represents a particular invoice, sphere or rectangle. Typically, you may have many instances of a given class, each containing independent copies of data (properties), but all supporting the same program logic (methods).

Inheritance

When you define a class, you can specify that it *inherits* from another class. The new class is said to be the *child*, and the class it inherits from is the *parent* or *base* class. Inheritance means that (unless you explicitly change their definition), all of the properties and methods defined in the parent class are also available in the child class. This works for further levels of inheritance as well, so that methods and properties can be inherited from the immediate parent, or from the parent's parent, and so on. The terms *derived classes* or *descendants* are sometimes used to denote the children of a class, and the children's children, and so on. Similarly, the term *ancestors* of a class is used to denote the parent, parent's parent, and so on.

For example, you might have a class Shape, representing an abstract geometric shape. This might have properties called 'X' and 'Y' giving the center point of the shape, and methods called 'Move' and 'Area'.

A Circle class might inherit from Shape, introducing further properties such as 'radius'. Equally, a class Polygon might also inherit from Shape, and further classes Triangle and Square inherit from Polygon. All of the classes Circle, Polygon, Triangle and Square are derived from Shape. Because of the way inheritance works, they would all include the properties X and Y, and the methods Move and Area.

When a class inherits from another, you can specify that the definition of a given method of the parent (or the initial value of a property) is different in the child class. In our example, you would need to supply a different definition of the Area method for a Circle and a Square. This is known as *overriding* the method.

For classes defined in APLX, all methods can be overridden, and all methods are *virtual*, that is to say if method A in a base class calls another method B, and the second method B is overridden in a child class, then running method A in the child class will cause the overridden version of B to be called, not the version of B defined in the parent.

APLX uses an inheritance model known as *single inheritance*. This means that a child class can be derived from only one parent (which may itself derive from another class, and so on). However, APLX also allows you to 'mix-in' one or more other classes (including external classes, such as those written in .Net or Java) into your objects at runtime. This is a very flexible feature which can be used in much the same way as multiple inheritance is used in some other languages. See the section on Mixins for more details.

User-defined, System and External classes

APLX supports the following types of class:

- User-defined classes, written in APL (also known as 'Internal' or just 'APL' classes)
- System classes, which are built-in to the APLX interpreter in the same way as System functions. System classes are currently used mainly for user-interface programming, and replace the older `⎕WI` syntax.
- External classes, written in other languages, such as Java or C#.

Object References and Class References

When you create an object, i.e. an instance of a class (using the system function `⎕NEW` as described below), the explicit result that is returned is not the object itself, but a *reference* to the object. This reference is held internally as just a number, an index into a table of objects which APLX maintains in the workspace. If you assign the reference to another variable, the object itself is not copied; instead, you have two references to the same object.

Of course, because APLX is an array language, you can have arrays of object references, and you can embed object references in nested arrays along with other data. For example, you might have an array containing references to hundreds of Rectangle objects.

You can also have a reference to a Class. This makes it possible for general functions to act on classes without knowing in advance which class applies.

Creating objects (instances of classes)

The system function `⎕NEW` is the principal means by which you create an object, i.e. an instance of a class. The class can be either written in APL (an internal or user-defined class), or a built-in System class, or a class written in an external environment such as .Net, Java or Ruby (an external class). `⎕NEW` creates a new instance of the class, runs any constructor defined for the class, and returns a reference to the new object as its explicit result.

The class is specified as the right argument (or first element of the right argument). It can be specified either as a class reference, or as a class name (i.e. a character vector). Any parameters to be passed to the constructor of the class (the method which is run automatically when a class is created) follow the class name or reference.

If you specify the class by name, you also need to identify in the left argument the environment where the class exists, unless it is internal.

Creating instances of internal (user-defined) classes

Normally, you create an instance of a user-defined class by passing the class reference directly as the right argument (or first element of the right argument). For example, if you have a class called Invoice, you can create an instance of it by entering:

```
     I←⎕NEW Invoice
```

What is really happening here is that the symbol Invoice refers to the class definition, and when it is used in this way, it returns a reference to the class.

Note that you can also pass the class name rather than a class reference. The following are alternative ways of creating an instance of a user-defined class:

```
      I←⎕NEW 'Invoice'
      I←'apl' ⎕NEW 'Invoice'
```

Passing arguments to the constructor

A constructor is a special method of a class, which is run automatically when the class is created using ⎕NEW, and is used to initialize the class. For APL classes, the constructor is a method whose name is the same as the name of the class. It should be a function which takes a right argument, and does not return a result. (It can be a method which takes no argument, if you are sure that no parameters will ever be passed to it via ⎕NEW). Any arguments to the constructor can be provided as extra elements on the right argument of ⎕NEW. When the constructor is run, these extra elements are passed as the right argument to the constructor. If there are no extra elements, an empty vector is passed as the right argument to the constructor.

For example, suppose the class Invoice looks like this:

```
Invoice {
  TimeStamp
  Account
  InvNumber
  {Serial←0}

 ∇Invoice B
  ⍝ Constructor for class Invoice.  B is the account number
  Account←B
  TimeStamp←⎕TS
  Serial←Serial+1
  InvNumber←Serial
 ∇
}
```

This is a class which has a constructor and four properties. One of the properties (Serial) is a class-wide property, which means it has only a single value shared between all instances of the class. When a new instance of this class is created, the constructor will be run. It will store the account number (passed as an argument to `⎕NEW`) in the property Account, and store the current time stamp in the property TimeStamp. It will then increment the class-wide property Serial (common to all instances of this class), and store the result in the property InvNumber. (To see the properties, we use the system method `⎕DS` which summarizes the property values):

```
      S←⎕NEW Invoice 23533
      S.⎕DS
Account=23533, TimeStamp=2007 10 11 15 47 34 848, InvNumber=1
      T←⎕NEW Invoice 67544
      T.⎕DS
Account=67544, TimeStamp=2007 10 11 15 48 11 773, InvNumber=2
```

Default display of a class or object reference

When you call the `⎕NEW` system function to create an object (an instance of a class), the explicit result is a reference to that object. The question therefore arises: what happens if you display such an object reference?

By default, APLX displays an object reference as the unqualified class name contained in square brackets. Class references are displayed as the class name in curly braces:

```
      )CLASSES
Queue                      ⍝ User-defined APL class
      Queue
{Queue}                    ⍝ Default display of class reference
      QUEUE23←⎕NEW Queue
      QUEUE23              ⍝ Default display of APL object reference
[Queue]
```

However, if the APL programmer wishes to override the default display form of an object, this can easily be done by using the `⎕DF` system method (see the section on system methods below):

```
      QUEUE23.⎕DF 'Checkout Queue23'

      QUEUE23
Checkout Queue23
```

Object references and object lifetimes

When you use `⎕NEW` to create a new object, that object persists until there are no more references to it in the workspace. It is then deleted immediately, if it is an internal or system object. If it is an external object, such as an instance of a .Net class, the fact that there are no more references to it in the APL workspace means that it available for deletion by the external environment (unless the external environment itself has further references to the same object). However, in typical external environments such as .Net, Java and Ruby, the actual deletion of the object may not occur until later.

Consider this sequence, where we create an instance of a class called Philosopher which has a property Name:

```
      A←⎕NEW Philosopher
      A.Name←'Aristotle'
```

At this point, we have created a new instance of the class, and we have a single reference to it, in the variable A. We now copy the reference (not the object itself) to a variable B:

```
      B←A
      B.Name
Aristotle
```

We now have two references to the same object. So if we change a property of the object, the change is visible through either reference - they refer to the same thing:

```
      B.Name←'Socrates'
      A.Name
Socrates
```

Now we erase one of the references:

```
      )ERASE A
```

We still have a second reference to the object. The object will persist until we delete the last reference to it:

```
      B.Name
Socrates
      )ERASE B
```

At this point, there are no more references to the object left in the workspace, and the object itself is deleted.

It follows from this that, if you use `⎕NEW` to create an object, and do not assign the result to a variable, it will immediately be deleted again. In this example, we create an instance of the class Philosopher. The explicit result of `⎕NEW` is a temporary workspace entry (of type object reference), which is displayed using the default display format for objects, and then deleted. At that point the object itself is also deleted, as there are no references left:

```
     ⎕NEW Philosopher
[Philosopher]
```

The Null object

As its name implies, the Null object is a special case of an object, which has no properties and no methods of its own (although System methods may apply to it). A reference to the Null object displays in the special form:

```
[NULL OBJECT]
```

A reference to the Null object can arise for a number of different reasons:

- If you have an array of object references, the prototype of the array is a reference to the Null object. For example:

```
•
•       VEC←⎕NEW ¨Rectangle Sphere Triangle
•       VEC
•  [Rectangle] [Sphere] [Triangle]
•       1↑3↓VEC
[NULL OBJECT]
```

- An external call may return a Null object, for example if you are looping through a linked list of objects and reach the last one.

- An APLX System method may return a Null object, for example if you ask for the parent class of a top-level class:

```
•
•      Point.⎕PARENT
[NULL OBJECT]
```

- Your application code can deliberately set an object reference to Null (by calling `⎕NULL`), for example to indicate that it has not yet been initialized.

- APLX may be forced to set an object reference to Null, because it is no longer valid. For example, this will happen if you `)SAVE` a workspace which contains a reference to an external object (e.g. a Java or .NET object). On re-loading the workspace at a later date, the object reference is no longer valid since the external object no longer exists.

Types of Property

When you define a class, you specify the names of the properties of that class, which can be used to hold data associated with the class. You can optionally specify a default value for the property, that is the value which the property will have in a newly-created instance of the class. You can also specify that the property is read-only, which means it is not possible to assign a new value to it.

Most properties are *instance properties*, which means that each instance of the class has a separate copy of the property (for example, the X and Y position of a Shape). Occasionally, however, it is useful to define a *class-wide* property (known in some other languages as a *static* or *shared* property). This is a property where there is a single copy of the data, shared between all instances. This is useful for cases such as keeping a unique incrementing serial number (the next invoice number, for example).

Combining these concepts, you have the following main types of property:

- A read-write instance property, with a default value specified in the class definition
- A read-write instance property, with no default value specified in the class definition
- A read-write class-wide property, with a default value specified in the class definition
- A read-write class-wide property, with no default value specified in the class definition
- A read-only class-wide property, with a default value specified in the class definition

You can also in principle have a read-only property with no initial value, but this is not very useful! You can also have a read-only instance property, but this is indistinguishable from a read-only class-wide property because you can't assign a different value to it in different instances.

Implementation note: APLX uses a 'create-on-write' approach when you assign to an instance property. This means that, if you have never changed the value of a property for a particular instance since the instance was first created, the value which is returned when you read the property is the default value stored in the class definition. It follows that, if you change the class definition so that the property has a different default value, the change will immediately be reflected in all instances of the class, unless the property has been modified for that instance.

Name scope, and Public versus Private members

The members of a class (i.e its properties and methods) can be either public or private. Public members can be accessed from outside the class, whereas private members can only be accessed from within methods defined in the class (or from desk calculator mode, if a method has been interrupted because of an error or interrupt and the method is on the `)SI` stack). Private members can also be accessed by methods defined in a child (derived) class. If you are familiar with other object-oriented languages such as C++ or Visual Basic, this means that private methods in APLX correspond to 'protected' methods in those languages.

If you want to access a public member of an object from outside the class (i.e. not within a method of the class), then you use *dot notation* to refer to it. This takes the form ObjectReference.MemberName. For example, suppose you have a variable `myrect` which is a reference to an object of class `Rectangle`. You could call the `Move` method and access the `X` and `Y` properties for that object as follows:

```
      myrect.X←45
      myrect.Y←78
      myrect.Move 17 6
      myrect.X
62
      myrect.Y
84
```

Within the methods of the class itself, you do not normally need to use dot notation. This is because the search order for symbols encountered when executing a method is as follows:

1. First, APLX looks to see if the symbol refers to a member defined in the class of the object.
2. If not, it looks to see if the member is defined in the parent class (if any), iterating through each of the ancestors in turn.
3. If it is not found in any of the ancestors, it then looks in the local variables of the method.
4. Finally, it looks in the global symbol table.

Thus, a simple implementation of the Move method above (defined in the Shape class from which Rectangle derives) might be something like this:

```
    ∇ Move B
[1]   ⍝ Move shape by amount B specified as change to X, Y
[2]   (X Y)←(X,Y)+B
    ∇
```

Constructors

As we saw earlier, a constructor is a special type of method, which is run automatically when an instance of a class is created using ⎕NEW. It can be used to initialize the object, optionally using parameters passed to ⎕NEW. For example, you might use this mechanism to specify the initial position of a Rectangle object.

For a user-defined class, a constructor is defined as a method which takes a right argument, and which has the same name as the class itself.

In some other object-oriented programming languages, constructors are a very important part of the language because they are the only way of initializing property values. For user-defined classes in APLX, default values can be set up in the class definition, so constructors are not always needed.

Where a class inherits from another class, the constructor which gets run automatically is that of the class itself (if it has a constructor), or of the first ancestor class which has a constructor. Normally, in a constructor, you will want to do some initialization specific to the class itself, and also call the constructor of the parent class (using ⎕PARENT) to do any initialization which it and its ancestors require. You can do this at any point in the constructor; there is no restriction on where you make this call to the parent's constructor; indeed, you don't have to call it at all if it is not appropriate.

In APLX, a constructor is also a perfectly ordinary method; it can be called in the normal way by one of the other methods in the class, or from outside (if it declared as Public). This can be useful for re-initializing an object.

Some object-oriented languages also include a special method called a *destructor*, which is called just before the object is deleted. APLX user-defined classes do not have destructors. This means that, if you need to release system resources (for example, close a file or a database connection), you need to call a method to do that explicitly before erasing the last reference to the internal object. However, APLX will automatically take care of deleting all the properties of the object, and releasing the memory back to the workspace.

Using Classes without Instances

So far, we have concentrated on using objects as instances of classes. However, classes can also be very useful in their own right, without the need to make instances of them. There are two major reasons why you might want to define a class which can be used directly:

Defining a set of constants

If you define a class with a set of read-only properties, those properties can be used as a set of constant values or 'enumerations'. For example, you might have a class called Messages, which holds all the messages which your application displays to the user:

```
Messages {
OutOfMemory←←'There is not enough memory to continue'
AskModelName←←'Enter the name of the model'
OpComplete←←'Operation Complete'
AskReset←←'Do you want to reset the model?'
...etc
}
```

You can then use this class in your application (without having to make an instance of it) to encapsulate all the messages and refer to them by name:

```
      ∇R←CheckWS
[1]   :If R←⎕WA<MIN_FREE_WS
[2]     ShowError Messages.OutOfMemory
[3]   :EndIf
      ∇
```

This keeps all the messages together in one place, allows you to refer to them by a name which is easy to remember and is self-documenting, but does not pollute the global symbol space with hundreds of APL variables.

Keeping namespaces tidy

In traditional APL systems, it often used to be the case the number of global functions was very large. By placing related functions in a class, the workspace can be kept tidy.

For example, in a statistical application, you might have a class `Average` which contained methods for calculating many different types of average (`Mean`, `Median`, `Mode` etc). As long as these methods do not write to any property of the class, there is no need to make an instance of the class to run them; you can just run them using dot notation as `Average.Mean`, `Average.Median` etc.

Note that, in APLX classes, there is no pre-determined difference between a method which can only be run inside an instance (sometimes known as an *instance method*), and a method which can be run as a class member without an instance being created (sometimes known as a *static method*). The only difference is that, at run time, if a method writes to a property, an error will be generated if there is no instance to write to.

However, you do need to be aware of the difference between static and instance methods when using classes written in other languages such as Java or C#. See the system function `⎕CALL` for more details.

Editing User-Defined Classes

You can create and edit user-defined classes in a number of ways:

- Using the on-screen class editor, invoked from the Edit menu or `)EDIT`. The class editor allows you to edit each method of the class, as well as set up properties and default values;
- Using the line ('del') editor;
- Using the system function `⎕FX`, to convert a text representation into a class;

- Using the system function `⎕IC`, to transfer global functions, operators and variables into the class as methods and properties.

Mixins

What are Mixins?

As we saw in the previous sections, classes which you write in APLX can inherit from other classes; this means that the methods and properties of the parent class (or classes) are available in the child class.

Although the concept of inheritance is very powerful, there are some circumstances where more flexibility is required. In APLX, a class cannot inherit from multiple different classes, only from one parent class (although that might itself inherit from its parent, and so on). Nor can a class inherit from an external class; for example, you cannot write an APL class which directly inherits from a Java class.

'Mixins' address both of these requirements. They allow you to extend your user-defined classes so that, at run-time, they dynamically 'mix in' functionality (i.e. methods and properties, and perhaps events) from one or more other classes; these can be internal (user-defined, and written in APL), or external (.Net, Java, Ruby etc, or a built-in APLX system class).

Because mixins are attached dynamically at runtime, they are very flexible. For example, in a commercial application you might have an `Invoice` class (which perhaps inherits from an `AccountingDocument` class). If you wanted to add functionality which would allow the `Invoice` class to be faxed or e-mailed to the client, you could dynamically (at run time) mix-in a `Fax` or `EMail` class to handle the transmission of the document. This is similar to multiple inheritance as implemented in some other languages, but more flexible because you don't need to know in advance which mixin will be required; different instances of the same class can, if appropriate, mix-in different classes.

When you 'mix-in' another class, what effectively happens is that a new object of the mixed-in class is created, and merged into the original object. The public properties and methods of the mixed-in class now become available in the original object, very much as though they were defined in the original class.

You can mix-in as many other classes as you like; you can even mix in classes from multiple different architectures. For example, you could write (in APL) a `FinancialClock` class to display the time in London, New York and Singapore. It could mix-in the System Class `Window` for the display, and the Java class `timeZone` to handle the different time-zone information.

Using Mixins

To use mix-ins, you first create an object (i.e., an instance of your APL class) in the normal way using `⎕NEW`. You then use the System Method `⎕MIXIN` to mix another class into the object. `⎕MIXIN` has a similar syntax to `⎕NEW`; the right argument is the class reference (or name, as a text vector), followed by any arguments to the constructor for the class you are mixing-in. The left argument can be omitted if you are mixing-in an APL class, otherwise it defines the architecture for the mix-in. For example, if you have a class called `Invoice`, and another class called `Fax`, you can mix the `Fax` class into an `Invoice` object as follows:

Create an instance of Invoice:

```
      inv←⎕new 'Invoice'
      ⍝ Properties:
      inv.⎕nl 2
customer
invoice_number
lines
order_number
      ⍝ Methods:
      inv.⎕nl 3
SetStatus
```

Mix class Fax into the Invoice object:

```
      inv.⎕mixin 'Fax'

      ⍝ Properties and methods now include those of Fax class:
      inv.⎕nl 2
cover_page          ⍝ <--- From Fax class
customer
fax_number          ⍝ <--- From Fax class
invoice_number
lines
order_number

      inv.⎕nl 3
Send                ⍝ <--- From Fax class
SetStatus
```

You can mix-in further classes in the same way.

Although in this example we have mixed-in the Fax class (using dot notation) after creating the original object, in many cases the natural place to do this will be in the Constructor of the original class. If you do that, the mix-in facility effectively becomes like multiple inheritance in some other languages.

Mixing-in an external class

You can mix an external class (.Net. Java, Ruby, or a built-in APLX system class) in to your APL class in the same way. In this case, you need to provide a left argument to `⎕MIXIN` to specify the architecture, in the same way as you would with `⎕NEW`. For example, we could add a second mixin, based on a Java class, to the `Invoice` class shown in the example above. All the properties and methods of the Java class then become available in the object:

```
      'java' inv.⎕mixin 'java.util.Date'
      ⎕box inv.⎕nl 3
Send SetStatus UTC after before clone compareTo equals getClass getDate getDay
      getHours getMinutes getMonth getSeconds getTime getTimezoneOffset getYear
      hashCode notify notifyAll parse setDate setHours setMinutes setMonth
      setSeconds setTime setYear toGMTString toLocaleString toString wait
      inv.toLocaleString
20-Mar-2009 11:43:03
```

Referencing the mixed-in object directly

Sometimes you may need to access the underlying object which has been merged into your APL object. For this, you need a reference to the underlying object. You can get this in two ways:

(1) `⎕MIXIN` actually returns as an explicit result the underlying object reference (but with display potential switched off, as a 'shy' result). So you can assign this to a variable or property of your APL class, and use this to call the underlying object directly:

```
      jd←'java' inv.⎕mixin 'java.util.Date'
      jd.⎕classname
java:java.util.Date
```

(2) You can use the system method `⎕MIXINS` to get a vector of references to the mixins:

```
      my_mixins←inv.⎕mixins
      my_mixins
[Fax] [java:Date]
      my_mixins[2].⎕classname
java:java.util.Date
```

Search order and over-riding a method

When a member of the class is referenced (either using dot notation, or as unadorned symbols when running methods of the class), APLX will use the following search order to find the named symbol:

- First it will search the original class, (and its parent classes, if any)
- Then it will search in the first mixin (and its parent classes, if any)
- If there are further mixins, it will search these in the order in which they were mixed-in.

It follows from this that you can 'over-ride' a property or method from a mixed-in class; if your own APL class defines a member of the same name as a member of the mixed-in class, the APL version will be the one which is accessed; the mixed-in version will be hidden.

However, you can still call the mixed-in version by accessing it directly using the object reference returned either when it is created (explicit result of `⎕MIXIN`), or from `⎕MIXINS`. In our example, you could define a method `toString`, which overrides the Java version, but calls it to get the date as text:

```
      ∇r←toString
[1]   ⍝ String representing invoice
[2]   r←'Invoice number ',(⍕invoice_number),' dated ',inv.⎕mixins[2].toString
[3]   ∇

      ⍝ Insert toString as a method into class Invoice:
      'Invoice' ⎕ic 'toString'
1
      inv.toString
Invoice number 11345301 dated Fri Mar 20 11:57:32 GMT 2009
```

Removing mixins from an object

The System Method ⎕UNMIX can be used to remove one or more mixins from an object. It takes a right argument which is a scalar or vector list of mixin-references to delete, and returns a binary vector with 1 for each mixin removed, and 0 if the mixin reference could not be found:

```
      inv.⎕mixins
[Fax] [java:Date]
      inv.⎕unmix inv.⎕mixins
1 1
      inv.⎕mixins

      inv.⎕nl 3
SetStatus
toString
```

Note that you don't normally need to do this; the mixins will be deleted automatically when the object which owns them is deleted.

Branching and labels

Traditionally, the APL right arrow '→' has been used to control execution in user-defined functions and operators. It can be used as a conditional or unconditional branch, and thus allows conditional execution and loops to be programmed. (Note that APLX alternatively allows you to control execution using structured-control keywords, which are preferable in many contexts).

The symbol → is usually followed by an integer scalar, vector, or label name which identifies the line to branch to. If the argument is a vector, the first element of the vector determines the line at which execution will continue, and subsequent elements are ignored. If the line number does not exist, the function terminates (often a line number of 0 is used for this purpose). If the argument is an empty vector, no branch is taken and execution continues at the next statement. Thus, conditional branches can be programmed by using a right argument which, at run-time, evaluates either to an integer scalar/vector, or to an empty vector.

A label is a name which is followed by a colon. It is placed at the start of the line which it identifies it. When the function is running, it is treated as a local variable whose value is the number of the line on which it is placed. It can thus be used directly as the argument of the right arrow.

A special case arises if no argument is given to the right arrow (a 'naked branch'). This terminates execution of the current function and of all functions which called it, removes them from the state indicator, and returns to desk-calculator mode. If the APL interpreter is already in desk-calculator mode, this will have the effect of removing the top function and all thouse down to the next function marked with an asterisk in the `)SI` display. A naked branch can also be used to end `⎕` input.

Examples:

To branch back from line 10 to line 3:

```
[10] →3
```

To branch unconditionally to a line labelled `LAB2`:

```
[10] →LAB2
[11] ...
[12] LAB2:TOTAL←QTY×PRICE
```

To branch to a line labelled `LAB2` only if `LOOP` has the value 10, by using an expression which evaluates to an empty vector if the condition is not true or to the label value if it is true:

```
[10] →(LOOP=10)/LAB2
[11] ...
[12] LAB2:TOTAL←QTY×PRICE
```

To branch to one of several lines depending on the value of the variable `INDEX`:

```
[6] →(CASE1 CASE2 CASE3)[INDEX]
```

To branch to one of several lines using a boolean vector to select which (execution will continue at the label corresponding to the first 1 in the vector. If there is none, a message will be displayed and the function will end):

```
      [6] →SELECT/(CASE1 CASE2 CASE3) ⋄ 'No case applies' ⋄ →0
```

Control Structures

As well as the conventional branch arrow, APLX supports *structured-control keywords* for flow control, often making for more readable functions.

The structured control keywords are not part of the International Standards Organisation (ISO) specification of the APL language, but they are supported by a number of APL implementations including APLX.

The structured control keywords include:

Function	***Keywords***
Conditional execution	`:If :OrIf :AndIf :ElseIf :Else :EndIf`
For loop	`:For :In :Leave :Continue :EndFor`
While loop	`:While :Until :Leave :Continue :EndWhile`
Repeat loop	`:Repeat :Until :Leave :Continue :EndRepeat`
Case selection	`:Select :Case :CaseList :Else :EndSelect`
Error trapping	`:Try :CatchIf :CatchAll :EndTry`
Terminate current function	`:Return`
Branch	`:GoTo`

Note: The general keyword `:End` can be use in place of any of `:EndIf :EndFor :EndWhile :EndRepeat :EndSelect :EndTry`

Using Control Structures

The keywords all begin with a colon character, and usually appear at the start of the line (APLX will automatically indent lines within a block for you). For example:

```
     ∇ITERATE N
[1]   :If N<0
[2]     'Negative argument not supported'
[3]     :Return
[4]   :EndIf
[5]   ...
```

You can also place a block on a single diamond-delimited line:

```
     ∇ITERATE N
[1]   :If N<0 ⋄ 'Negative argument not supported' ⋄ :Return ⋄ :EndIf
[2]   ...
```

Multi-line sequences can be nested to any depth, but single-line sequences cannot contain further nested control structures. *Note:* The single-line form cannot be used with `:Try..:EndTry`.

The APLX function editor prompts you with the correct indentation as you type. If you cut or paste lines, you can clean up the indentation from the Edit menu (or press Ctrl-I in Windows, Cmd-I under MacOS)

The supported set of structured-control phrases is as follows (items in square brackets are optional). You can end any sequence with `:End` rather than the more specific ending keyword shown. Note that in APLX, structured-control keywords are not case-sensitive when you enter them, but APLX will re-display them in the case shown.

Conditional execution

Syntax:

```
:If <boolean expression>
...
[:ElseIf <boolean expression>]
...
[:Else]
...
:EndIf
```

The expression following the `:If` keyword is evaluated. If it is true, the block which follows it is executed, until an `:ElseIf`, `:Else`, `:End` or `:EndIf` is encountered, at which point execution transfers to the statement after the `:End` or `:EndIf`. If the expression is false, the same procedure is followed for any `:ElseIf` blocks in the sequence. If none of the tests is true, the `:Else` block (if any) is executed. It is permissible to have as many `:ElseIf` sections as you like.

For example, this function returns a string which depends on the value of B:

```
      ∇R←CLASSIFY B
[1]   :If B=0
[2]     R←'Zero'
[3]   :ElseIf B>0
[4]     R←'Positive'
[5]   :Else
[6]     R←'Negative'
[7]   :End
      ∇
```

You can also add `:AndIf` or `:OrIf` phrases after an `:If` or `:ElseIf` phrase. If you use `:AndIf`, each expression must be true for the block to be executed, whereas if you use `:OrIf` only one of them needs to be true. (The `:AndIf` and `:OrIf` conditional expressions are evaluated only if necessary). For example:

```
      ∇R←A CLASSIFY B
[1]   :If B=0
[2]   :AndIf A=0
[3]     R←'Both arguments are zero'
[4]   :ElseIf B=0
[5]   :OrIf A=0
[6]     R←'One argument is zero'
[7]   :Else
[8]     R←'Neither argument is zero'
[9]   :End
      ∇
```

For loop

Syntax:

```
:For <control variable name> :In <vector expression>
...
:EndFor
```

The control variable is assigned successive values from the vector expression and the loop is executed once for each value. The values can be of any type, not just integers. The vector expression is evaluated only once, at the start of the loop. For example:

```
      :For W :In "It's" "Off" "To" "Work" ⋄ 'HiHo' W ⋄ :EndFor
 HiHo It's
 HiHo Off
 HiHo To
 HiHo Work
```

You can use the `:Continue` keyword within the loop to force premature termination of a particular iteration - execution continues at the top of the loop with the next value (if there is one). You can also use the `:Leave` keyword to exit the loop completely and continue execution with the line after the `:EndFor`.

While loop

Syntax:

```
:While <boolean expression>
...
:EndWhile
```

If the boolean expression is true (value 1), the loop body is executed. At the end, control returns to the `:While` statement and the loop is re-executed as long as the boolean expression remains true.

```
      ∇Evaluate B
[1]   :While B>0
[2]     B←NextNode B
[3]   :EndWhile
      ∇
```

An alternative form allows a test at the end of the loop as well:

Syntax:

```
:While <boolean expression>
...
:Until <boolean expression>
```

The `:Continue` and `:Leave` keywords can again be used to force an early termination of a particular iteration or of the whole loop.

Repeat loop

Syntax:

```
:Repeat [<integer expression>]
...
:EndRepeat
```

The loop body is repeated a maximum of N times, where N is the value of the integer expression (evaluated only once, at the start of the loop). If the integer expression is omitted, the loop is repeated for ever, unless terminated in another way. For example:

```
      :Repeat 3 ⋄ ⎕TS ⋄ :EndRepeat
2002 7 30 14 36 2 228
2002 7 30 14 36 2 228
2002 7 30 14 36 2 228
```

The `:Continue` and `:Leave` keywords can again be used to force an early termination of a particular iteration or of the whole loop:

```
      ∇GUESS;VAL
[1]   'Guess a number'
[2]   :Repeat
[3]     VAL←⎕
[4]     :If VAL=231153
[5]       'You were right!'
[6]       :Leave
[7]     :EndIf
[8]     'Sorry, try again..'
[9]   :EndRepeat
      ∇
```

You can also end the loop with an `:Until` statement so that execution repeats only if a boolean expression remains true:

```
:Repeat [<integer expression>]
...
:Until <boolean expression>
```

Case selection

Syntax:

```
:Select <expression>
:Case <expression>
...
[:CaseList <vector expression>]
...
[:Else]
...
:EndSelect
```

The `:Select` expression can be any APL scalar or array. It is matched against each of the `:Case` expressions (or elements of the `:CaseList` expressions) in turn. If they match in value and shape (using the same rules as the APL `≡` (match) primitive), the body of lines following is executed, until the next control keyword in the sequence is reached when execution jumps to the line following the

:EndSelect (For :CaseList the match is done against each of the elements of the vector expression in turn, and if any of them match then the test is regarded as true). If none of the expressions match, the :Else clause (if any) is executed. For example:

```
      ∇R←CLASSIFY B;⎕IO
[1]   ⎕IO←1
[2]   :Select B
[3]   :Case 0
[4]     R←'Scalar zero'
[5]   :Case 1⍴0
[6]     R←'Length 1 vector, value 0'
[7]   :CaseList ⍳10
[8]     R←'Scalar in the range 1 to 10'
[9]   :Else
[10]    R←'None of the above'
[11]  :EndSelect
      ∇

    CLASSIFY 0
Scalar zero
    CLASSIFY 2
Scalar in the range 1 to 10
    CLASSIFY 1⍴2
None of the above
    CLASSIFY 1⍴0
Length 1 vector, value 0
```

Error Trapping

Syntax:

```
:Try
...
[:CatchIf <boolean expression>]
...
[:CatchAll]
...
:EndTry
```

The block of code following the :Try keyword is executed, until either an error occurs, or a :CatchIf, :CatchAll, :End or :EndTry is encountered. (Unlike the other control structures, :Try...:EndTry blocks cannot be placed on a single line).

If no error has occurred within the :Try block, execution transfers to the line after the :End or :EndTry.

If an error occurs in the :Try block (either in the statements in this function, or in any functions called from it), control transfers to the first :CatchIf line, and the expression is evaluated. If it is true, the block of code following the :CatchIf is executed, and execution then resumes after the :EndTry or :End. If the expression is false, the same procedure is followed for any further :CatchIf blocks in the sequence. If none of the tests is true, the :CatchAll block (if any) is executed. It is permissible to have as many :CatchIf sections as you like.

Typically, you use the :CatchIf statement to catch specific types of error, by looking at ⎕LER or ⎕ET. See the section Error trapping using :Try..:EndTry for more information.

Miscellaneous keywords

The `:GoTo` keyword (followed by a line label name) can be used to branch directly to a label. It is equivalent to using a conventional APL → symbol to branch to a label. You can branch to a label inside the same control structure, or to a label outside the control structure, but not to a label which is more deeply nested than the line you are branching from.

The `:Return` keyword causes the current function execution to terminate. It is equivalent to a conventional APL branch to line 0.

Named loops

Normally the `:Continue` and `:Leave` keywords apply to the current loop in which they are executed, so that if you have a loop nested within a loop, execution resumes at the start or end of the innermost loop. However, you can also name loops by including a label at the start line, and follow the `:Continue` or `:Leave` with the name to apply it to a particular level of nesting. In this example, if the `:If` clause is true, execution continues at line 9:

```
    ∇ITERATE;N
[1]  OUTER: :Repeat
[2]    :For N :In SAMPLES
[3]      :If INTERRUPTED
[4]        :Leave OUTER
[5]      :EndIf
[6]      ...
[7]    :EndFor
[8]  :EndRepeat
[9]  ...
```

Errors

If there is an error in the usage of structured keywords, APLX will report a `STRUCTURED CONTROL ERROR`. This typically arises if the keywords at the beginning and end of the block do not match up correctly, or if you branch to a line label within a control structure without executing the initial keyword line at the start of the block.

Note that APLX does not prevent you from fixing a function which is syntactically incorrect. However, the APLX editor will warn you of mismatched structured-control keywords if you select 'Clean up indentation' from the Edit menu (Ctrl-I in Windows, Cmd-I under MacOS).

System commands

APLX supports a range of commands ('system commands') which are used to communicate directly with the system. They are not part of the APL language itself. APL system commands start with a right parenthesis:

```
              )SAVE
```

The display generated by a system command cannot directly be used as the argument to a function. However, in APLX, system commands can be executed using the ⍎ (execute) primitive:

```
              ∇LIB
      [1] ⍝ Show contents of library 0
      [2] ⍎')LIB'
      [3] ∇
```

The output from executed system commands can be captured in a variable or passed as an argument to a function:

```
            X←⍎')SYMBOLS'
            X
      IS 1026, USED 21
```

See the reference section on System Commands for a full list of available commands.

System Functions and Variables

APL system functions implement a wide variety of system-related or utility features. They are built-in to the APL interpreter, but often call out to the operating system to perform some function (such as reading from a database, or fetching the current date and time). They have names of one or more characters and start with a `⎕` (`⎕BOX` `⎕TS` `⎕WS` etc).

System functions can be niladic, monadic or dyadic.

See the reference section on System Functions and Variables for a full list.

System Methods

Just as traditional APL interpreters have system variables and system functions (whose names all begin with the `⎕` character), system methods are pre-defined methods (also with names beginning with `⎕`) which apply to internal user-defined object classes, and in most cases to external classes as well.

You can call a system method exactly as you would call an ordinary method of a class, either using dot notation (for example `MyPoint.⎕CLASSNAME`), or (within a user-defined class method) by simply using the system method's name such as `⎕CLASSNAME` (equivalent to `⎕THIS.⎕CLASSNAME`).

System methods can be niladic, monadic or dyadic.

See the reference section on System Methods for a full list.

System Classes

A *System Class* is a pre-defined class which is part of APLX. They are mostly used for user-interface programming. Examples are the `Form`, `Timer`, `ChooseColor` and `Chart` classes. (In previous versions of APLX, these classes were accessed through `⎕WI`. Although you can continue to use `⎕WI`, you may find the new class-based syntax more readable and more consistent.)

To create an instance of a top-level System class (such as a Form or a pre-defined dialog), you provide the name of the class as the right argument to `⎕NEW`, and use `'⎕'` as the left argument to indicate that this is a System class:

```
      DLG←'⎕' ⎕NEW 'ChooseColor'
      DLG.⎕NL 3
Close
Create
Delete
New
Open
Send
Set
Show
Trigger
```

You can then use dot notation to access the properties and methods of the object:

```
      DLG.color←234 23 56
      DLG.Show
1
```

See the separate manual on *System Classes and User-Interface Programming* for more details.

Files and Databases

APLX offers a range of features for accessing data in files. These include facilities both for storing and retrieving data within your APL applications, and for exchanging data with other applications. They include:

Component Files

For simple APL applications, you can often keep all the data you need in the current workspace in APL variables. However, for more sophisticated applications, this may not fit your requirements. For example, if you wrote a suite of functions which produced monthly profit and loss accounts, you might want to store the data for each month separately. You could arrange to keep the data in a series of stored workspaces, but you would not want to replicate the functions in each of these workspaces.

Component files provide an efficient and easy-to-use method to store APL variables (of any type, shape, and size) in a file, and read them into the workspace when they are needed. Each individual item in the file is known as a *component*. A single number may constitute one component, while a matrix containing several thousand numbers may be its next door neighbour.

Functions, operators and classes can also be stored in component files, but they must first be converted into character arrays by the system function `⎕CR`, or stored via the overlay system function, `⎕OV`.

APLX supports two different component file systems. The first is based on system functions such as `⎕FTIE`. It uses a syntax which is compatible with APL interpreters from other vendors.

The second of these is based on the file-access primitives `⍇ ⍈ ⍐ ⍗`, as implemented in the predecessor to APLX, APL.68000.

For more information, see the separate section on Component File Systems.

Native Files

'Native' files are operating-system files which are not necessarily associated with APL, and which are typically used for exchanging data with non-APL applications. For example, they might include text files, HTML pages, or binary files produced by a Fortran application. Unlike component files (which retain information about the type and shape of APL data), the structure of native files is unknown to APLX, so you as the programmer are responsible for specifying how the data should be interpreted. For example, you can specify that you want to read the first four bytes of a file as an integer, and the next 32 bytes as a character vector.

See the section on Native File Functions for details.

System Functions for Data Import/Export

Although native files provide a general, low-level way to exchange data with other applications, there are a number of common file formats for which APLX provides an easier alternative, by means of the `⎕IMPORT` and `⎕EXPORT` functions. These allow you to read or write the entire contents of a file in a single call. They support a number of common file formats, for example Unicode text, or Comma-Separated Variable (CSV) files used for spreadsheets. The advantage of `⎕IMPORT` and `⎕EXPORT` is that you do not have to write any code to interpret the file format yourself, since APLX already has the necessary logic built-in.

Accessing Database Records

Much of the data in modern computer systems, especially for large commercial applications with many thousands of records, is held in relational databases. These are accessed and updated using SQL ('Structured Query Language'). You can easily interface to such databases by using the `⎕SQL` system function. This allows you to read and write records in most major commercial database systems including Oracle, SQL Server and DB2, as well as open-source databases such as MySQL and PostgreSQL. You can also exchange data with popular desktop file systems such as Microsoft Access.

Other facilities

In addition to the above facilities for directly reading and writing data into the APL workspace, APLX provides a number of System Classes which can manipulate specific types of files such as images and movies. See the documentation on the Picture, Movie, Image classes in the separate manual *System Classes and User-Interface Programming*.

Section 2: APL Primitives

+ Conjugate

One-argument form *See also two-argument form* Add

`+` returns the value of the numeric expression to its right. (See also ⎕ output which can be used for a similar purpose.) Contrast the first example below, which places the result of a multiplication in COST, but does not display it, with the second:

```
      COST  ← 320×8.56          (No result displayed)
      +COST ← 320×8.56          (Result displayed)
2739.2
      +(¯1 0 17) (2 3⍴⍳6)       (Argument displayed unchanged)
 ¯1 0 17    1 2 3
            4 5 6
```

+ Add

Two-argument form *See also one-argument form* Identity (Conjugate)

Adds the numbers in the right and left-hand arguments:

```
      12+3                      (Adds two scalars)
15
      3 7 2+0 1 ¯4              (Adds the corresponding numbers
3 8 ¯2                           in vectors of equal size)
      11+65 23 98 3             (Adds 11 to each number in a vector)
76 34 109 14
      TABLE ← 2 3 ⍴ ⍳ 6         (Puts the numbers 1 to 6 in TABLE)
      10+TABLE                  (Adds 10 to each number in TABLE)
11 12 13
14 15 16
      TABLE+TABLE               (Adds corresponding numbers in
2 4 6                           matrices of equal size and dimensions)
8 10 12
      1 +5 (2 2 ⍴⍳4) (⍳5)       (Scalar left argument added to all
 6    2 3   2 3 4 5 6            elements of right argument)
      4 5
      2  3+5 (2 2 ⍴⍳4) (⍳5)     (Arguments must be the same length)
LENGTH ERROR
      2 3+5(2 2⍴⍳4)(⍳5)
      ^
      (2 2⍴⍳4) 10 (5⍴3)+5 (2 2 ⍴⍳4) (⍳5)
6 7    11 12   4 5 6 7 8        (Corresponding elements added)
8 9    13 14
```

- Negate

One-argument form *See also two-argument form* Subtract

Reverses the signs of the right-hand argument:

```
      - 5 6 2
¯5 ¯6 ¯2
      - ¯1 4 ¯6 1
1 ¯4 6 ¯1
      -(¯11 17 ¯23) (2 3⍴¯3+⍳6)
 11 ¯17 23     2  1  0
              ¯1 ¯2 ¯3
```

- Subtract

Two-argument form *See also one-argument form* Negate

Subtracts the number(s) in the right-hand argument from the number(s) in the left-hand argument:

```
       240-1                  (Subtracts one number from another)
239
       8 4 6-1 0 2            (Subtracts each number in a vector from
7 4 4                          the corresponding number in a vector of
                               equal size)
      22 ¯4 15-1              (Subtracts 1 from each number in a
21 ¯5 14                       vector of numbers)
      TABLE ← 3 3 ⍴ ⍳ 9
      100-TABLE               (Subtracts each number in a matrix
99 98 97                       from a scalar)
96 95 94
93 92 91
      TABLE-TABLE             (Subtracts each number in a matrix
0 0 0                          from the corresponding number in a
0 0 0                          matrix of equal size and dimensions)
0 0 0
      1 - 5 (2 2 ⍴⍳4) (⍳5)    (Each element in the right argument is
 ¯4     0 ¯1    0 ¯1 ¯2 ¯3 ¯4   subtracted from 1)
       ¯2 ¯3
      2  3-5 (2 2 ⍴⍳4) (⍳5)   (Arguments must be the same length)
LENGTH ERROR
      2 3-5(2 2⍴⍳4)(⍳5)
      ^
      (2 2⍴⍳4) 10 (5⍴3)-5 (2 2 ⍴⍳4) (⍳5)
  ¯4 ¯3    9 8   2 1 0 ¯1 ¯2  (Corresponding elements subtracted)
  ¯2 ¯1    7 6
```

Note: Remember that APL uses a special symbol (the high minus ¯) to indicate negative numbers. You will see some examples above.

× Sign of

One-argument form *See also two-argument form* Multiply

Shows the sign of the number(s) in the right-hand argument. Each positive number is represented by a 1, each negative number by a ¯1 and each zero by a 0.

```
      ×33 98 0 ¯5
1 1 0 ¯1
      ×(¯33.1 0 27) 55 (2 2ρ¯2+ι4)
 ¯1 0 1  1    ¯1  0
               1  1
```

× Multiply

Two-argument form *See also one-argument form* Sign of

Multiplies the number(s) in the right-hand argument by the number(s) in the left- hand argument.

```
      23.8×0.12            (Multiplies one number by another)
2.856
      12 8 39×9 81 2       (Multiplies each number in a vector by
108 648 78                  the corresponding number in a vector of
                            equal size)
      12×89 91 ¯2 87       (Multiplies a scalar by each number in
1068 1092 ¯24 1044          a vector)
      TABLE ← 3 5 ρ ι 15
      TABLE × 5            (Multiplies each number in a matrix
 5 10 15 20 25              by a scalar)
30 35 40 45 50
55 60 65 70 75
     TABLE×TABLE           (Multiplies each number in a matrix
  1   4   9  16  25         by the corresponding number in a matrix
 36  49  64  81 100         of equal size and dimensions)
121 144 169 196 225
      2 × 5 (2 2 ρι4) (ι5) (Multiplies every element in the right
 10     2 4   2 4 6 8 10    argument by 2)
        6 8
      2  3 × 5 (2 2 ρι4) (ι5)
LENGTH ERROR               (Arguments must be the same length)
      2 3×5(2 2ρι4)(ι5)
      ^
      (2 2ρι4) 10 (3ρ3)×5 (2 2 ρι4) (ι3)
   5 10    10 20   3 6 9   (Corresponding elements multiplied)
  15 20    30 40
```

÷ Reciprocal

One-argument form *See also two-argument form* Divide

Gives the reciprocal of the right-hand argument, that is, the result of dividing 1 by each number in the right-hand argument.

```
      ÷2                        (Reciprocal of 2, ie 1 divided by 2)
0.5
      ÷10 5 1 ¯1                (Reciprocal of each number in a vector)
0.1 0.2 1 ¯1
      ÷.5 .25 .01               (Reciprocal of each number in a vector)
2 4 100
      ÷(2 2ρ1 2 4 5)(÷ι4)       (Reciprocal of each element in the vector)
 1                  0.5               1 2 3 4
 0.25               0.2
```

If the right argument contains a zero, APLX will generate a `DOMAIN ERROR`.

÷ Divide

Two-argument form *See also one-argument form* Reciprocal

Divides the number(s) in the left-hand argument by the number(s) in the right- hand argument.

```
      9÷3                       (One number is divided by another)
3
      0÷0                       (This is a special case)
1
      21 15 75 13÷5             (Division of each number in a vector by
4.2 3 15 2.6                     a single number)
      12 8 24÷2 8 6             (Each number in one vector is divided by the
6 1 4                            corresponding number in another vector)
      TABLE ← 2 5 ρ ι 10        (Numbers 1 to 10 assigned to TABLE.
      TABLE÷10                  (Each number in TABLE is divided by 10
0.1  0.2  0.3  0.4  0.5
0.6  0.7  0.8  0.9  1
      TABLE÷TABLE               (Each number in one matrix is divided by the
1 1 1 1 1                        corresponding number in another of the same
1 1 1 1 1                        size and dimensions)
      1 ÷5 (ι5)                 (1 divided by each element on the right)
 0.2  1 0.5 0.3333333333 0.25 0.2
      2  3÷5 (ι4) (ι5)          (Arguments must be the same length)
LENGTH ERROR
      2 3÷5(ι4)(ι5)
      ^
      (ι4) 10 (ι3)÷ 5 (2 2 ρι4) (ι3)
 0.2 0.4 0.6 0.8      10                 5                1 1 1
                       3.333333333       2.5
```

```
                                           (Corresponding elements divided)
```

If the right argument contains a zero, APLX will generate a `DOMAIN ERROR`.

⌈ Ceiling

One-argument form *See also two-argument form* Greater of

The number or numbers in the right-hand argument are rounded up to the next whole number.

```
      ⌈ 45.9
46
      ⌈ ¯3.8                      (Note effect of rounding up on a
¯3                                 negative number)
      ⌈1.2 ¯0.3 99.1 2.8          (Each number in a vector is rounded up)
2 0 100 3
      ⌈¯0.5+1.2 ¯0.3 9.1 2.8      (0.5 is subtracted from each number
1 0 9 3                            before ⌈ is applied, producing
                                   'true' rounding)
     TABLE
62.8  3.0  ¯2.9
 9.1  7.3   0.01
      ⌈ TABLE                     (Each number in TABLE is rounded up)
63 3 ¯2
10 8  1
      ⌈(1.7 11.99 ¯2.3) (2 2⍴¯1.1 17.3 ¯0.1 103.4)
 2 12 ¯2     ¯1  18               (Each element is rounded up)
              0 104
```

Comparison tolerance

When acting on a number which is very close to but slightly bigger than an integer, Ceiling may round down to that integer rather than round up. This will happen if the argument is within comparison tolerance of the integer, and is therefore considered in APL to be equal to it.

Effect on internal representation

See the description of `⌊` Floor for information on the internal representation of the result of Ceiling.

⌈ Greater of

Two-argument form *See also one-argument form* Ceiling

Finds the larger of two numbers. Each number in the right-hand argument is compared with the corresponding number in the left-hand argument. The result is the larger number from each comparison. (This operation is affected by `⎕CT`, the comparison tolerance)

```
      87 ⌈ 91
91
     ¯5 ⌈ ¯9                    (The negative number nearer 0
¯5                               is considered the greater)
      20 7 40 ⌈ 91 3 41         (Each number in a vector is compared
91 7 41                          with the corresponding number in a
                                 vector of equal size)
      ⌈/TABLE                   (The / operator used with ⌈
62.8 9.1                         to select the biggest in each row.
                                 See the entry for /.)
      2 ⌈1 (2 2 ρι4) (ι3)       (The result of comparing 2 with each
 2    2 2   2 2 3                element in the right argument)
      3 4
      2  3⌈2 (2 2 ρι4) (ι3)     (Arguments must be the same length)
LENGTH ERROR
      2 3⌈2(2 2ρι4)(ι3)
      ^
      (2 2ρι4) 3 (ι3)⌈3 (2 2 ρι4) (3 2 1)
  3 3    3 3   3 2 3            (Corresponding elements compared)
  3 4    3 4
```

⌊ Floor

One-argument form *See also two-argument form* Lesser of

The number or numbers in the right-hand argument are rounded down to the next whole number.

```
      ⌊ 45.9
45
      ⌊ ¯2.3                    (Note the effect on a negative number)
¯3
      ⌊ 1.2 ¯0.3 99.1 2.8       (Each number in a vector is rounded
1 ¯1 99 2                        down)
      ⌊0.5+1.2 ¯0.3 99.1 2.8    (0.5 is added to each number before
1 0 99 3                         ⌊ is applied to it, ensuring 'true'
                                 rounding)
      TABLE
62.8  3.0  ¯2.9
 9.1  7.3   0.01
      ⌊ TABLE                   (Each number in TABLE is rounded down)
62  3 ¯3
```

```
        9  7  0
             ⌊(¯0.1 ¯10.1 11.3 7.4) ( 2 2⍴¯0.3 2.8 99.1 ¯2.3)
        ¯1 ¯11 11 7     ¯1  2          (Each element is rounded down)
                        99 ¯3
```

Comparison tolerance

When acting on a number which is very close to but slightly smaller than an integer, Floor may round up to that integer rather than round down. This will happen if the argument is within comparison tolerance of the integer, and is therefore considered in APL to be equal to it.

Effect on internal representation

If the argument to Floor or Ceiling is an array which is held internally in boolean or integer form, then the result will always be represented in integer form and the numbers in the array will be unchanged.

If the argument to Floor or Ceiling is internally in floating-point form, then in general, provided all the numbers within the argument are in the range of numbers which can be represented as integers, the result will internally be represented as integers rather than floating points. Floor or Ceiling can therefore be used to force the internal representation of numbers to integer form:

```
      X←3.0 100.0 ¯20.0
      ⎕DR X
3
      Y←⌊X
      ⎕DR Y
2
      X
3 100 ¯20
      Y
3 100 ¯20
      X=Y
1 1 1
```

In this example, X is held internally in floating-point format, but Y is held internally in integer format. The values of the array elements are, however, equal.

See ⎕DR for more information on data representation.

Differences between 32-bit and 64-bit implementations of APLX

In the 32-bit version of APLX, numbers can be represented as integers if they are in the range ¯2147483648 to 2147483647. If the argument to Floor or Ceiling contains numbers which round to numbers outside this range, the result will internally be represented in floating-point format.

In the 64-bit APLX64 interpreter, numbers can be represented as integers if they are in the range ¯9223372036854775808 to 9223372036854775807. However, the floating-point representation of a number is limited to 53 bits of precision, which is equivalent to saying that at 2*53 and above, several integers all map to the same floating-point representation. For this reason, if the argument to Floor or Ceiling is in floating-point form, and contains numbers whose magnitude is equal to or greater than 2*53, the result will be left in floating-point form so as not to introduce a spurious precision to numbers which are inherently imprecise.

In this example using APLX64, `X` is represented internally as a 64-bit integer, and `Y` is represented internally as a floating-point number:

```
      X←2*53
      X
9007199254740992
      ⎕DR X
2
      Y←X×1.0
      Y
9.007199255E15
      ⎕DR Y
3
      ⌊Y
9.007199255E15
      ⎕DR ⌊Y
3
      ⌊Y-1
9007199254740991
      ⎕DR ⌊Y-1
2
```

⌊ Lesser of

Two-argument form *See also one-argument form* Floor

Finds the smaller of two numbers. Each number in the right-hand argument is compared with the corresponding number in the left-hand argument. The result is the smaller number from each comparison. (This operation is affected by `⎕CT`, the comparison tolerance)

```
      87 ⌊ 91
87
      ¯5 ⌊ ¯9                 (The negative number further from 0
¯9                             is considered the smaller)
      20 7 40 ⌊ 91 3 41       (Each number in a vector is compared
20 3 40                        with the corresponding number in a
                               vector of equal size)
      TABLE1
0  ¯3  66
9  16   4
      TABLE2
12  ¯8  17
 7   0   1
      TABLE1 ⌊ TABLE2         (Each  number  in  a  matrix  is  compared
0  ¯8  17                      with the corresponding number in
7   0   1                      a matrix with the same number of rows
                               and columns)
      2 ⌊1 (2 2 ρι4) (ι3)     (Each element in the right argument is
 1     1 2   1 2 2             compared with 2)
       2 2
```

| Absolute value

One-argument form *See also two-argument form* Residue

Makes any negative numbers in the right-hand argument positive.

```
      | 2 ¯4 ¯7.8 3
2 4 7.8 3
      |(¯0.1 ¯10.1 11.3 25) (2 2ρ¯10 3 ¯45 2.1)
 0.1 10.1 11.3 25  10 3
                   45 2.1
```

| Residue

Two-argument form *See also one-argument form* Absolute value

For positive arguments, gives the remainder resulting from dividing the right- hand argument by the left-hand argument. When the arguments are of the opposite sign, the result is the complement of the result that you would get if they had the same sign. So for non-zero results, you must subtract the remainder from the divisor. (This operation is affected by `⎕CT`, the comparison tolerance)

```
      3 | 10                  (The remainder of 10÷3)
1
      7 | 24 5 0 25           (The remainder of dividing each
   3 5 0 4                     number in a vector by 7)
     3 17 2 | 5 20 3          (The remainder after dividing each
2 3 1                          number in a vector by the corresponding
                               number in another similar vector)
     ¯7 | ι 10
¯6 ¯5 ¯4 ¯3 ¯2 ¯1 0 ¯6 ¯5 ¯4
     TABLE ← 2 3 ρ ι 6        (The remainder of dividing each
    4 | TABLE                  number in a matrix by 4)
1 2 3
0 1 2
     TABLE | TABLE            (The remainder after dividing each
0 0 0                          number in a matrix by the
0 0 0                          corresponding number in another)
      2 |1 (2 2 ρι4) (ι3)     (Divide every element by 2)
 1     1 0   1 0 1
       1 0
      2  3|2 (2 2 ρι4) (ι3)   (Arguments must be of equal length)
LENGTH ERROR
      2 3|2(2 2ρι4)(ι3)
      ^
      (2 2ρι4) 3 (ι3)|3 (2 2 ρι4) (3 2 1)
  0 1    1 2   0 0 1          (Corresponding elements divided)
  0 3    0 1
```

⍳ Index generator

One-argument form *See also two-argument form* Index of

`⍳` generates a series of integers which start at the index origin `(⎕IO)` and whose length is specified by the right argument which must be 0 or a positive integer scalar. The examples below assume the default index origin of 1 (see `⎕IO` for more details). The argument to `⍳` must be a simple numeric scalar or one-element vector.

```
            ⎕IO                     (Default setting of ⎕IO)
      1
```

To generate the series from 1 to 10:

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

To generate the series 1 to 5 to be used in selecting the first five elements from a vector (see separate entry for `[ ]`):

```
            PRICE←29 4 61 5 88 2 18 90 3 201 12 53 27 80
            PRICE[⍳ 5]
      29 4 61 5 88
```

To generate a vector of five elements containing the series 1, 1 2, and so on, use the `¨` ('each') operator.

```
            ⍳¨⍳5
       1  1 2  1 2 3  1 2 3 4  1 2 3 4 5
```

A common mechanism to generate an empty vector is:

```
            ⍳0
                                    (No display)
```

⍳ Index of

Two-argument form *See also one-argument form* Index generator

`⍳` finds whether the items in the right argument occur in the left argument (which must be a vector) and if so in what positions. For each element in the right argument a number is returned showing its position in the left argument. If an element is not found in the left argument (or if the arguments are of different types), a number one greater than the position of the last element in the left argument is returned. The shape of the result is the same as that of the left argument.

The result of dyadic `⍳` is influenced by `⎕IO`, which will control whether the index positions start at 0 or 1 - for more details see the entry for `⎕IO`. The comparisons done by this operation are affected by `⎕CT`, the comparison tolerance value.

```
      2 5 9 14 20 ⍳ 9
3                              (9 is in position 3)

      2 5 9 14 20 ⍳ 12         (12 isn't in the left argument, so a
6                               number 1 greater than the number of
                                elements on the left results)
      'GORSUCH' ⍳ 'S'
4                              (S occurs in position 4)

      'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ⍳ 'CARP'
3 1 18 16                      (The characters 'CARP' are in positions
                                3 1 18 and 16)

       'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ⍳ 'PORK PIE'
16 15 18 11 27 16 9 5          (The 27 in the result indicates
                                characters not found in the 26-character
                                left argument. In this case the 'space'
                                character.)
      DAYS←'MON' 'TUES' 'WED'
      ⍴DAYS
3                              (DAYS is a 3 element vector)
      DAYS ⍳ 'MON' 'THURS'     ('MON' found in first position, 'THURS'
1 4                             is not found)
```

See `≡` (Match) for a discussion of the criteria which determine whether two elements are considered the same.

? Roll

One-argument form *See also two-argument form* Deal

Generates numbers chosen at random from the series of the first N integers which start at the index origin (`⎕IO`),where N is specified by the right argument. In the examples below `⎕IO` is set to 1, the default.

```
      ? 100                   (Generates  a  random   number   between  1
 14                            and 100)
      ? 10 100 1000           (Generates 3 random numbers, the first
 10 39 520                     between 1 and 10, the second between
                               1 and 100, the third between 1 and 1000)
      DATA ← ?100 ⍴ 100       (Generates 100 random numbers in the range
                               1 to 100 - not necessarily unique)
      ?(3⍴5) (2 3⍴10)
 1 4 3     6  3  1
           7  7 10
```

Note: The system variable `⎕RL` (random link) contains a value used to generate random numbers. To generate the same number(s) on two occasions, set `⎕RL` to the same value before each use of `?`.

? Deal

Two-argument form *See also one-argument form* Roll

Generates as many random numbers as are specified in the left-hand argument from the first N numbers starting at `⎕IO`, where N is specified in the right-hand argument. Each number generated is unique; that is to say there are no repetitions. The left and right arguments must be simple numeric scalars or one-element vectors.

```
      10 ? 100                  (A request for 10 unique random numbers
46 54 22 5 68 94 39 52 84 4      in the range 1 to 100, assuming ⎕IO is 1 )
      LIST ← 3 ? 10             (3 random numbers between 1 and 10
                                 are put in LIST)
      BINGO←4 4ρ16?100
      BINGO                     (16 random numbers between 1 and 100
41 12 46 71                      are put into a 4-by-4 matrix called
 6 54 68  4                      BINGO)
63 94 87 58
21 70 50 75
      4 ? 3                     (A request for 4 unique integers in
DOMAIN ERROR                     the range 1 to 3 causes an error)
```

Note: The system variable `⎕RL` (random link) contains a value used to generate random numbers. To generate the same number(s) on two occasions, set `⎕RL` to the same value before each use of `?`.

```
      ⎕RL ← 12345
      5 ? 10000
97 834 948 36 12
      ⎕RL ← 12345
      5 ? 10000
97 834 948 36 12
```

* Exponential

One-argument form *See also two-argument form* Power

Returns the mathematical constant e (approximately 2.718) raised to the power of the right argument.

```
      * 1
2.718281828                     (e to the power of 1 is e itself)
      * 2
7.389056099                     (e squared)
      *ι3                       (e to the power 1 2 3)
2.718281828 7.389056099 20.08553692
      *(ι2) (2 2ρι4)
 2.718281828 7.389056099        2.718281828      7.389056099
                                20.08553692      54.59815003
```

* To the power of

Two-argument form *See also one-argument form* 'e' to power

Raises the left-hand argument to the power of the right-hand argument.

```
      2 * 3                     (2 to the power 3, or 2 cubed)
8
      ¯1 * 2 3 4
1 ¯1 1                          (¯1 to the power 2 3 4)
     2 * 0.5
1.414213562                     (Square root of 2)
     2 4 6 8 16 * 2
4 16 36 64 256                  (Square of 2 4 6 8 16)
      ¯1*0.5                    (No unreal number result allowed)
DOMAIN ERROR
      ¯1*0.5
        ^
```

If the right argument is negative, the result is the reciprocal of the result obtained from using a right argument which is the absolute value of the negative argument.

```
      2*¯3
0.125
      ÷2*3                      (Reciprocal of 2*3)
0.125

      ((⍳3)(2 2⍴⍳4))*2
 1 4 9  1  4
        9 16
```

⍟ Natural log

One-argument form *See also two-argument form* Log to the base

Finds the natural logarithm, that is the log to the base e, of the number or numbers in the right-hand argument (e is approximately 2.7182). The numbers must be positive.

```
      ⍟ 10                      (Finds the log to base e of 10)
2.302585093
      ⍟ 3 9 18                  (Finds the log to base e of 3 9 and 18)
1.098612289 2.197224577 2.890371758
      ⍟ 3.3                     (Finds the log to base e of 3.3)
1.193922468
      ⍟(2 2⍴⍳4) (⍳3)
     0                  0.6931471806     0 0.6931471806 1.098612289
     1.098612289        1.386294361
```

⍟ Log to the base

Two-argument form *See also one-argument form* Natural Logarithm

Computes the log of a number or numbers to an arbitrary base. The left-hand argument is the base and the right-hand argument is the number whose log is to be found.

```
      3 ⍟ 9                     (The log of 9 to the base 3)
2
      2 3 4 5 6⍟4 9 16 25 36    (The log of each number on the right to
2 2 2 2 2                        the corresponding base on the left)
      2 ⍟ 2 4 8 16 32 64
1 2 3 4 5 6
      2 ⍟ 13.9
3.797012978
      2 3⍟(2 2⍴2 4 8 16) (3 9 27)
1 2  1 2 3
3 4                              (Corresponding elements of left and right
                                 arguments used as successive arguments to
                                 the ⍟ function)
```

○ Pi times

One-argument form *See also two-argument form* Circular & Hyperbolic functions

The value of pi (approx. 3.141592654) is multiplied by the right-hand argument.

```
      ○ 1                       (pi times 1 is pi)
3.141592654
      ○÷4                       (pi divided by 4, or 45 degrees, in
0.7853981634                      radians)
      ○ 1 2 3                   (pi times each number in the vector)
3.141592654 6.283185307 9.424777961
      10 30 45×○÷180            (converts 10 20 45 degrees to radians)
0.1745329252 0.5235987756 0.7853981634
```

○ Circular and Hyperbolic functions

Two-argument form *See also one-argument form* pi times

This form of ○ provides you with a group of related functions. The left argument identifies which of these functions you wish to use, the right argument is the data the function works on. (Data to trigonometric functions is expressed in radians.)

```
     Left argument 0 or positive          Left argument negative

     0  square root of 1-X*2
     1  sin X                             ¯1  arcsin X
     2  cos X                             ¯2  arccos X
     3  tan X                             ¯3  arctan X
     4  square root of (X*2)+1            ¯4  square root of (X*2)-1
     5  sinh X                            ¯5  arcsinh X
     6  cosh X                            ¯6  arcosh X
     7  tanh X                            ¯7  arctanh X
```

For example:

```
              1 ○ ○÷4                   (45 degrees is ○÷4 radians and
        0.7071067812                    Sin of 45 degrees is 1÷root 2)
```

The functions `0○`, `4○`, `¯4○` are known as the 'Pythagorean functions'. For example, given a right-angled triangle with hypotenuse of length 1, the length of one of the other two sides is 0 ○ times the length of the third side. Conversely, if one of the sides in the triangle adjacent to the right angle is of length 1, the length of the hypotenuse is given by `4○` times the length of the third side and the length of the third side is `¯4` ○ times the length of the hypotenuse.

Numeric Accuracy

Calculations of trigonometric functions are subject to accuracy limitations, especially near mathematical singularities. In addition, for very large arguments, the circular functions become meaningless because of limitations in the resolution of floating-point numbers, since the 'correct' answer depends on bits which have been lost from the representation. For these reasons, APLX gives a DOMAIN ERROR if you ask for the sine, cosine or tangent of a number greater than `2*51`.

! Factorial

One-argument form *See also two-argument form* Binomial

When applied to a positive whole number, `!` gives the product of the whole numbers from 1 to that number, inclusive.

```
      ! 3                     (Equivalent of 1×2×3)
6
```

If the argument is non-integer and positive, `!` gives the mathematical 'gamma function' of the argument + 1.

```
      ! 2.5
3.32335097
```

! Binomial

Two-argument form *See also one-argument form* Factorial or Gamma function

In its two-argument form, with positive arguments, `!` tells you how many different ways there are of selecting the number of items specified on the left from the population of items specified on the right. The order of items in each pair is ignored. So if the population of four consisted of the letters A B C D, the 6 possible combinations of 2 letters would be: AB AC AD BC BD CD. The combination BA would be regarded as the same as AB.

```
      2 ! 4                   (Number of unique pairs from a population
6                              of 4)
      3 ! 20                  (Number of groups of three from a
1140                           population of 20)
      2 ! 6 12 20             (Number of pairs from a population
15 66 190                      of 6 12 20 respectively)
      TABLE1 ← 2 3ρι6
      TABLE2 ← 2 3 ρ 3 6 9 12 15 18
      TABLE1 ! TABLE2         (TABLE1 is table of group sizes, TABLE2
  3              15                84        is table of populations)
495            3003             18564
```

Other cases, such as negative or non-integer arguments, are also catered for. The various results that can be obtained are:

Left Argument	Right Argument	Right-Left	Result
+ve	+ve	+ve	(!RIGHT)÷(!LEFT)×!RIGHT-LEFT
+ve	+ve	-ve	0
+ve	-ve	-ve	(¯1*LEFT)×LEFT!LEFT-RIGHT+1

```
      -ve         +ve         +ve           0
      -ve         -ve         +ve           (¯1*RIGHT-LEFT)×(|RIGHT+1)!(|LEFT+1)
      -ve         -ve         -ve           0
```

⌹ Matrix inverse

One-argument form *See also two-argument form* Matrix division

Produces the inverse of the matrix in the right-hand argument. The right argument must be a simple numeric array. The inverse of a matrix is itself a matrix. It is constructed so that, if matrix-multiplied by the original matrix, it gives the identity matrix, that is the matrix analogue of unity. In matrix algebra, an inverse is usually found only for a square matrix. APL further defines a matrix inverse for a matrix with more rows than columns. In this case the shape of the inverse is the reverse of the shape of the matrix being inverted, and the expression:

```
            (⌹Y)+.×Y
```

still gives the identity matrix. The result of the inverse is the left inverse.

```
            TABLE
      7  9  8
      3  4  5
      6  2  1
            6  2 ⍕ ⌹ TABLE
      ¯.11  .12  .23
       .47 ¯.72 ¯.19
      ¯.32  .70  .02
```

Matrix multiplication is carried out by the inner product operation `+.×` (see Inner product).

```
            (⌹2 2⍴5 1 0 1)+.×2 2⍴5 1 0 1
      1 0                        (A matrix multiplied by its inverse gives
      0 1                         the unit matrix)
```

If the right argument to `⌹` is a scalar, the result is the reciprocal of the argument.

```
            ⌹2
      0.5
```

If the matrix is singular (i.e. does not have an inverse), a DOMAIN ERROR will be reported. Note that matrix inversion is subject to accuracy limitations imposed by the representation of floating-point numbers and the algorithm used to calculate the result. In particular, matrices which are nearly singular may give results of limited accuracy, and small changes to the input can produce very big changes to the output.

⌹ Matrix divide

Two-argument form *See also one-argument form* Matrix inversion

The right and left-hand arguments are conformable simple numeric matrices (arrays of rank 2). Vectors are treated as one column matrices and scalars are treated as matrices of shape 1 1. The result is a matrix which, if matrix- multiplied by the right-hand argument, would yield the left-hand argument.

```
      X
1  2
3  6
9 10
      Y
1 0 0
1 1 0
1 1 1
      X ⌹ Y
1          2
2          4
6          4
```

This last operation is the same as

```
      ( ⌹ Y ) +.× X
```

which is another way of defining the operation.

An important use for matrix divide is to give the least squares solution to the set of simultaneous linear equations:

```
      B = A +.× X            for a matrix A and vector B, or columns of
                              matrix B
```

The solution is:

```
      B ⌹ A
```

If the matrix division does not have a solution, DOMAIN ERROR will be reported. Note that matrix division is subject to accuracy limitations imposed by the representation of floating-point numbers and the algorithm used to calculate the result.

< Less than

Compares each element in the left-hand argument with the corresponding element in the right-hand argument. If an element in the left-hand argument is less than the corresponding right-hand element, the result of that comparison is 1. Otherwise it is 0. (This operation is affected by `⎕CT`, the comparison tolerance)

```
      12 < 1
0
      2 < 12
1
      12 < 12
0
      11 7 2 5 < 11 3 2 6     (Compares each element in a vector with
0 0 0 1                        the corresponding element in a vector of
                               equal length)

      ¯3 < ¯4                 (Compares negative numbers. The number
0                              nearer 0 is considered the greater.)

      8 < 2+2+2+2             (The right argument is evaluated
0                              before the comparison is made)

      TABLE ← 2 3 ⍴ 1 2 3 4 5 6
      MABLE ← 2 3 ⍴ 3 3 3 5 5 5
      TABLE < MABLE           (Compares each element in a matrix
1 1 0                          with the corresponding element in
1 0 0                          a matrix of equal size and dimensions)
      3<TABLE
0 0 0
1 1 1
      3 < TABLE MABLE         (Compares 3 with the elements of the
 0 0 0  0 0 0                  nested vector)
 1 1 1  1 1 1
```

≤ Less than or equal

Compares each element in the left-hand argument with the corresponding element in the right-hand argument. If an element in the left-hand argument is less than, or equal to, the corresponding right-hand element, the result of that comparison is 1. Otherwise it is 0. (This operation is affected by `⎕CT`, the comparison tolerance)

```
      12 ≤ 1
0
      2 ≤ 12
1
      12 ≤ 12
1
      11 7 2 5≤11 3 2 6       (Compares each element in a vector with
1 0 1 1                        the corresponding element in a vector of
```

```
                                    equal length)
      ¯3 ≤ ¯4                      (Compares negative numbers. The number
0                                   nearer 0 is considered the greater.)
      8 ≤ 2+2+2+2                  (The right argument is evaluated
1                                   before the comparison is made)

      TABLE ← 2 3 ρ 1 2 3 4 5 6
      MABLE ← 2 3 ρ 3 3 3 5 5 5
      TABLE ≤ MABLE                (Compares each element in a matrix
1 1 1                               with the corresponding element in
1 1 0                               a matrix of equal size and dimensions)
      3≤TABLE
0 0 1
1 1 1
      3 ≤ TABLE MABLE              (Compares 3 with the elements of the
 0 0 1  1 1 1                       nested vector)
 1 1 1  1 1 1
```

= Equal

Compares each element in the right-hand argument with the corresponding element in the left-hand argument and returns 1 if they are equal, 0 if they are not. (This operation is affected by `⎕CT`, the comparison tolerance)

This function works on both numeric and character data. A numeric element is never considered equal to a character element.

```
      12 = 12
1
      2 = 12
0
      'Q' = 'Q'                    (Compares character data)
1
      1 = '1'                      (Comparisons between numeric and character
0                                   data  are allowed, but always give 0)
      11 7 2 9 = 11 3 2 6          (Compares each element in a vector with
1 0 1 0                             the corresponding element in a vector of
                                    equal length)
      'STOAT' = 'TOAST'
0 0 0 0 1
      8 = 2+2+2+2                  (The right argument is evaluated
1                                   before the comparison is made)
      TABLE←2 3ρ1 2 3 4 5 6
      MABLE←2 3ρ3 3 3 5 5 5
      TABLE = MABLE                (Compares each element in a matrix
0 0 1                               with the corresponding element in
0 1 0                               a matrix of equal size and dimensions)
      3=TABLE
0 0 1
0 0 0
      3 = TABLE MABLE              (Compares 3 with the elements of the
 0 0 1  1 1 1                       nested vector)
 0 0 0  0 0 0
```

See also the `≡` (match) function which tests for depth, rank and shape equality as well.

If the arguments contain object (or class) references, the elements are considered equal if the reference indices are the same, i.e. if they refer to the same entry in APL's internal table of objects. For internal objects, this will be true if and only if the elements refer to the same object. Note that different objects which happen to contain the same properties are not considered equal. For example, if Point is a simple class with properties X and Y:

```
      PT←⎕NEW Point
      PT.X←63 ⋄ PT.Y←42
      A←PT
      B←PT.⎕CLONE 1
      A.⎕DS
X=63, Y=42
      B.⎕DS
X=63, Y=42
      A=PT      ⍝ References to the same object
1
      B=PT      ⍝ Objects are different, but have the same property values
0
```

For external objects, there might be two references which APL does not know refer to the same object. Therefore the use of the APL Equals primitive on external objects is not recommended.

≥ Greater than or equal

Compares each element in the left-hand argument with the corresponding element in the right-hand argument. If an element in the left-hand argument is greater than, or equal to, the corresponding right-hand element, the result of that comparison is 1. Otherwise it is 0. (This operation is affected by `⎕CT`, the comparison tolerance)

```
      12 ≥ 1
1
      2 ≥ 12
0
      12 ≥ 12
1
      11 7 2 5 ≥ 11 3 2 6     (Compares each element in a vector with
1 1 1 0                        the corresponding element in a vector of
                               equal length)
      ¯3≥¯4                   (Compares negative numbers. The number
1                              nearer 0 is considered the greater.)
      8 ≥ 2+2+2+2             (The right argument is evaluated
1                              before the comparison is made)
      TABLE ← 2 3 ρ 1 2 3 4 5 6
      MABLE ← 2 3 ρ 3 3 3 5 5 5
      TABLE ≥ MABLE           (Compares each element in a matrix
0 0 1                          with the corresponding element in
0 1 1                          a matrix of equal size and dimensions)
      3≥TABLE
1 1 1
0 0 0
      3 ≥ TABLE MABLE         (Compares 3 with the elements of the
 1 1 1  1 1 1                  nested vector)
 0 0 0  0 0 0
```

> Greater than

Compares each element in the left-hand argument with the corresponding element in the right-hand argument. If an element in the left-hand argument is greater than the corresponding right-hand element, the result of that comparison is 1. Otherwise it is 0. (This operation is affected by `⎕CT`, the comparison tolerance)

```
      12 > 1
1
      2 > 12
0
      12 > 12
0
      11 7 2 5 > 11 3 2 6     (Compares each element in a vector with
0 1 0 0                        the corresponding element in a vector of
                               equal length)
      ¯3 > ¯4                 (Compares negative numbers. The number
1                              nearer 0 is considered the greater.)
      8 > 2+2+2+2             (The right argument is evaluated
0                              before the comparison is made)
      TABLE ← 2 3 ρ 1 2 3 4 5 6
      MABLE ← 2 3 ρ 3 3 3 5 5 5
      TABLE > MABLE           (Compares each element in a matrix
0 0 0                          with the corresponding element in
0 0 1                          a matrix of equal size and dimensions)
      3>TABLE
1 1 0
0 0 0
      3 > TABLE MABLE         (Compares 3 with the elements of the
 1 1 0  0 0 0                  nested vector)
 0 0 0  0 0 0
```

≠ Not equal

Compares each element in the right-hand argument with the corresponding element in the left-hand argument and returns 1 if they are not equal and 0 if they are. (This operation is affected by `⎕CT`, the comparison tolerance function)

```
      12 ≠ 12
0
      2 ≠ 12
1
      'Q' ≠ 'Q'               (Compares character data)
0
      11 7 2 9≠11 3 2 6       (Compares each element in a vector with
0 1 0 1                        the corresponding element in a vector of
                               equal length)
      'STOAT' ≠ 'TOAST'
1 1 1 1 0
```

```
      8 ≠ 2+2+2+2            (The right argument is evaluated
0                             before the comparison is made)
      TABLE ← 2 3 ρ 1 2 3 4 5 6
      MABLE ← 2 3 ρ 3 3 3 5 5 5
      TABLE ≠ MABLE          (Compares each element in a matrix
1 1 0                         with the corresponding element in
1 0 1                         a matrix of equal size and dimensions)
      3≠TABLE
1 1 0
1 1 1
      3 ≠ TABLE MABLE        (Compares 3 with the elements of the
 1 1 0  0 0 0                 nested vector)
 1 1 1  1 1 1
```

≡ Depth

One-argument form *See also two-argument form* Match

Depth is used to indicate the level of nesting. For a simple scalar, depth is 0. For other arrays, the depth of the array is `1+` the depth of the item of maximum depth in the array.

```
      ≡4                     (Depth of a scalar is 0)
0
      ≡ι4                    (Depth of a vector is 1)
1
      ≡2 2ρι6                (Depth of a matrix is 1)
1
      ≡'ABC' 1 2 3 (23 55)   (Maximum depth is 1+ depth of a vector)
2
      'ABC' (2 4ρ('ABC' 2 3 'K'))
ABC  ABC 2 3 K               (Maximum depth object within the array is
     ABC 2 3 K                2 - a matrix)
      ≡'ABC' (2 4ρ('ABC' 2 3 'K'))
3                            (Overall depth is thus 3)
```

See also `ρ` (shape) to enquire about the shape of an array.

≡ Match

Two-argument form *See also one-argument form* Depth

The match function will test whether its arguments are the same in every respect - depth, rank, shape and corresponding elements. The result is always a scalar 1 or 0.

```
      3≡3                    (Two scalars are identical)
1
      3≡,3                   (Scalar does not match a vector)
0
```

```
            4 7.1 8 ≡ 4 7.2 8         (Shape is the same but values are not)
      0
            (3 4⍴⍳12)≡3 4⍴⍳12         (Two matrices are identical)
      1
            (3 4 ⍴⍳12)≡⊂3 4⍴⍳12       (Simple matrix does not match an enclosed
      0                                version of itself)
            VEC←'ABC' 'DEF'           (Two element vector of 'ABC' 'DEF)
            VEC
      ABC DEF
            ⍴VEC                      (Length 2)
      2
            VEC≡'ABCDEF'              (Does not match the 6 element vector
      0                                'ABCDEF')
```

Empty arrays are considered the same only if they have the same type, rank, shape and prototype.

```
            (⍳0)≡''                   (Types are different)
      0
            (2 0⍴0)≡0 2⍴0             (Shapes are different)
      0
            (0⍴⊂1 2 3)≡0⍴⊂2 2⍴⍳4      (Prototypes are different)
      0
```

The comparisons done by this operation are affected by `⎕CT`, the comparison tolerance value.

If the arguments contain object (or class) references, the elements are considered equal if the reference indices are the same, i.e. if they refer to the same entry in APL's internal table of objects. For internal objects, this will be true if and only if the elements refer to the same object. Note that different objects which happen to contain the same properties are not considered equal.

≢ Not Match

The Not Match function ≢ will test whether its arguments are different in any respect - depth, rank, shape or corresponding elements. The result is always a scalar 1 or 0. It is equivalent to `~L ≡ R`

```
            3≢3                       (Two scalars are identical)
      0
            3≢,3                      (Scalar does not match a vector)
      1
            4 7.1 8 ≢ 4 7.2 8         (Shape is the same but values are not)
      1
            (3 4⍴⍳12)≢3 4⍴⍳12         (Two matrices are identical)
      0
            (3 4 ⍴⍳12)≢⊂3 4⍴⍳12       (Simple matrix does not match an enclosed
      1                                version of itself)
            VEC←'ABC' 'DEF'           (Two element vector of 'ABC' 'DEF)
            VEC
      ABC DEF
            ⍴VEC                      (Length 2)
      2
            VEC≢'ABCDEF'              (Does not match the 6 element vector
      1                                'ABCDEF')
```

The comparisons done by this operation are affected by `⎕CT`, the comparison tolerance value.

See `≡` Match for more information on how the comparisons are done.

∊ Enlist

One-argument form *See also two-argument form* Membership

Enlist produces a vector containing every element from every item in its argument. None of the properties of an array are preserved - rank, shape or depth. The result is always a simple vector. Note that empty vectors in the argument do not appear in the result.

```
      A←(1 2 3) 'ABC' (4 5 6)
      ⍴A                       (Length 3 vector containing 3 length 3
3                               vectors)
      ⍴∊A                      (Enlist produces one 9 element vector)
9
      B←2 2⍴(2 2⍴⍳4) 'DEF' (2 3⍴⍳6) (7 8 9)
      B
1 2     DEF
3 4

1 2 3   7 8 9
4 5 6
      ⍴B
2 2
      ∊B                       (Enlist produces a 16 element vector,
1 2 3 4 DEF 1 2 3 4 5 6 7 8 9   processing the rows first - as ravel)
      ⍴∊B
16
```

For simple arguments, enlist is the equivalent of the ravel (,) function.

Enlist can be used for selective specification.

```
      (∊A)←⍳9                  (A as above)
      A
1 2 3  4 5 6  7 8 9
      ⍴A                       (Shape preserved)
3
```

∊ Membership

Two-argument form *See also one-argument form* Enlist

Checks on whether a data element exists in the right argument. It returns 1 for each element of the left argument found in the right argument and 0 for each element of the left argument not found in the right argument. (This operation is affected by `⎕CT`, the comparison tolerance)

The arguments compared need not have the same number of elements, nor need they have the same number of dimensions. The result has the same shape as the left argument. See `≡` (Match) for a discussion of the criteria which determine whether two data elements are considered the same.

```
      4 ∊ 4 4 5
1                             (1 means 4 is found in 4 4 5)
      'A' ∊ 'ABRACADABRA'
1                             (A is in ABRACADABRA)
      'ABRACADABRA' ∊ 'A'
1 0 0 1 0 1 0 1 0 0 1
```

(Elements 1 4 6 8 and 11 of the left-hand argument occur in the right-hand argument.)

```
      'A B C' ∊ 'ABCDE'
1 0 1 0 1                     (The 0s represent spaces in 'A B C'
                               which don't occur in 'ABCDE')
      12 24 36 ∊ 6 12 18 24 30 36
1 1 1                         (Vectors don't need to have the same
                               number of elements)
      TABLE ← 3 3 ⍴ 1 2 3 4 5 6 7 8 9
      3 6 9 ∊ TABLE
1 1 1
      TABLE ∊ 3 6 9           (Notice that the result always has
0 0 1                          the same shape as the left-hand
0 0 1                          argument)
0 0 1
      '4' ∊ ⍳10               (The character '4' does not appear in
0                              the numeric vector ⍳10)
      NAMES←'JOHN' 'MARY' 'HARRY'
      ⍴NAMES                  (3 element vector)
3
      'MARY'∊NAMES            (None of the 4 characters 'MARY' are
0 0 0 0                        elements in NAMES)
      (⊂'MARY')∊NAMES         (The scalar containing 'MARY' does exist in
1                              NAMES)
      NAMES∊⊂'MARY'           (Those  elements of names which contain the
0 1 0                          scalar 'MARY')
      'MARY' 'JIM' 'JOHN' ∊ NAMES
1 0 1                         ('JIM' not found in NAMES)
```

⍷ Find

Find searches for instances of the left argument within the right argument. A boolean result is returned with a 1 where the start of the left argument is found in the right argument. The shape of the result is the same as the shape of the right argument.

```
      'ME'⍷'HOME AGAIN'       (Find the pattern 'ME' in 'HOME AGAIN')
0 0 1 0 0 0 0 0 0 0
      WEEK
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
      'DAY' ⍷ WEEK            (Find the pattern 'DAY' in WEEK)
0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0
      WEEK ⍷ 'DAY'            (WEEK not found in 'DAY' - wrong rank)
0 0 0
```

The arguments can be of any rank, but ⍷ always searches for the whole of the left argument in the right argument.

```
      (1 2) (3 4) ⍷ 'START' (1 2 3) (1 2) (3 4)
0 0 1 0                       (Search within nested vector)
      MAT
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
      (2 2⍴7 8 12 13)⍷MAT     (Search pattern is a matrix)
0 0 0 0 0                     (1 shows top left corner)
0 1 0 0 0
0 0 0 0 0
0 0 0 0 0
```

See also the system function `⎕SS` for string search operations on vectors.

∪ Unique

One-argument form *See also two-argument form* Union

Unique is used to remove duplicated items from a vector. The result is a vector containing all the unique items in the argument, in the order in which they first appear. The argument must be a vector (or scalar).

When the argument is nested, an exact match in data and structure must be found before an item is removed as a duplicate. This operation is affected by ⎕CT, the comparison tolerance.

```
       ∪'THE QUALITY OF MERCY IS NOT STRAINED'
THE QUALIYOFMRCSND

      ∪1 4 17 23 12 4 2 7 99 33 ¯1 4 17 99 100 101
1 4 17 23 12 2 7 99 33 ¯1 100 101

      ∪'THIS' 'THAT' 'THE' 'OTHER' 'OTHER' 'THAN' 'THIS' 'AND' 'THAT'
 THIS THAT THE OTHER THAN AND
```

See also the other 'set' operations: ∪ Union, ∩ Intersection and ~ Without .

∪ Union

Two-argument form *See also one-argument form* Unique

Union returns all items which can be found in both the left and right arguments. The right argument can be of any shape or rank. The left argument must be a scalar or vector. The result is always a vector.

The result first contains all the items in the left argument (in the order in which they appear), followed by all the items found in the right argument but not in the left argument. If a particular item appears more than once in the left argument, it will also appear more than once in the result. Equally, if a particular item does not appear in the left argument, but does appear multiple times in the right argument, it will appear multiple times in the result.

This operation is affected by ⎕CT, the comparison tolerance.

```
       'THE QUALITY OF MERCY IS NOT STRAINED'∪'HIP HOP DOWN TO THE ZOO'
THE QUALITY OF MERCY IS NOT STRAINEDPPWZ

      1 4 17 23 12 2 7 99 33∪¯1 4 17 99 100 101
1 4 17 23 12 2 7 99 33 ¯1 100 101
```

```
      'THIS' 'THAT' 'THE' 'OTHER'∪'OTHER' 'THAN' 'THIS' 'AND' 'THAT'
 THIS THAT THE OTHER THAN AND

      ⎕display (23 43 21) (⍳5) (2 2⍴'BLOT') ∪ (⍳5) ('BLOT')
┌→────────────────────────────────────┐
│ ┌→───────┐ ┌→────────┐ ┌→─┐ ┌→───┐ │
│ │23 43 21│ │1 2 3 4 5│ ↓BL│ │BLOT│ │
│ └~───────┘ └~────────┘ │OT│ └────┘ │
│                        └──┘        │
└∊────────────────────────────────────┘
```

See also the other 'set' operations: ∪ Unique, ~ Without and ∩ Intersection.

∩ Intersection

Intersection returns a vector containing all those items in the left argument which can also be found in the right argument. The right argument can be of any shape or rank. The left argument must be a scalar or vector. The result is always a vector.

The items are returned in the order in which they appear in the left argument. If a particular item appears more than once in the left argument, it will also appear more than once in the result.

When the arguments are nested, an exact match in data and structure must be found for two items to be considered identical. This operation is affected by ⎕CT, the comparison tolerance.

```
       'THE QUALITY OF MERCY IS NOT STRAINED'∩'AEIOU'
EUAIOEIOAIE
       A←'THIS' 'AND' 'THAT'
       A∩'T'
                              ⍝ (No match for the single character T)
       A∩'AND'
                              ⍝ (No match for any of the three characters A N D)
       A∩⊂'AND'
 AND
      1 4 17 23 12 2 7 99 33∩2 2⍴⍳4
1 4 2
```

See also the other 'set' operations: ∪ Unique, ∪ Union and ~ Without.

~ Not

One-argument form *See also* Without

The right argument must consist only of the numbers 1 or 0. The effect of ~ is to change each 1 to 0 and each 0 to 1.

```
      ~1
0

      ~1 1 1 0                (Each 1 in a vector is changed to 0
0 0 0 1                        and each 0 to 1)

      TABLE
1 1 1
0 0 0
1 0 1
      ~TABLE                  (Each 1 in a matrix is changed to 0
0 0 0                          and each 0 to 1)
1 1 1
0 1 0
```

~ Without

Two-argument form *See also* Not

Without is used to remove items from a vector. Items in its left argument which are found in its right argument are removed from the result. When the arguments are nested, an exact match in data and structure must be found before an item is removed. The right argument can be of any shape of rank. This operation is affected by ⎕CT, the comparison tolerance.

```
      'ABCDEFGHIJKLMNOPQRSTUVWXYZ'~'AEIOU'
BCDFGHJKLMNPQRSTVWXYZ         (Vowels removed)
      1 2 3 4 5 6 ~2 4 6
1 3 5                         (Even numbers removed)
      'THIS   IS   TEXT'~' '
THISISTEXT                    (Removal of blanks - see also ⎕DBR)
      A←'THIS' 'AND' 'THAT'   (Three element nested vector)
      A~'T'
THIS AND THAT                 (No match for the single character T)
      A~'AND'
THIS AND THAT                 (No match for the length three vector)
      A~⊂'AND'
THIS THAT                     (Match found for nested scalar ⊂'AND')
      A~'TH' 'AND'
THIS THAT
```

See also the other 'set' operations: ∪ Unique, ∪ Union and ∩ Intersection.

∨ Or

Compares two arguments which must consist only of 0's and 1's. If either or both elements compared are 1's, the result for that comparison is 1. Otherwise the result for that comparison is 0.

```
      1 ∨ 1
1
      1 ∨ 0
1
      0 ∨ 0
0
      0 0 0 1 0 ∨ 1 1 1 1 0   (Each element in a vector is compared
1 1 1 1 0                      with the corresponding element in a
                               vector of equal size)

      TABLE ← 3 3⍴ 1 1 1 0 0 0 1 0 1
      0 ∨ TABLE               (Each element in a matrix is
1 1 1                          compared with 0)
0 0 0
1 0 1

      ∨\0 0 0 1 0 1 0         (The result is all 1's after the
0 0 0 1 1 1 1                  first 1)
```

∧ And

Compares two arguments which must consist only of 0's and 1's. If both elements compared are 1's, the result for that comparison is 1. Otherwise the result for that comparison is 0.

```
      1 ∧ 1
1
      1 ∧ 0
0
      0 ∧ 0
0
      0 0 0 1 1∧1 1 1 1 0     (Each element in a vector is compared
0 0 0 1 0                      with the corresponding element in a
                               vector of equal size)

      TABLE←3 3⍴1 1 1 0 0 0 1 0 1
      1∧TABLE                 (Each element in a matrix is compared
1 1 1                          with 1)
0 0 0
1 0 1
      ∧/TABLE                 (Applies ∧ to each row of the matrix.
1 0 0                          A 1 in the result shows that the
                               corresponding line contained only 1's)

      ∧\1 1 1 0 1 0 1 0 1     (The result is all 0's after the first 0)
1 1 1 0 0 0 0 0 0
```

⍱ Nor

Compares two arguments which must consist only of 0's and 1's. If neither element compared is a 1, the result for that comparison is 1. Otherwise the result for that comparison is 0.

```
      1 ⍱ 1
0
      1 ⍱ 0
0
      0 ⍱ 0
1
      0 0 0 1 0 ⍱ 1 1 1 1 0   (Each element in a vector is compared
0 0 0 0 1                      with the corresponding element in a
                               vector of equal size)

      TABLE←3 3⍴1 1 1 0 0 0 1 0 1
      0 ⍱ TABLE               (Each element in a matrix is compared
0 0 0                          with 0)
1 1 1
0 1 0
```

⍲ Nand

Compares two arguments which must consist only of 0's and 1's. If either or both elements compared are 0's, the result for that comparison is 1. Otherwise the result for that comparison is 0.

```
      1 ⍲ 1
0
      1 ⍲ 0
1
      0 ⍲ 0
1
      0 0 0 1 1 ⍲ 1 1 1 1 0   (Each element in a vector is compared
1 1 1 0 1                      with the corresponding element in a
                                vector of equal size)

      TABLE ← 3 3 ⍴ 1 1 1 0 0 0 1 0 1
      1 ⍲ TABLE               (Each element in a matrix is
0 0 0                          compared with 1)
1 1 1
0 1 0
```

ρ Shape of

One-argument form *See also two-argument form* Reshape

Enquires about the shape of each dimension of the data in the right-hand argument. See also ≡ (depth) to enquire about the depth of an array.

```
      ρ 95 100 82 74 2        (A vector has 1 dimension, length.
5                              This vector is 5 elements long.)

      ρ 'SEA SCOUT'           (Counting the space, this vector is
9                              9 elements long)
      ρ 'SEA' 'SCOUT'         (Two element nested vector)
2
      TABLE                   (A matrix has two dimensions, height,
1 3 5 7 9                      which is the number of rows, and
2 4 6 8 0                      width, which is the number of
8 6 4 2 1                      columns)
      ρ TABLE
3 5                           (This matrix has 3 rows and 5 columns)

      ρ 501                   (A single number or letter is like a
                               point. It has no dimensions so ρ
      ρρ501                    displays no answer, i.e. It returns a
0                              vector of size 0 - an empty vector. An empty
                               vector, being a vector, has a size of 0)
```

ρ Reshape

Two-argument form *See also one-argument form* Shape of

Forms the data in the right-hand argument into the 'shape' specified in the left- hand argument which must be a simple numeric scalar or vector. Excess elements are ignored. If there are not enough the data wraps around

```
      2 3 ρ 1 2 3 4 5 6       (The numbers 1 to 6 are to be formed
1 2 3                          into 2 rows and 3 columns)
4 5 6

      3 6 ρ 'ABCDEFGHIJKL'    (The 12 characters 'A' to 'L' are to
ABCDEF                         be formed into 3 rows of 6 columns.
GHIJKL                         Since there aren't enough different
ABCDEF                         characters for 3 rows, the last row
                               repeats the first 6 characters)

      2 2 ρ 1 2 3 4 5         (The numbers 1 to 5 are to be formed
1 2                            into 2 rows of 2 columns. The super-
3 4                            fluous number is ignored)
      3ρ'ABC'                 (Simple right argument)
```

```
ABC
      3⍴⊂'ABC'                 (Nested right argument is copied 3 times)
ABC ABC ABC
      X←6                       (A single number is put in X and its size
      ⍴X                        is asked. Since the number has no dimensions
                                dimensions, the result is an empty vector)
      X←1⍴6                     The same number is put in X, but
      X                         is formed into a 1-element vector. When
6                               displayed, X contains 6, but its
      ⍴X                        size is 1 since it was defined as a
1                               vector and has the dimension of length)
```

To produce an empty array (for example to initialise a variable) the right argument may be any value and the left argument must contain at least one zero (corresponding to the empty axis or axes of the result).

```
      ⍴0 33⍴3
0 33
      ⍴0 45⍴'A'
0 45
```

The empty array has a prototype (see Chapter 1) which is the prototype of the right argument.

If, conversely, the right argument is an empty array, the prototype of the right argument occupies each position of the result.

```
      10⍴⍳0
0 0 0 0 0 0 0 0 0 0
      ' '=2 3⍴''               (Note the convention of using ''
1 1 1                           to indicate a character empty vector)
1 1 1
```

Since a scalar has no shape, a scalar can be produced by using an empty vector left argument to `⍴`:

```
      X←(⍳0)⍴1⍴6               (We deliberately create a vector -
      X                         1⍴6 - which is then forced to be a
6                               scalar
      ⍴⍴X
0
```

We could have equally used `⍬⍴` to produce the scalar (see Zilde).

`⍴` can be used for selective specification.

```
      ALF
ABCDEFGHIJKLMNOPQRSTUVWXYZ
      (5⍴ALF)←'.....'          (First 5 elements selected and used in
      ALF                       the specification)
.....FGHIJKLMNOPQRSTUVWXYZ
```

, Ravel

One-argument form *See also two-argument form* Catenate, Laminate

Ravel

Ravel converts data into a vector. If applied to a scalar, it produces a one- element vector. If applied to a matrix or higher dimensional array, it produces a vector made from the elements in the array.

```
      NUM ← ,34               (34 is converted to a 1-element vector)
      ⍴ NUM                   (An enquiry about the size of NUM
1                              produces the answer, 1)
      TABLE                   (TABLE contains 2 5-row columns)
1 3 7 3 8
3 6 2 8 1
      ,TABLE                  (TABLE is converted to a 10-element
1 3 7 3 8 3 6 2 8 1            vector)
      TWIG←2 4⍴'ABC' 1 2 (⍳3) (2 2⍴⍳4) 'DEF' (2 2⍴'CART') 102.2
      TWIG                    (Nested matrix, shape 2 4)
 ABC    1    2 1 2 3

 1 2   DEF   CA 102.2
 3 4         RT
      ⍴TWIG
2 4
      ,TWIG                   (Ravel produces a nested vector)
 ABC  1   2   1 2 3    1 2 DEF    CA 102.2
                       3 4        RT
      ⍴,TWIG
8
```

See also the function ∊ (enlist) which entirely removes nesting.

Ravel with axis

When used in conjunction with an axis specification, ravel can either increase or decrease the rank of its argument. Fractional axis specifications will increase the rank, whilst integer axis specifications will decrease the rank.

A fractional axis specification must be not more than one less than the first dimension or not greater than one more than the last axis. A new axis of length 1 is added in a position governed by the value of the axis specification. As with other axis operations this is affected by the value of `⎕IO`. With `⎕IO` set to 1, the default:

```
      ⎕IO
1
      MAT←2 2⍴⍳4
      MAT
1 2
3 4
```

```
      ,[.1]MAT                  (Add a length 1 axis before the first
   1 2                             axis)
   3 4
         ⍴,[.1]MAT                 (Shape of result)
   1 2 2
         ,[1.1]MAT                 (Add a length 1 axis between axes 1 and 2)
   1 2

   3 4
         ⍴,[1.1]MAT                (Shape of result)
   2 1 2
         ,[2.8]MAT                 (Add a length 1 axis after the second axis)
   1
   2

   3
   4
         ⍴,[2.8]MAT                (Shape of result)
   2 2 1
```

When used with an integer axis specification ravel will reduce the rank. The axes must be contiguous and in ascending order.

```
         SAT←2 3 2⍴⍳12
         SAT
   1  2
   3  4
   5  6

   7  8
   9 10
  11 12
         ⍴SAT
  2 3 2
         ,[2 1]SAT                 (Axes not in ascending order)
  AXIS ERROR
         ,[2 1]SAT
         ^
```

With a correctly formed set of axes, the rank of the result is one more than the difference between the rank of the right argument and the number of axes in the axis specification. The shape may be predicted by adding the lengths of the axes specified and combining the result with those axes left unspecified.

```
         ,[1 2]SAT                 (SAT is rank 3, two axes in the
     1  2                           axis specification.)
     3  4
     5  6
     7  8
     9 10
    11 12
         ,[2 3]SAT                 (Rank of result is 1+3-2 or 2, a matrix)
     1  2  3  4  5  6
     7  8  9 10 11 12
         ,[1 2 3]SAT               (Three axes in the specification)
   1 2 3 4 5 6 7 8 9 10 11 12      (Rank of result is 1+3-3 or 1, a vector)
         ⍴,[1 2]SAT                (Shape of result from adding lengths of
   6 2                              axes 1 and 2, plus length of axis 3)
```

If the axis specification contains an empty vector, the result will have a dimension of length one added after the last dimension of the argument.

```
      ⍴,[⍳0]SAT                (Empty vector axis specification)
2 3 2 1                        (Dimension added at the end)
```

Ravel with axis can be used for selective specification:

```
      ⍴(,[1 2]SAT)             (The variable SAT as used above)
6 2
      (,[1 2]SAT)←6 2⍴'ABCDEFGHIJKL'
      SAT
AB
CD
EF

GH
IJ
KL
```

, Catenate, Laminate

Two-argument form *See also one-argument form* Ravel

Catenate

Catenate joins data items together.

With single-element items and vectors, catenate works very simply.

```
      10,66                    (2 numbers are joined to form
10 66                           a 2-element vector)

      '10 ','MAY ','1985'      (3 vectors of characters are joined
10 MAY 1985                     to form an 11-element vector)
```

With matrices and other multi-dimensional arrays, catenate expects the dimension at which the join is made to be specified. (See `[]`, 'axis'.) If no dimension is specified, the last dimension is assumed. `⍪` (comma-bar) behaves in exactly the same manner as `,` except that the default dimension is the first. Again, if an axis is specified `⍪` will use that axis.

The dimension at which items are joined must be the same length. Thus if two matrices are joined 'at' the columns dimension, the columns must be the same length. If a scalar is joined to a matrix, it's repeated along the dimension at which the join takes place. The examples below assume `⎕IO` is 1, the default.

Given the following three matrices called A, B, and D

```
      A                   B                   D
1  2  3  4            5  6            13 14 15 16 17 18
7  8  9 10           11 12            19 20 21 22 23 24
      C ← A,B                       (A and B are joined to form C,
      C                              Since no dimension is specified,
1  2  3  4  5  6                     the join is at the last dimension
7  8  9 10 11 12                     ie the columns.   Note that A and
                                     B have the same number of rows.)

      C,[1]D                        (C and D are joined at the
 1  2  3  4  5  6                    first dimension, ie at the rows.
 7  8  9 10 11 12                    Note that C and D have the same
13 14 15 16 17 18                    number of columns.)
19 20 21 22 23 24

      A,[1]0                        (A single number is joined to A.
1  2  3  4                           The join is at the row dimension.
7  8  9 10                           The number is repeated along that
0  0  0  0                           dimension.)
```

Laminate

Catenate can only produce a result of the same dimension but of enlarged shape - a two-dimensional structure becomes a larger two-dimensional structure. Laminate joins two objects of identical shape and dimension to form a higher dimensional object.

```
      'ABC',[0.5]'DEF'              (Two 3-element vectors are joined
ABC                                  to form a 2-row, 3-column matrix.
DEF                                  Note the figure in square brackets
                                     and the fact that it is less than 1)
```

Laminate creates a new object which has the same shape as the constituent parts except for the addition of a new dimension.

So in the example above the original vectors are size 3. Their lamination produces a matrix of size 2 3. The dimension added by laminate is of size 2. This dimension is placed in respect to the old dimension according to the number in brackets. With `⎕IO` set to 1, the default, if this number is less than 1, the size 2 dimension goes before the old dimension. So in the example above the 2 goes before the dimension of size 3, giving a 2-row 3-column matrix.

If the number in brackets is greater than 1, the 2 goes after the old dimension. In the example below, 1.5 is specified. The new dimension of size 2 therefore goes after the existing dimension of size 3, giving a 3-row 2-column matrix.

```
      'ABC',[1.5]'DEF'
AD
BE
CF
```

If `⎕IO` is set to 0, then the examples above would be

```
      'ABC',[¯0.5]'DEF'
```

and

```
               'ABC',[0.5]'DEF'
```

respectively

⍪ 1st axis catenate

⍪ behaves in the same way as catenate (,), except that if no axis is specified, the FIRST axis is assumed rather than the last.

⌽ Reverse

Reverses the order of the numbers or letters in the right-hand argument. (See also ⍉, the transpose function.)

```
      ⌽ 1 2 3 4 5 6
6 5 4 3 2 1
      ⌽(1 2) (3 4) (5 6)      (The three element are reversed, but not
5 6  3 4 1 2                   their contents)
      ⌽ 'BOB WON POTS'
STOP NOW BOB
      TABLE
1 2 3 4 5
6 7 8 9 0
      ⌽ TABLE                 (When applied to a matrix, it
5 4 3 2 1                      reverses the order within each
0 9 8 7 6                      row. You can use the operator []-
      ⌽[1]TABLE                'axis' to make the rotation apply
6 7 8 9 0                      to a different dimension.)
1 2 3 4 5
```

By default reverse, `⌽`, applies to the last dimension. Thus, above, TABLE was reversed about its columns. The first axis reverse, `⊖`, behaves exactly as `⌽` but operates by default about the first axis. Both will respond in the same way to the axis operator. The axis operator will depend on the setting of `⎕IO`.

⌽ Rotate

The numbers or letters in the right-hand argument are shifted by the number of places specified in the left-hand argument. The shift is to the left if the left- hand argument is a positive number and to the right if it's a negative number.

```
      1 ⌽ 1 2 3 4 5 6          (Each number moves one place to
2 3 4 5 6 1                     the left. This displaces the first
                                number to the end of the line)
      3 ⌽ 'ABCDEFGH'           (Each letter moves left 3 places.
DEFGHABC                        This displaces the first 3 letters
                                to the end of the line)
      TABLE
1 2 3 4 5
6 7 8 9 0
      3 ⌽TABLE                 (The numbers in each row are moved
4 5 1 2 3                       3 places to the left. Equivalent to
9 0 6 7 8                       3⌽[2]TABLE)
      ¯2 ⌽'ABCDEFGH'           (The negative number means a shift
GHABCDEF                        to the right. All letters are
                                moved 2 places right)
```

Similar axis considerations apply to rotate. By default ⌽ applies to the last dimension (as in the example above. First axis rotate, ⊖, applies by default to the first dimension, but otherwise behaves similarly.

Reverse and rotate can be used for selective specification.

⊖ 1st axis rotate

⊖ behaves in the same way as rotate (⌽), except that if no axis is specified, the FIRST axis is assumed rather than the last.

⍉ Transpose

Monadic (one-argument) form:

Transpose reverses the order of the axes of an array. Thus the first row of a matrix becomes the first column and vice versa, and similarly for arrays of more dimensions. It has no effect on scalars or vectors.

```
            TABLE
      1 2 3
      6 7 8
            ⍉ TABLE
      1 6
      2 7
      3 8
            ⍉ 1 2 3
      1 2 3                          (Has no effect on a vector)
            DATA ← 12 4 9 ρ ι 432
            ρ DATA
      12 4 9
            ρ ⍉ DATA
      9 4 12
            DATA[1;;2]=(⍉DATA)[2;;1]
      1 1 1 1
```

Dyadic (two-argument) form:

Changes the order of the rows and columns in the right-hand argument according to instructions in the left-hand argument and selects a subset of the right argument. There must be as many elements in the left argument as there are dimensions in the right. This operation has most effect when applied to data which has more than two dimensions. There must be a number in the left-hand argument for each dimension of the result. The result can have any rank greater than zero and not greater than the right argument. Thus for a rank 3 result you must have the numbers 1 2 3 appearing at least once each in the left argument. The positions of the values within the left argument correspond to the axes of the right argument and the values of the left argument refer to the axes of the result.

There are two cases to consider. The first is where all numbers in the left argument are unique. In this case all axes (and all elements) of the right argument appear in the result.

```
            TABLE
      1  2
      3  6
      9 10
            2 1 ⍉ TABLE             (First element of left argument shows that
      1 3  9                         axis 1 of TABLE becomes axis 2 of result.
      2 6 10                         Same as one argument ⍉)
            1 2 ⍉TABLE              (Co-ordinates stay in their original
      1  2                           order so matrix is unchanged)
      3  6
      9 10
            ρDATA
      12 4 9
            ⍴3 1 2⍉DATA            (1st axis of DATA becomes 3rd axis of
      4 9 12                        result, 2nd axis of DATA the 1st, etc)
            DATA[10;3;7]=(3 1 2⍉DATA)[3;7;10]
      1
```

When there are repetitions within the left argument, then the appropriate axes of the right argument will be mapped together and the rank of the result will be less than that of the right argument. Thus if the left argument to ⍉ is 1 2 1 then axis 1 of the result is formed from axes 1 and 3 of the right argument. This is done by selecting those elements whose position is the same on those axes. The operation is selecting diagonals. A simple case is when a rank 1 result is specified (a vector):

```
      TABLE1
1  2
3  4
      1 1 ⍉ TABLE1           (Result is those elements whose row and
1 4                           column positions match - [1;1] and [2;2])

      ⍴DATA
12 4 9
      ⍴1 2 1⍉DATA
9 4
      DATA[4;3;4]=(1 2 1⍉DATA)[4;3]
1
```

If the axes that are being mapped together are of different lengths, those positions that are common are only as many as the length of the shortest axis.

Transpose can be used in selective specification.

↑ First

One-argument form *See also two-argument form* Take

First selects the first item of its argument. When the argument is an empty array, first returns the prototype of the array.

```
      ↑2 2⍴⍳4
1
      A←('A.S.FREEMAN') 35 15000
      A
A.S.FREEMAN 35 15000
      ⍴A
3
      ↑A                     (First item of A is a text vector)
A.S.FREEMAN
      ⍴↑A
11
      TABLE←2 2⍴(2 2⍴⍳4) (⍳5) ('TEXT') ('EVEN MORE TEXT')
```

```
         TABLE                   (2 row, 2 column nested array)
       1 2   1 2 3 4 5
       3 4

      TEXT   EVEN MORE TEXT
            ↑TABLE                  (First item is a 2 by 2 numeric matrix)
      1 2
      3 4
            ρ↑TABLE
      2 2
```

First can be used in selective specification.

↑ Take

Two-argument form *See also one-argument form* First

The left-hand argument of take specifies how many elements are to be selected from the right-hand argument in each of its dimensions. If the left-hand argument is positive, the elements are selected from the start of the appropriate dimension, if negative, from the end. The result is the data selected.

```
            5 ↑ 'A.S.FREEMAN'
      A.S.F
            ¯7 ↑ 'A.S.FREEMAN'
      FREEMAN
            3 ↑ 22 2 19 12
      22 2 19
            ¯1 ↑ 22 2 19 12
      12
            LIST←(2 2ρι4) (ι10)
            ρ↑LIST                  (Note that first removes depth)
      2 2
            ρ1↑LIST                 (Take does not affect the depth)
      1
```

If the left argument specifies more elements than the right argument contains, all elements are selected and the prototype of the array is added for each missing element:

```
            5 ↑ 40 92 11
      40 92 11 0 0
            ¯5↑40 92 11
      0 0 40 92 11
```

If the right argument is a matrix, the first number in the left argument specifies the number of rows to be selected, and the second, the number of columns:

```
            TABLE ← 4 3 ρ ι 12
            TABLE
       1  2  3
       4  5  6
       7  8  9
      10 11 12
            2 3 ↑ TABLE             (Selects all three columns of the
      1 2 3                          first two rows)
      4 5 6
            ¯1 3 ↑ TABLE            (Selects all three columns of the
       10 11 12                      last row)
            1 2 ↑ TABLE             (Selects row 1, columns 1 and 2)
       1 2
```

The overtake operation on matrices or higher dimensional arrays uses the prototype of the first element of each row already in existence to extend rows. New rows use the array prototype.

```
      MAT
  1 A
  B 2
      ⍴MAT
2 2
      3 3↑MAT
  1 A 0                       (Extension of row 1 uses row 1 prototype)
  B 2                         (Row 2 prototype is a blank character)
  0 0 0                       (Row 3 is new and uses the array prototype)
```

Similar considerations apply to higher dimension arrays. Take can be used for selective specification.

Take used with axis

Take used with the axis operator will select only from the axes specified. Any axis not specified by the axis operator remains unchanged. Each successive element of the left argument indicates how many items to take from the corresponding axis within the axis specification (and from which end).

```
      MAT
  1  2  3  4
  5  6  7  8
  9 10 11 12
      2↑[1]MAT                (Take the first 2 members of the first
  1 2 3 4                      dimension, the rows, and leave the number
  5 6 7 8                      of columns unchanged)
      3↑[2]MAT                (First 3 columns, the second dimension)
  1  2  3
  5  6  7
  9 10 11
```

Overtake will follow the same rules as for take (see above).

```
      TABLE
1 A 2
B 3 4
      3↑[1]TABLE
1 A 2                         (New row uses array prototype)
B 3 4
0 0 0
      4↑[2]TABLE
1 A 2 0                       (Prototype of row 2 is the blank
B 3 4                          character)
```

↓ Drop

The number of elements specified in the left-hand argument are dropped from the right-hand argument. If the left-hand argument is positive, the elements are dropped from the left-hand end, if negative, from the right-hand end. The result is the original data without the dropped elements.

```
      4 ↓ 'A.S.FREEMAN'       (Drops the first 4 characters)
FREEMAN
      ¯6 ↓ 'A.S.FREEMAN'      (Drops the last 6 characters)
A.S.F
      3 ↓ 22 2 19 12          (Drops the first 3 numbers)
12
      ¯1 ↓ 22 2 19 12         (Drops the last number)
22 2 19
```

If the left argument specifies more elements than the right argument contains, all elements are dropped:

```
      5 ↓ 40 92 11
```

The result is in fact an empty vector, as we see if we apply ρ to the result:

```
      ρ 5 ↓ 40 92 11
0
```

If the right argument is a matrix, the first number in the left argument specifies the number of rows to be dropped, and the second, the number of columns:

```
      TABLE ← 4 3 ρ ι 12
      TABLE
 1  2  3
 4  5  6
 7  8  9
10 11 12
      2 0 ↓ TABLE               (Drops the first two rows, but NO
 7  8  9                        columns)
10 11 12
       ¯3 0 ↓ TABLE             (Drops the last three rows)
1 2 3
       1 2 ↓ TABLE              (Drops the first row and the first
 6                              two columns)
 9
12
```

Similar considerations apply to higher dimension arrays. Drop may be used for selective specification.

Drop with axis

Drop used with the axis operator will drop only from the axes specified. Any axis not specified by the axis operator remains unchanged. Each successive element of the left argument indicates how many items to drop from the corresponding axis within the axis specification (and from which end).

```
            MAT
    1   2   3   4
    5   6   7   8
    9  10  11  12
            2↓[1]MAT                (Drop the first 2 members of the first
    9  10  11  12                    dimension, the rows, and leave the number
                                     of columns unchanged)
            3↓[2]MAT                (Drop first 3 columns, the second dimension)
     4
     8
    12
```

⊂ Enclose

One-argument form *See also two-argument form* Partition

Enclose produces a scalar from its argument. If the argument is already a simple scalar the result is also a simple scalar, otherwise it has a depth of one greater than the argument.

```
            TABLE←2 3⍴⍳6
            TABLE
      1 2 3
      4 5 6
            ≡TABLE
      1
            ⍴⊂TABLE                 (Enclose produces a scalar)
                                    (Shape of a scalar is an empty vector)
            ⍴⍴⊂TABLE
      0                             (Rank of scalar is 0)
            ≡⊂TABLE
      2                             (Depth has been increased by 1)
```

Enclose with axis

When used with an axis specification, enclose will only enclose the axes indicated within the axis specification.

```
            ⊂[1]TABLE               (Enclose the  rows , leaving columns)
       1 4  2 5  3 6
            ⍴⊂[1]TABLE              (Result is length 3 vector)
      3
            ≡⊂[1]TABLE              (Depth increased by 1)
      2
            ⊂[2]TABLE               (Enclose the  columns  leaving rows)
       1 2 3  4 5 6
            ⍴⊂[2]TABLE              (Result is length 2 vector)
      2
            ≡⊂[2]TABLE              (Depth increased by 1)
      2
```

Enclose with axis can also be used to carry out a rearrangement of its argument (see also ⍉, transpose) by using a non ascending set of axes in the axis specification. Including all the axes in ascending order is equivalent to enclose.

```
      ⍴⊂[1 2]TABLE            (Same as ⊂TABLE)
 EMPTY
      ⍴⊂[2 1]TABLE            (Scalar result)
 EMPTY
      ⊂[2 1]TABLE             (Order of axes reversed)
1 4                           (Columns become rows within the first item
2 5                            of the result)
3 6
```

When the axis specification is an empty vector, enclose with axis has no effect on a simple array, but with non-simple arguments increases the depth of the argument by 1. Each item of the argument is enclosed, but the overall shape is not altered.

```
      ⍴TABLE                  (TABLE, as above)
2 3
      ≡TABLE                  (Depth 1)
1
      ⍴⊂[⍳0]TABLE             (Enclose with empty vector axis
2 3                            specification has no effect)
      ≡⊂[⍳0]TABLE
1
      TAB←2 2⍴(⍳3) (⍳3) 'ABC' 'DE'
      TAB                     (Nested matrix)
 1 2 3  1 2 3
 ABC    DE
      ⍴TAB                    (Shape 2 2)
2 2
      ⍴⊂[⍳0]TAB               (Enclose with empty vector axis
2 2                            specification preserves shape)
      ≡TAB
2
      ≡⊂[⍳0]TAB               (Increases depth)
3
```

⊂ Partition (with axis)

Two-argument form *See also one-argument form* Enclose

Partition will divide its right argument into an array of vectors according to the specification contained in its left argument. The left argument must be a scalar or a simple vector of integers that are either zero or positive, with one element for every item in the right argument. A new item is created in the result whenever the corresponding element in the left argument is greater than its predecessor. Elements in the left argument that are zero cause the corresponding items in the right argument to be omitted. If used without an axis specification, partition will select along the last axis. When used with an axis specification, selection takes place along the nominated axis.

```
      1 1 2 2 3 3⊂1 2 3 4 5 6
1 2  3 4  5 6                 (Result is 3 element vector, with each
      ⍴1 1 2 2 3 3⊂1 2 3 4 5 6 element a length 2 vector)
3
      1 1 0 1 1 0⊂1 2 3 4 5 6
1 2  4 5                     (Do not select 3rd and 6th elements)
      ⍴1 1 0 1 1 0⊂1 2 3 4 5 6
2
      MAT←3 3⍴'CATSATMAT'
      MAT
CAT
SAT
MAT
      ⍴MAT
3 3
      1 0 1⊂MAT              (Drop the second column)
C T
S T
M T
      ⍴1 0 1⊂MAT
3 2                          (Result is nested array)
      ⍴¨1 0 1⊂MAT
 1  1
 1  1
 1  1
      ≡MAT
1
      ≡1 0 1⊂MAT
2                            (Depth increased by 1)
      1 0 1⊂MAT
 C T
 S T
 M T
      1 0 1⊂[2]MAT           (Specification of  last  axis is the same
 C T                          as no axis specification)
 S T
 M T
      1 0 1⊂[1]MAT           (Specification of  first  axis causes
 C A T                        selection by first axis -  rows )
 M A T
      ⍴1 0 1⊂[1]MAT
2 3

      1 2 3⊂MAT              (Create a new element  every  column)
C A T
S A T
M A T
      ⍴1 2 3⊂MAT
3 3
      ≡MAT                   (MAT is depth 1 - a simple matrix)
1
      ≡1 2 3⊂MAT             (The partition of MAT is depth 2 - a nested
2                             matrix)
      1 2 2⊂MAT              (MAT is partitioned into two columns,
 C AT                         the first with one element, the second
 S AT                         with two)
 M AT
      ⍴1 2 2⊂MAT
3 2
      ⍴¨1 2 2⊂MAT
1  2
1  2
1  2
```

⊃ Disclose

One-argument form *See also two-argument form* Pick

Disclose will produce an array made up of the items in its right argument. If its argument is a scalar, then the result is the array that is within that scalar, and, in this form, disclose will reverse the effect of enclose. However, if the argument to disclose is a nested vector, the result will be a matrix.

```
      TABLE←2 3⍴⍳6
      ⍴⊂TABLE                 (Result of enclose is a scalar)
                              (Shape of a scalar is an empty vector)
      ⍴⊃⊂TABLE                (Disclose reverses the enclosure)
2 3
```

The shape of the result of disclose is a combination of the shape of the right argument **followed by** the shape of the items in the right argument.

```
      ⊃(1 2 3) (4 5 6)        (Shape of argument is 2, and of each item
1 2 3                          within the argument is 3)
4 5 6
      ⍴⊃(1 2 3) (4 5 6)       (Shape of result is 2 3)
2 3
```

In general, each item in the argument of disclose must be of the same **rank**, or be a scalar. If some of the items are scalar or have different shapes, they will be padded to a shape that matches the greatest length along each axis of all of the items in the argument. The prototype of each item in the right argument will be used as the fill item.

```
      ⊃(1 2) (3 4 5)          (First element length 2, second length 3)
1 2 0                         (First element padded to length 3)
3 4 5
      ⊃(1 2 3) ('AB')         (First element length 3, second length 2)
1 2 3
A B                           (Second element padded to length 3)
```

This can be a simple way to make a matrix from a series of different length vectors (but see also `⎕BOX`).

```
      ⊃'JOE' 'JAMES' 'JEREMY'
JOE
JAMES
JEREMY
```

Disclose with axis

When used with an axis specification, disclose will combine the shape of the right argument and the shape of the items within the right argument according to the axis specification. The overall shape of the result is formed from the combination of the shapes as before, but the axis specification will indicate which axis or axes in the result will be formed from the shape of the items within the right argument.

```
      NUMS←(1 2 3) (4 5 6) (7 8 9)
      ⊃[1]NUMS           (Elements of the vectors within the right
   1 4 7                  argument form rows in the result
   2 5 8                  ith element becomes ith row)
   3 6 9
      ⊃[2]NUMS           (ith element becomes ith column)
   1 2 3
   4 5 6
   7 8 9
```

The same rules will apply for higher dimensional arrays. Thus when forming a rank 3 array from a vector of matrices:

```
      DATA←(2 3⍴⍳6) (2 3⍴'ABCDEF')
      DATA               (Length 2 vector of shape 2 3 matrices)
   1 2 3 ABC
   4 5 6 DEF
      ⊃[1 2]DATA         (First and second axes of result made up
     1A                   from shape of elements of right argument.
     2B                   ith plane, jth row from ith row jth col
     3C                   of each element of right argument)

     4D
     5E
     6F
      ⊃[1 3]DATA         (First and third axes of result from shape
     1 2 3                of elements of right argument.
     A B C                ith plane, jth column from ith row jth col
                          of each element of right argument)
     4 5 6
     D E F
      ⊃[2 3]DATA         (Second and third axes of result from shape
     1 2 3                of elements of right argument.
     4 5 6                ith row jth column from ith row jth column
                          of each element of right argument)
     A B C
     D E F
```

Disclose with axis can also be used to carry out a rearrangement of its arguments (see also ⍉, transpose) by using a non ascending set of axes in the axis specification.

```
      ⊃[3 2]DATA         (Second and third axes of result made up
     1 4                  from shape of elements of right argument.
     2 5                  jth row ith column from ith row jth
     3 6                  column of each element of right argument)

     A D
     B E
     C F
```

⊃ Pick

Two-argument form *See also one-argument form* Disclose

Pick is used to select an item from its right argument according to the specification contained in its left argument. Each element in the left argument is used to specify successively deeper selections in the right argument. At each level of specification the element in the left argument being used must be of the appropriate shape - a single number for a vector, a two element vector for a matrix and so on.

```
      A←'FIRST' 'SECOND' 'THIRD'
      ⍴A                       (Three element vector)
3
      2⊃A                      (Pick the second element)
SECOND
      2 3⊃A                    (Pick the third element of the second
C                               element)
      A←(1 'FIRST') (2 'SECOND') (3 'THIRD')
      ⍴A                       (Three element vector, with each element
3                               a two element vector)
      3⊃A
3 THIRD                        (Third element selected)
      3 2⊃A
THIRD                          (Second element of third element selected)
      3 2 1⊃A
T                              (First element of second element of third
                                element)
```

When operating on arrays with two or more dimensions, care must be taken to ensure that the left argument to ⊃ is correctly formed.

```
      TABLE←2 2⍴(⍳3) 'NAMES' (2 2⍴4 5 6 7) (3 3⍴'ABCDEFGHI')
      TABLE
1 2 3  NAMES

  4 5   ABC
  6 7   DEF
        GHI
```

Selection of one of the outermost items from TABLE must be by means of a two element vector (given the shape of TABLE is 2 2), but this selection item must be formed as a scalar to indicate that it refers to the outmost layer.

```
      1 2⊃TABLE
RANK ERROR
      1 2⊃TABLE
      ^
```

In the example above, the left argument to pick is interpreted as 'first element from outermost layer' then 'second element from next layer deep'. A correctly formed left argument is:

```
      (⊂1 2)⊃TABLE
NAMES
```

```
      (1 2) 2⊃TABLE
A                               (Select row 1 column 2, then element 2)
      (2 1) (2 2)⊃TABLE         (Select row 2 column 1, then row 2
7                                column 2)
```

Pick may be used with selective specification, in which case the whole array picked will be replaced by the object being assigned.

⌷ Index

The ⌷ ('index') function selects from the array which forms its right argument according to the index array formed as its left argument. The left argument cannot be of depth greater than 2. Each element in the left argument addresses successive dimensions of the right argument and multiple index selections may be formed by creating a suitably nested vector. The dimensions specified in the left argument are used in the same order as with the ⍴ function, that is columns last, preceded by rows and so on. Index is affected by the Index Origin (⎕IO).

```
      2 ⌷ 1 2 3 4 5             (Scalar for vector indexing - only one
2                                dimension)
      (⊂3 4)⌷ 1 2 3 4 5         (Nested scalar for multiple index)
3 4
      TAB←2 5⍴⍳10
      TAB
1  2  3  4  5
6  7  8  9 10
      2 3 ⌷ TAB
8
      2 (2 3)⌷ TAB              (Second element of the indexing vector
7 8                              is the enclosed vector 2 3)
      (1 2) (2 3)⌷TAB           (Nested 2 element vector for multiple
2 3                              index. Result is rows 1 2 and columns
7 8                              2 3)
```

If the index function is given an empty left argument, and a scalar right argument, it will return the scalar as the result.

```
      (⍳0)⌷37
37
```

Index with axis

Index can be used with an axis specification. In this case the left argument only applies to those axes specified. Other axes are not indexed.

```
      2⌷[1]TAB                  (Select the second member of the first
6 7 8 9 10                       dimension - the rows)
      (⊂2 3)⌷[2]TAB             (Select the second and third members of the
2 3                              second dimension - the columns)
7 8
```

⍋ Grade up

Grade up enables numbers or characters to be sorted into ascending order. The arguments to grade up must be simple and not mixed. The right argument is a simple numeric or character array containing the data you want to sort. A left argument may be used to specify a sort sequence for character arrays.

The result is a vector which identifies elements by their position in the original data. For matrices or higher dimensional arrays, the sort is carried out on the first dimension. The result of grade up can be used to index the right argument into ascending order. ⍋ is affected by `⎕IO`, the index origin.

Identical elements or subarrays within the right argument will have the same relative positions in the result.

One-argument form

With the one-argument form, a numeric argument is sorted into ascending order. With a character argument `⎕AV` (the 'atomic vector') determines sorting order. It puts numeric characters before alphabetic characters and uses normal alphabetic order. So '1' is before (or less than) 'A', and 'A' is before 'Z'.

```
      ⍋13 8 122 4             (Produces vector showing ranking:
4 2 1 3                        4th number is first, 2nd number next)
      (13 8 122 4)[4 2 1 3]   (Ranking order is used as an index
4 8 13 122                     to put numbers in ascending order)
      ⍋'ZAMBIA'               (Produces vector showing ranking.
2 6 4 5 3 1                    'A' in position 2 is first)
      'ZAMBIA'[⍋'ZAMBIA']     (The ranking order found in the []'s
AABIMZ                         is used as an index to
                               put the characters in order)
      TABLE                   (A 3-row 3-column matrix of names)
BOB
ALF
ZAK
      ⍋TABLE                  (Ranks the names in alpha order)
2 1 3                         (By row)
      TAB
4 5 6                         (Sorts TAB by row)
1 1 3
1 1 2
      ⍋TAB
3 2 1
      TAB[⍋TAB;]              (TAB in ascending order)
1 1 2
1 1 3
4 5 6
      ARRAY                   (Three dimensional array is sorted by the
 2  3  4                       first dimensions, the planes)
 0  1  0

 1  1  3
 4  5  6
```

```
  1  1  2
 10 11 12
       ARRAY[⍋ARRAY;;]          (ARRAY in ascending order by planes)
  1  1  2
 10 11 12

  1  1  3
  4  5  6

  2  3  4
  0  1  0
       NAMES                     (Three dimensional character array)
 JOE
 DOE

 BOB
 JONES

 BOB
 ZWART
       ⍋NAMES
 2 3 1
       NAMES[⍋NAMES;;]
 BOB
 JONES

 BOB
 ZWART

 JOE
 DOE
```

Two argument form

The two argument form can only be used with simple character arrays. The left argument specifies the collation order you want to use.

```
       'ZYXWVUTSRQPONMLKJIHGFEDCBA' ⍋ 'ZAMBIA'
  1 3 5 4 2 6                    (Collation order reversed. Compare
                                 result with the example above)
```

The system variable `⎕A`, containing the alphabet, and the function `⌽` are used to reverse the alphabet in the next example:

```
       TABLE
 BOB
 ALF
 ZAK
       (⌽⎕A)⍋TABLE
 3 1 2                          (Compare with example above)
```

When the left argument is a character matrix (or higher dimension array), more sophisticated sorts can be devised. When elements of the right argument are found in the left argument they are assigned a priority depending on their position in the collation array. For this purpose, the last axis of the collating array is deemed to have most significance, and the first the least significance.

If elements in the right argument are not present in the collating array, they given priorities as if they were found at the end of the collating array and in the order of their occurrence in the unsorted right argument.

A common use of a matrix collation sequence is to carry out a case-insensitive sort. In the following example, lowercase characters are used in the array to be sorted. (Some implementations of APLX will use underlined letters instead of lowercase letters).

```
      DATA
ABLE
aBLE
ACRE
ABEL
aBEL
ACES
      COLL
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
      COLL⍋DATA
4 5 1 2 6 3
      DATA[COLL⍋DATA;]
ABEL
aBEL
ABLE
aBLE
ACES
ACRE
```

The collation array COLL contains lowercase characters in its second row. When the variable DATA is sorted, the first sort is by column order in the collation array. Thus rows in the matrix being sorted beginning with the letter 'A' or 'a' will be given highest priority, followed by 'B' or 'b' and so on for successive columns within the array being sorted. The next sort is by the rows of the collation matrix, and 'A' is given a higher priority than 'a' and so on. Contrast the example above, with a similar sort using a one dimensional (vector) collation sequence:

```
      COLL1
 AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
      DATA[COLL1⍋DATA;]
ABEL
ABLE
ACES
ACRE
aBEL
aBLE
```

Here, all rows beginning with 'A' are given a higher priority to rows beginning with 'a'.

⍒ Grade down

Grade down enables numbers or characters to be sorted into descending order. The arguments to grade down must be simple and not mixed. The right argument is a simple numeric or character array containing the data you want to sort. A left argument may be used to specify a collation sequence for character arrays.

The result is a vector which identifies elements by their position in the original data. For matrices or higher dimensional arrays, the sort is carried out on the first dimension. The result of grade down can then be used to index the right argument into descending order. `⍒` is affected by `⎕IO`, the index origin.

Identical elements or subarrays within the right argument will have the same relative positions in the result.

One-argument form

With the one-argument form, a numeric argument is sorted into descending order. With a character argument `⎕AV` (the 'atomic vector') determines sorting order. It puts numeric characters before alphabetic characters and uses normal alphabetic order. So '1' is before (or less than) 'A', and 'A' is before 'Z'.

```
      ⍒13 8 122 4             (Produces vector showing ranking: 3rd
3 1 2 4                        number is biggest, 1st is next etc)
      (13 8 122 4)[3 1 2 4]   (Ranking order used as index to put
122 13 8 4                     numbers in descending order)
      ⍒'ABRACADABRA'          (Produces vector showing ranking: 'R'
3 10 7 5 2 9 1 4 6 8 11        in position 3 is 'biggest', etc)
      KEY←⍒'ABRACADABRA'      (The ranking vector is put in KEY
      'ABRACADABRA'[KEY]       and is used as an index to put the
RRDCBBAAAAA                    original data into descending order)
      TABLE                   (A 3-row 3-column matrix)
BOB
ALF
ZAK
      ⍒ TABLE                 (Ranks the names in descending
3 1 2                          alphabetic order)
      TAB
4 5 6                         (Sorts TAB by row)
1 1 3
1 1 2
      ⍒TAB
1 2 3
      TAB[⍒TAB;]              (TAB in descending order)
4 5 6
1 1 3
1 1 2
      ARRAY                   (Three dimensional array is sorted by the
 2  3  4                       first dimensions, the planes)
 0  1  0

 1  1  2
10 11 12
```

```
 1  1  3
 4  5  6
      ARRAY[⍒ARRAY;;]         (ARRAY in descending order, by planes)
 2  3  4
 0  1  0

 1  1  3
 4  5  6

 1  1  2
10 11 12

      NAMES                    (Three dimensional character array)
JOE
DOE

BOB
JONES

BOB
ZWART
      ⍒NAMES
1 3 2
      NAMES[⍒NAMES;;]
JOE
DOE

BOB
ZWART

BOB
JONES
```

Two-argument form

The two argument form can only be used with simple character arrays. The left argument specifies the collation order you want to use.

```
      'ZYXWVUTSRQPONMLKJIHGFEDCBA' ⍒'ABRACADABRA'
1 4 6 8 11 2 9 5 7 3 10       (Collation order reversed. Compare
                               results with the example above)
```

The system variable `⎕A`, containing the alphabet, and the function `⌽`, are used to reverse the alphabet in the next example.

```
      TABLE
BOB
ALF
ZAK
      (⌽⎕A)⍒TABLE
2 1 3                         (Compare with the example above)
```

When the left argument is a character matrix (or higher dimension array), more sophisticated sorts can be devised. When elements of the right argument are found in the left argument they are assigned a priority depending on their position in the collation array. For this purpose, the last axis of the collating array is deemed to have most significance, and the first the least significance.

If elements in the right argument are not present in the collating array, they given priorities as if they were found at the end of the collating array and in the order of their occurrence in the unsorted right argument.

A common use of a matrix collation sequence is to carry out a case-insensitive sort. In the following example, lower case characters are used in the array to be sorted. (Some implementations of APLX will use underlined letters instead of lowercase letters).

```
      DATA
ABLE
aBLE
ACRE
ABEL
aBEL
ACES
      COLL
ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
      COLL⍒DATA
3 6 2 1 5 4
      DATA[COLL⍒DATA;]
ACRE
ACES
aBLE
ABLE
aBEL
ABEL
```

The collation array COLL places lower case characters in the second row of the collation matrix. When the variable DATA is sorted, the first sort is by column order in the collation array. Thus rows in the matrix being sorted beginning with the letter 'A' or 'a' will be given highest priority, followed by 'B' or 'b' and so on for successive columns within the array being sorted. The next sort is by the rows of the collation matrix, and 'A' is given a higher priority than 'a' and so on. Contrast the example above, with a similar sort using a one dimensional (vector) collation sequence:

```
      COLL1
 AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
      COLL1⍒DATA
2 5 3 6 1 4
       DATA[COLL1⍒DATA;]
aBLE
aBEL
ACRE
ACES
ABLE
ABEL
```

⊤ Encode

Represents a value in a given number system, for example, represents inches as yards, feet and inches.

The left-hand argument gives the base, or bases of the number system you want to use, the right-hand argument is the value to be encoded. Both arguments must be simple numeric arrays.

To convert 75 inches to yards feet and inches:

```
      1760 3 12 ⊤ 75          (There are 12 inches to a foot, 3 feet
2 0 3                          to the yard, 1760 yards to a mile)
```

Note: Since three numbers were required in the result (yards, feet and inches) three numbers were given in the left argument. If you don't put sufficient numbers in the left argument, you lose some of the result:

```
      3 12 ⊤ 75
0 3
```

You can be sure of not losing any of the result by making the first element of the left argument a number greater than the number to be encoded.

```
      100000 12⊤75
6 3
```

To express the base-10 number 100 in base-16 (hexadecimal)

```
      16 16 16 16 ⊤100
0 0 6 4
```

In addition, the right argument does not have to be an integer, and indeed ⊤ can be a handy way to separate the fractional part of a number from the integer part.

```
      1760 3 12⊤75.3
2 0 3.3
      0 1⊤75.3                (The second element of the left argument
75 0.3                         being 1 ensures that all of the right
                               argument except the fractional part
                               appears in the first element of the result)
```

Although the encode function is defined for scalar right arguments, it is possible to use encode with any array as the right argument. In this case the encode operation is applied to each element of the right argument to produce a vector result for each element. Similarly, if the left argument of ⊤ is not a vector, but a higher dimensional array, then each base vector across the first axis of the left argument is applied to obtain the representation of each element of the right argument. The shape of the result is the same as the shape generated by an outer product operation, `(⍴LEFTARG),⍴RIGHTARG`.

To convert a series of values expressed as decimal numbers to their binary (base 2) equivalent.

```
      2 2 2 2 2 ⊤ 1 2 3 4 5
0 0 0 0 0                       (The vector left argument is applied to
0 0 0 0 0                        each element of the right argument. The
0 0 0 1 1                        results are displayed along the first
0 1 1 0 0                        axis (rows) of the result)
1 0 1 0 1
```

⊥ Decode

Finds the value in units of a number represented in a particular number system, for example, how many inches are in one yard. In general, the result is a scalar value generated from a vector representation of a value.

The left argument contains the base (or bases) of the number system being used. The right argument is a value represented in the given number system. A scalar left argument is treated as if it is a vector which matches the length of the right argument. Similarly, a scalar right argument is extended to match the length of the left argument.

To reduce the vector 3 2 6 9 representing, say, the readings of the separate dials on a meter, to a single number:

```
      10 ⊥ 3 2 6 9              (10 is the base for the conversion)
3269
```

To convert a number represented in octal (base 8) to decimal:

```
      8 ⊥ 3 1                   (Note that as before, the right argument
25                               is a vector)
```

To reduce 1 yard 2 feet 8 inches to inches:

```
      1760 3 12 ⊥ 1 2 8         (12 is the base for converting feet to
68                               inches, 3 is the base for converting
                                 yards to feet. For 1760 see the note
                                 below)
```

If both arguments to ⊥ are vectors, they must contain the same number of elements. To make the left-hand argument up to the same length as the right, an extra number was included: 1760 (the conversion factor for miles to yards) is irrelevant to the conversion of yards feet and inches to inches, and any other sufficiently large value could have been used instead.

```
      2 2 2 ⊥ 1                 (The 1 is extended to match the length of
7                                the left argument)
```

To reduce 2 pounds 15 shillings 6 pence and 3 farthings to farthings (4 farthings to one penny, 12 pence one shilling, 20 shillings one pound):

```
            0 20 12 4 ⊥ 2 15 6 3
      2667
```

Again note the first number in the left-hand argument. Its only purpose is to make the arguments the same length.

The more general form of decode allows both left and right arguments to be numeric arrays. When the left argument is an array of rank 2 or more, it contains a set of vectors which describe different bases to be used independently. Each base lies along the last axis of the left argument, and is applied to each of the vectors on the first axis of the right argument. `⊥` follows the same rules as inner product, the length of the last axis of the left argument must match the length of the first axis of the right argument, and the shape of the result is given by deleting the two inner axes and joining the others in order.

To convert a matrix of yards, feet and inches to inches:

```
            TABLE
      1  1  1                  (The numbers to be decoded lie along the
      2  0  3                   first axis, so the first value is 1 yard
      0  1  8                   2 feet 0 inches and so on)

            1760 3 12 ⊥ TABLE  (The left argument is applied to each
      60 37 80                  column of the right)
```

⍺ Picture format

Displays the numbers in the right argument according to the instructions in the left argument.

Numeric left argument

`⍺` can be used in a similar way to the two-argument form of `⍕`. (See `⍕` for more information and examples.) The main differences are:

(a) If a number is too big for the field specified, `'*'`s are displayed.

(b) If the left argument is a single number, it specifies the number of characters in the field and no decimals are displayed.

(c) If the left argument consists of the numbers 1 or 2, only the absolute value of the data will be displayed.

(d) If any field width is specified to be 0, the result will not contain that column.

Character left argument

With `⍺` you can also define the way the data is to look by using editing symbols to build up the pattern you require. This 'picture' is enclosed in single quotes and forms the left-hand argument. Each number in the right-hand argument is displayed in the way defined by the picture. If the right argument is an array, each field in the specification is taken to apply to all of the relevant columns.

You can have one picture for all the numbers on the right, or several pictures, one for each number. If several pictures are defined, each one must be separated by a semicolon (;). If there is nothing between two semicolons, the previous picture repeats. Any valid APL character except ; may appear in the left argument.

Numeric Field Specification (9 Z)

The main editing symbols are Z and 9. If a 9 is used in the picture, a digit is displayed at that position; if the position is blank, a zero is displayed. Z causes a digit, if present, to be displayed, but it does not display leading zeros. If no 9 is found in a picture, full zero suppression is assumed, but a single leading 9, or a 9 with Z on either side of it has the special effect of forcing a display only if there is any significance to the right of the 9. More than one leading or embedded 9 causes a DOMAIN ERROR. A space in the picture causes a corresponding space in the number and a point . in the picture inserts a decimal point if required by the format specification. Absence of a decimal point means that none will be printed.

```
      '9999' ⍺ 101 15
01010015                      (Each number fills 4 positions - no
                               space between numbers was allowed for)
      ' 9999' ⍺ 101 15
 0101 0015                    (The space in the picture is put at the
                               beginning of each number. So if we
                               represent spaces by dots, we have:
                               .0101.0015)

      'ZZZZ' ⍺ 101 15 0 10
101  15      10               (Z suppresses non-significant zeros)

      'ZZZ9.99 ' ⍺ 11  12.1  13  .2
11.00   12.10   13.00    0.20 (Three leading zeros are suppressed,
                               but a zero in the digits position is
                               displayed. The point is inserted)

      'ZZZ9.ZZ' ⍺ 10 .3 .03 0 .009 .003
10.00   0.30   0.03          0.01  (The 4th and 6th numbers do not print)
```

A single or embedded residue symbol (|) behaves exactly like a leading or embedded 9 except that it forces significance in the field immediately to the right. Only one | may be used, and no 9s.

```
           'ZZZ|.ZZ' ⍺10 .3 .03 0 .009 .003
10.00    .30    .03           .01  (Note that there are no zeros
                                    before the '.' this time)
```

The floor symbol (⌊) used in a field specification behaves like the decimal point (.) except that it specifies where the point will print, rather than where it actually is. If a ⌊ symbol is used in a specification, the decimal point of the right argument is assumed to be at the right of the field, and the fractional part of the right argument is never displayed.

```
      'ZZZZZ9⌊99'⍺123456.78
1234.57                       (The number is treated as 123456 and the
                               . inserted before the last 2 digits)
```

Commas and other text characters will print where indicated if significance has started before they are reached. Text following the last Z or 9 will only print if the value of the field is negative. If the picture is too small for the formatted value, `*`s are used to fill the field.

```
      'ZZZ,ZZ9.99;ZZ9.99 CR;ZZZ,ZZZ.ZZ' ⍺ 101789.356 ¯22 7777777
101,789.36 22.00 CR**********

      'Z9/99/99'⍺011086
1/10/86                        (A date formatter)
```

Text may be placed between columns of the specification, and will repeat on every output line. The text must be placed within quotes (') and any number of such fields may be specified.

```
      'ZZ9;''  VERSUS  '';ZZ9' ⍺ 2 2⍴1 2 3 4
  1  VERSUS    2
  3  VERSUS    4
```

Negative numbers, floating characters, fill characters

The minus sign is not displayed unless specified in the picture (nor indeed is the plus sign). A `+`, `¯`, or `-` put at the beginning of the picture will cause the specified sign to be displayed where applicable. Negative numbers can alternatively be displayed in brackets, if brackets are placed round the picture. The symbols `[]`, (), `∆⍪` and `⊢⊣` are treated as alternative ways of displaying the minus sign.

The `+` sign or the various negative signs are shown at the very beginning of the relevant field. If you want the sign to appear immediately before the first displayed digit, use two of the signs at the beginning of the picture. This is known as 'floating' the character. Any character may be floated by placing it twice at the beginning of the picture. The second declaration is converted to a Z internally after the 'float' is noted.

```
      '-ZZ9.99 ' ⍺ 17 ¯2.3
17.00 -  2.30

      '((ZZ.99) ' ⍺ 17 ¯2.3
17.00 (2.30)

      '$$ZZ9.99 ;$ZZ9.99 '⍺ 35.45 33.75
$35.45 $ 33.75                 (Floating versus fixed character)
```

If a character is put at the beginning of a picture and followed by the `|` symbol, it will be used as the fill character instead of the normal blank. Any character except . and `|` may be used as a filler, and the declaration does not affect the resultant field length.

```
      '*|$$ZZ,ZZ9.99 O/D ' ⍺ ¯1174.57 303.75
**$1,174.57 O/D ****≤303.75*****
```

⍕ Format

One-argument form *See also two-argument forms* Format by specification, Format by example

⍕ , applied to any argument (character or numeric, simple or nested), converts it to characters according to the default display rules. (The formatted data may still look numeric since it is composed of the digits 0 to 9 together with suitable spaces and decimal points but it has the properties of character data and can be mixed in displays with other characters). The result is always a simple character array.

```
      QTY ← 1760 2
      ⍴ QTY                   (Asks the size of the data in QTY.
2                              Answer is 2 numbers.)

      QTY ← ⍕ QTY             (Formats data in QTY.)
      ⍴ QTY                   (Asks the size of the data in QTY.
6                              Answer is now 6 characters.)

      'PRICE IS ',⍕22×1.15    (The numeric data is formatted and joins the
PRICE IS 25.3                  character data to form a simple character
                               vector)
      DATA←(⍳3) (2 2⍴⍳4) 'TEXT' 100
      DATA
 1 2 3   1 2   TEXT 100
         3 4
      ⍴DATA
4
      DATA←⍕DATA              (Format preserves the appearance of an
      DATA                     array, but makes it into a simple
 1 2 3   1 2   TEXT 100        character array)
         3 4
      ⍴DATA
2 24
```

⍕ Format by specification

Two-argument form *See also one-argument form* Format

The right argument must be either a simple array, or have a maximum depth of 2 (no element higher rank than a vector).

Like the one-argument form, this version of ⍕ also converts numeric data to characters. The right argument is formatted according to the instructions in the left argument which is a integer scalar or vector. These instructions specify the width of each field in characters, and the number of decimal places to be displayed. If necessary, numbers are rounded in order to display them in the positions available.

If the first number in the left argument is 0, the system uses the specified number of decimal places, and as many other characters as are needed. If a single number is used for the left argument, it is treated as two numbers with the first set to 0.

Scaled (or scientific) notation can be forced if the second number of a pair of numbers in the left argument is negative. In this case, the negative number specifies the number of digits before the E character.

To display each number on the right in a field which is 10 characters wide and has 2 decimal places:

```
      10 2 ⍕ 13.8765390 6 87.213 23.1
13.88     6.00      87.21     23.10

      TABLE
2.77       1.731     22.9
11         0.3301    2.3
```

To display each column in TABLE as a 5-character field with no decimal places:

```
      5 0 ⍕ TABLE
 3   2    23
11   0     2
```

To force scaled notation:

```
      8 ¯2 ⍕ 7.1
7.1E000
```

To specify the number of decimal places while allowing the rest of the number as many character positions as it needs (including one leading space):

```
      0 2 ⍕ 22.1987 999.1
22.20 999.10

      ⍴ 0 2 ⍕ 11.7               (Asks the size of the formatted number.
6                                 It has been allocated 2 positions after
                                  the point, plus the 4 positions needed
                                  for the 2 integers, the point itself
                                  and a leading space)
```

Note: the above examples show a single pair of numbers in the left argument being applied in turn to each number in the right argument. The left argument can instead contain a separate pair of numbers (ie separate instructions) for each term on the right.

```
      10 2 8 3 ⍕ 279.5547 10.1234
279.55     10.123
```

Using `⎕FC` with format by specification

Certain elements in the system variable `⎕FC Format Control` can influence the display generated by `⍕` when acting as 'format by specification'. In index origin 1:

`⎕FC[1]` specifies the character used for the decimal point. (. by default).

`⎕FC[4]` specifies the overflow character used for numbers too wide for the column width specified. (0 by default, causing a `DOMAIN ERROR` on overflow).

`⎕FC[6]` specifies the negative number indicator. (`¯` by default).

```
      ⎕FC
,,*0_¯                        (Default settings)

      5 3⍕1000                (DOMAIN ERROR for overflow by default)
DOMAIN ERROR
      5 3⍕1000
      ^
      ⎕FC[4]←'*'
      5 3⍕1000                (Alternative overflow character)
*****
      ⎕FC[1]←','
      10 3⍕12.15
    12,150
      ⎕FC[6]←'/'              (Change negative number indicator)
      10 3⍕¯12.15
   /12,150
```

⍕ Format by example

Two-argument form *See also one-argument form* Format

The right argument must be a simple numeric array.

Like the one-argument form, this version of ⍕ also converts numeric data to characters. The right argument is formatted according to the instructions in the left argument which is a simple character vector. The left argument is used as a pictorial model of the format which should be applied to the left argument.

The left argument can either be one field, in which case that field is used to format each element or column of the right argument, or a series of fields to be applied, one to each column of the right argument. Numbers will be rounded to fit into their formatted layout.

A field is made up of characters drawn from the characters '0123456789' and . (full stop), , (comma) and a special 'print-as-blank' character, usually _ and set by `⎕FC[5]`. Fields are separated by either one or more spaces or by a character identified as a field separator by a special indicator in the field. Any other characters used in the left argument are treated as decorator┐ characters. Decorators may appear adjacent to the characters defining a field or within a field.

In common with other formatters, format by example permits decorator characters to:

- Appear always

- Appear if the number being formatted is negative

- Appear if the number being formatted is positive

- Float against the number being formatted, that is appear immediately next to the front or back of the number when it is formatted

The standard character used for format by example fields is 5, which is used to indicate simple formatting with removal of leading zeroes and suppression of trailing blanks. Zero values print as blanks.

```
      '55.55' ⍕22.234 1.398 11.00
22.23 1.4 11                  (Trailing blanks in 11 suppressed)
      '55.55 5.555 55.55 55'⍕22.234 1.398 0.00 11.0
22.23 1.398       11          (0 prints as blank)
      '555,555,555.55'⍕1234567.89
  1,234,567.89                (The , only appears between digits, leading
                               blanks are suppressed)
```

The control character 5 does not print positive or negative indicators (+ or -) and indeed will not accept negative numbers.

```
      '55.55'⍕ ¯10
DOMAIN ERROR
      '55.55'⍕¯10
      ^
```

Decorator characters which appear at the beginning or end of a field specification without special control characters will print where they appear in the left argument, they will not float.

```
      ' SALARY IS : $555,555,555.00' ⍕ 1234567.95
SALARY IS : $  1,234,567.95   ($ decorator does not float)
```

Negative numbers and floating decorators

The field control characters 1 and 2 should be used if negative numbers are likely to be found in the right argument of ⍕. They will control any decorators which appear at the beginning or end of a field specification. These control characters will print their associated decorators if the number being formatted is negative (the character 1) or positive (the character 2). In addition the decorator will float against the number being formatted.

```
      '¯155.55' ⍕10.1 ¯12.346 11.5
  10.1  ¯12.35  11.5          (Negative numbers with high-minus)
      '(155.55)' ⍕10.1 ¯12.346 11.5
  10.1  (12.35) 11.5          (Negative numbers in brackets)
      ⍴'(155.55)'⍕10.1 ¯12.345 11.5
24                            (Overall field size is the same, floated
                               characters which do not appear are replaced
                               by spaces)
      '+255,555,555.55'⍕ ¯101.34 1000234 13.1
         101.34  +1,000,234            +13.1
```

The control character 3 will purely float a decorator against a number being formatted, and will not accept negative numbers.

```
      'THE BALANCE IS : $555,555.55' ⍕ 10027.34
THE BALANCE IS : $ 10,027.34
      'THE BALANCE IS : $555,555.53' ⍕ 10027.34
THE BALANCE IS :  $10,027.34
```

In the example above the currency sign is floated against the amount. Note that the overall field length is the same and that decorators which are not next to the field specification do not float .

If the control characters 1, 2 or 3 appear in a field specification on their own they will apply to the decorators on both sides of the field. If two of these characters appear in a field specification, then each will apply to the decorators on its side of the number. In the example below, 1 acts with the minus sign on the left, 2 acts with the characters CR on the right.

```
      '-155,555.52CR'⍕ 101.34 ¯1000.29 15367.346
     101.34CR  -1,000.29    15,367.35CR
```

Finally, the control character 4 can be used to switch off the effect of the control characters 1, 2 or 3. In the example below, the 4 switches off the effect of the 1 such that, on the right of the numbers, the characters DEG always appear.

```
      '-154.5DEG  ' ⍕95.8 32.5 ¯27.2
95.8DEG    32.5DEG   -27.2DEG
```

Contrast the effect when the character 4 is omitted

```
      '-155.5DEG  '⍕95.8 32.5 ¯27.2
  95.8       32.5      -27.2DEG
```

In this example, the characters DEG print when the number is negative, under the control of the character 1.

Leading and trailing zeroes

The printing of leading and trailing zeroes can be forced by the control characters 0 and 9. One of these control characters placed in a field will indicate that 0s should be used up to that position. The effect of the 0 and 9 only differs in their treatment of the number 0. Control character 0 will print the appropriate number of 0s, control character 9 will use blanks.

```
      '55.55  '⍕21.1 27.25 33
21.1   27.25  33              (Trailing zeroes suppressed)
      '55.50  '⍕21.1 27.25 33
21.10  27.25  33.00           (Always print to two decimal places)
      '55.5055  '⍕21.1 27.12345 33
21.10    27.1235  33.00       (0 only forces printing of zero up to its
                               position in the field)
      '55.00  '⍕21.1 0 33
21.10    .00  33.00           (Control character 0 prints value 0)
      '55.59  '⍕21.1 0 33
21.10         33.00           (Control character 9 does not)
```

```
      '055,555.50' ⍕ 1000.1
001,000.10                    (Leading zeroes forced)
```

Cheque protection

Control character 8 fills empty portions of a field with the contents of `⎕FC[3]` (by default the `*` character).

```
      'TOTAL AMOUNT $385,555,555.00'⍕1000
TOTAL AMOUNT $******1,000.00
```

Alternative end of field delimiter and blanks within numbers

It is sometimes useful to format numbers with no spaces between them. This may be achieved by use of control character 6 which can be used to mark the end of a field.

```
      '5556/06/05 '⍕3↑⎕TS     (Three fields in left argument to ⍕)
1991/06/14
```

Contrast the example above with the next example which inserts a decorator within a number being formatted.

```
      '0555/55/55 '⍕19910614  (Only one field in left argument to ⍕)
1991/06/14
```

The 'print-as-blank' character (`⎕FC[5]` and `_` by default) can be used to insert blanks between the digits of a number without ending the field.

```
      '5.555_555_555_555_555  '⍕ ○1
3.141 592 653 589 790
```

Using `⎕FC` with format by example

Certain elements in the system variable `⎕FC Format Control` can influence the display generated by `⍕` when acting as 'format by example'. In index origin 1:

`⎕FC[1]` specifies the character used for the decimal point. (. by default).

`⎕FC[2]` specifies the character used for the thousands indicator. (, by default).

`⎕FC[3]` specifies the fill character for empty portions of a field when 8 is used in the field specification. (`*` by default).

`⎕FC[4]` specifies the overflow character used for numbers too wide for the column width specified. (0 by default, causing a DOMAIN ERROR on overflow).

`⎕FC[5]` specifies the character to be used in the field specification to indicate that a blank should be inserted between the digits of a number. (The default is `_`) .

```
      ⎕FC                         (Default setting for ⎕FC)
.,*0_¯
      '55.55' ⍕1000               (DOMAIN ERROR on field overflow)
DOMAIN ERROR
      '55.55'⍕1000
      ^
      ⎕FC[4]←'*'                  (Overflow character set)
      '55.55' ⍕1000
*****
      ⎕FC[1 2]←',.'               (Reverse characters used for decimal point,
      '555,555.55  '⍕1170.45       thousands indicator)
  1.170,45
      ⎕FC[3]←'|'                  (Fill empty positions with |)
      '$855555  '⍕1002
$||1002
```

⍎ Execute

Execute, followed by an APL text expression, causes the expression to be evaluated as if it had been entered at the keyboard in calculator mode. This has numerous applications, some of which are briefly summarized below.

It can be used to turn character data, which contains numeric characters only, into numeric data:

```
      LIST ← '345 567'
      ⍴ LIST
7                                 (LIST contains 7 characters.)
      ⍴ ⍎ LIST                    (LIST is executed, and ⍴ is applied to the
2                                  result - 2 numbers)

      1 + ⍎ LIST                  (This demonstrates that the
346 568                            executed form of LIST can be
                                   used in arithmetic)
```

It can be used as an alternative to branching in a user-defined function:

```
[4]   ⍎ (LOOP=10)/'DATA←DATA×10'
```

If LOOP does not equal 10 when line 4 is executed, the `/` operator will give an empty vector to `⍎`, and nothing will happen. If LOOP does equal 10, the `/` operator will pass the character data to `⍎`, and the value of DATA will be multiplied by 10 after execution.

In APLX, system commands can be executed using the `⍎` primitive:

```
      ∇LIB
     [1] ⍝ Show contents of library 0
     [2] ⍎')LIB'
     [3] ∇
```

The output from executed system commands can be captured in a variable:

```
            X←⍎')SYMBOLS'
            X
      IS 1026, USED 21
```

⍎ can be used to execute single line function definition statements. The implicit result of the operation is an empty vector, as is the result of executing any statement which does not have a result.

With an existing function called FUNCTION:

```
             ⍎'∇FUNCTION[3]A←2∇'
             ⍎'∇FUNCTION[2]B←1'      (Note ⍎ supplies the closing ∇)
```

With an existing function called FN:

```
             ∇FN[⎕]∇                 (Function with no result)
       [1] A←1 2 3
             ∇
             ⍴FN
       VALUE ERROR
             ⍴FN
             ^
             ⍴⍎'FN'                  (Execution gives an empty vector result)
       0
```

⊣ Stop

One-argument form *See also two-argument form* Left

The monadic primitive function ⊣ (stop) takes a right argument of any type, rank and shape. It discards the argument, and always returns a result which is a (non-printing) empty matrix. It can therefore be used to discard an unwanted result from another function:

```
      ⊣⎕mount 'c:\temp'
```

⊣ Left

Two-argument form *See also one-argument form* Stop

The function ⊣ (left) takes left and right arguments of any type, rank and shape. It discards the right argument, and passes the left argument through unchanged.

It can be used as a statement separator, where (unlike using ⋄ diamond) the actual expressions are evaluated in normal APL right-to-left order:

```
      x←1 2 3 ⊣ y←4 5 6 ⊣ z←7 8 9
      x
1 2 3
      y
4 5 6
      z
7 8 9
```

⊢ Pass

One-argument form *See also two-argument form* Right

The monadic function ⊢ (pass) simply passes its argument through unchanged. The argument can be of any type, rank and shape; the result is identical.

Although at first sight this does not appear very useful, it can be used to force the display of a result which otherwise would be non-printing:

```
      ⊢a←⍳10
1 2 3 4 5 6 7 8 9 10
```

⊢ Right

Two-argument form *See also one-argument form* Pass

The function ⊢ (right) takes left and right arguments of any type, rank and shape. It discards the left argument, and passes the right argument through unchanged.

It can be used to embed pseudo-comments in an expression:

```
      +/'Samples per test'⊢233 348 297
878
```

⎕ Evaluated input

If ⎕ appears to the right of the ← symbol or is referenced in some other way, it causes numeric input to be accepted from the keyboard and to be put into the variable named in the assignment. Valid APL expressions can also be entered whilst in ⎕ input mode, and their results will be returned by ⎕. System commands can also be entered whilst in ⎕ input mode, and their results will be printed and the ⎕: prompt redisplayed. An empty input in response to ⎕ input is not accepted, and the prompt is redisplayed.

```
      PRICE ← 12.50
      QTY ←⎕                   (⎕ causes ⎕: to be displayed as a
⎕:                              prompt to the user to type a number.
      50                        Here the user types 50. This is put in
      'VALUE IS ' (PRICE×QTY) QTY and the expression is evaluated)
VALUE IS 625

      QTY←⎕                    (If the user types any expression
⎕:                              yielding a numeric result, this is
      50+50                     accepted.)
      QTY
100

      QTY←⎕                    (The user types in a vector of numbers)
⎕:
      1 2 3 4 5
      QTY
1 2 3 4 5
      1 2 3+⎕                  (Input is requested and then used in
⎕:                              the expression)
      4
5 6 7
```

⎕ Output with newline

If ⎕ appears to the left of the ← symbol, it causes the result so far to be displayed. This may not be the result of evaluating the complete line as ⎕ can occur anywhere on the line. The data is output together with a newline (carriage return) character, and is displayed subject to the values of printing precision (⎕PP) and printing width (⎕PW).

```
      3+⎕←9-7
2                              (The intermediate result of 9-7)
5                              (The final result of 3+9-7)
      ⎕← COST← 28×5            (The result is put in COST but is also
140                             displayed)
      CODE←⎕←⌽'DEATHROW
WORHTAED                       (Reversed 'DEATHROW' put in CODE and
                                displayed. Note characters are accepted)
```

⍞ Character input

The ⍞ symbol causes the computer to accept data typed on the keyboard. Whatever is typed is treated as characters, even if it is made up of the digits 0 to 9. (See ⎕ if you require numeric input, or alternatively use ⍎, (execute) to convert text data to numeric data.)

⍞ does not cause a carriage return when used for output.

```
      A←⍞                         (The cursor is placed at the beginning of
HELLO                              the next line and whatever is typed is
                                   accepted.  Note there is no prompt. The
                                   characters HELLO are put in A.
      A                            the contents of A are displayed)
HELLO

      X←⍞
12                                (12 is put in X as a 2-character data item)
      X+5
DOMAIN ERROR                      (The 12 in X is two characters . Characters
      X+5                          can't be used in arithmetic. See ⍎ if
      ^                            you want to convert characters to numbers.)
```

An extract from a user-defined function:

```
[3]   'PLEASE TYPE YOUR NAME.'
[4]   NAME←⍞                      (The response is put in NAME)
[5]   'THANK YOU ',NAME           (The contents of NAME are
                                   displayed after 'THANK YOU ')
```

The dialogue will look like this:

```
PLEASE TYPE YOUR NAME.
REGINALD
THANK YOU REGINALD
```

⍞ Bare output

In addition to its use for inviting and displaying keyboard input, ⍞ can be used to display values generated internally by APL statements, that's to say, a value or result can be assigned to ⍞. Bare output does not include a terminating newline (Carriage Return) character if it is followed by another bare output or character input. In addition, bare output does not include newlines if lines exceed printing width. Numeric values placed in ⍞ in this way (rather than from the keyboard) are treated as numeric.

```
      A←⍞←1000
1000                              (Note that the value is displayed
      A×3                          as well as being assigned to A)
      3000
```

An extract from a user-defined function

```
      [1]      ⍞ ← 'PLEASE TYPE YOUR NAME. '
      [2]      NAME←⍞
      [3]      'THANK YOU ',NAME
```

The dialogue will look like this (the user types the name REGINALD)

```
      PLEASE TYPE YOUR NAME. REGINALD
      THANK YOU                       REGINALD.
```

Note that there's no carriage return after the ⍞ on line 1 - the name is typed in on the same line as the text.

Note too the spaces when the name is output. The exact form of the result of ⍞ (here the variable NAME) will vary from implementation to implementation. In general, in a situation such as the one shown, APL notes the character position at which the response to ⍞ starts (REGINALD starts at position 24) and stores the response preceded by a corresponding number of blanks. So the characters REGINALD preceded by 23 blanks are put in NAME and are subsequently displayed. (The system function ⎕DBR gets rid of blanks for you if you don't want them.) Check with your implementation notes issued in case the rules are different for your system.

The system variable ⎕PR (which is set to be a blank character by default) controls the characters used to replace the prompt. In the example above, if ⎕PR was set to some other character, then that character would be used in place of the 23 blanks. If ⎕PR was set to be an empty vector, then the actual prompt is returned. For more details see the entry for ⎕PR.

/ Reduction

When used with a function operand the / operator is known as Reduction (see the entry for Compression for the other functions derived from /). The context in which the / is used should make clear the operation being carried out. / can be applied to any dyadic function , including user defined functions. When used with a scalar or one-element vector integer left argument, the / operator is used to perform 'N-wise reduction'.

The left operand of / is inserted between all elements of the array right argument. In the absence of an axis specification, the operand is inserted between items along the last axis (see also the entry for [], the Axis operator).

```
            +/ 2 4 6                (This is the same as 2+4+6)
      12
            SALES←25 5.7 8 50 101 74 19
            +/SALES
      282.7                         (The sum of the numbers in SALES)
            ⌈/82 66 93 13           (The same as 82 ⌈ 66 ⌈ 93 ⌈ 13.
      93                             The result of 93⌈13 is compared
```

```
                                  with 66; the result of this comparison
                                  is compared with 82; the result of the
                                  last comparison is the largest)
      ∨/0 1 1 0 0                (The same as 0 ∨ 1 ∨ 1 ∨ 0 ∨ 0)
1                                (Used to test if there are any 1s)
      ∧/0 1 1 0 0                (Are there any 1's?)
      ,/ 'ABC' 'DEF' 'HIJ'
 ABCDEFHIJ
      ⍴,/'ABC' 'DEF' 'HIJ'       (Result is a scalar)
 EMPTY
      TABLE
1 2 3
4 5 6
      ×/TABLE                    (Multiply is applied to the elements
6 120                             of a matrix. Since no dimension is
                                  specified, it works on the last
                                  dimension, the columns. 6 is the
                                  result of multiplying the columns in
                                  row 1. 120 is the product of those
                                  in row 2)
```

`/` applies by default to the last dimension, whilst the similar operator, `⌿`, applies by default to the first dimension.

```
      ×/[1]TABLE                 (The [1] specifies that the operation
4 10 18                           is to apply across the 1st dimension,
      ×⌿TABLE                     the rows. Each element in row 1 is
4 10 18                           multiplied by the corresponding
                                  element in row 2.)
```

N-Wise Reduction

The definition of N-wise Reduction is very similar to the definition of Reduction. The left argument, an integer scalar or length one vector, is used to specify the length of successive subsets of the right argument on which the Reduction operation is performed. If the left argument is negative, each subset is reversed before the reduction operation is carried out.

For a left argument of absolute value n and the selected axis of the right argument of length m, the number of subsets to which the reduction can be applied are:

```
       1 + m - n
```

and thus the limiting case is where the sample size is 1 greater than the length of the selected axis, giving a empty result.

```
      2+/⍳10                     (Add up the numbers 2 at a time, starting
3 5 7 9 11 13 15 17 19            at the beginning of the vector)
      5+/⍳10                     (5 at a time)
15 20 25 30 35 40
      10+/⍳10                    (10 at a time - the same as ordinary
55                                Reduction)
      11+/⍳10                    (Sample size 1 greater than right argument
                                 empty result)
      DATA←3 4⍴⍳12
```

```
            DATA
  1  2  3  4
  5  6  7  8
  9 10 11 12
       2+/[2]DATA              (Add up 2 at a time across the columns
  3  5  7                       the second dimension)
 11 13 15
 19 21 23
       2+/[1]DATA              (Add up 2 at a time across the rows, the
  6  8 10 12                    the fist dimension)
 14 16 18 20
       NUMS←10?10
       NUMS
 2 8 5 6 3 1 7 10 4 9
       2-/NUMS                 (Subtract sucessive pairs of elements)
 ¯6 3 ¯1 3 2 ¯6 ¯3 6 ¯5        (Reverse the elements before subtracting)
       ¯2-/NUMS
 6 ¯3 1 ¯3 ¯2 6 3 ¯6 5
       2,/'AB' 'CD' 'EF' 'HI'  (Join elements, 2 at a time)
 ABCD CDEF EFHI
       3,/'AB' 'CD' 'EF' 'HI'
 ABCDEF CDEFHI
```

N-wise reduction is commonly used for moving averages. For example, if SALES is a vector of monthly sales figures, then

```
       (12+/SALES)÷12
```

gives the annualised moving average sales figures by month.

⌿ 1st axis reduction

`/` applies by default to the last dimension, whilst the similar operator, `⌿`, applies by default to the first dimension.

```
       ×/[1]TABLE              (The [1] specifies that the operation
  4 10 18                       is to apply across the 1st dimension,
       ×⌿TABLE                  the rows. Each element in row 1 is
  4 10 18                      multiplied by the corresponding
                                element in row 2.)
```

\ Scan

When used with a function operand, the \ operator is known as 'scan'. The type of operation being carried out will be apparent from the context in which the symbol is used. Scan `(\)` can be applied to any dyadic function, including user- defined functions.

The left operand of \ is any dyadic function. The effect is as if the function had been entered between all the elements of the data. In the absence of an axis specification, the function is applied to the last dimension. (This is similar to `/)`. A given element of the result consists of the result of applying the function repeatedly over all the positions up to it. In each case the general rule for the order of execution is obeyed.

```
      +\20 10 ¯5 7              (Compare with 20+10+¯5+7. The result shows
20 30 25 32                      the running totals and the final sum)
      ,\'AB' 'CD' 'EF'          (Repeated applications of ,)
AB ABCD ABCDEF
      TABLE
5 2 3
4 7 6
      ×\ TABLE                  (Puts × between all elements of TABLE
5 10  30                         and shows the result of each
4 28 168                         multiplication in row 1 and in row 2.
                                 Note that since no dimension was
                                 specified, the operation takes place
                                 on the last dimension, the columns.
                                 See [] - the axis operator)

      ×⍀TABLE                   (First axis scan. Applies across each
 5  2  3                         row, i.e. Down the columns. Same as
20 14 18                         ×\[1]TABLE)

      ∧\ 1 1 1 0 1 1            (Applies logical 'and' over all
1 1 1 0 0 0                      elements. A series of 1's is produced
                                 up to the first 0. Shows where a test
                                 first failed)

      -\1 2 3 4                 (The intermediate results are
1 ¯1 2 ¯2                        1
                                 1 - 2
                                 1 - 2 - 3
                                 1 - 2 - 3 - 4
```

Useful examples of Scan include:

```
      ∧\                         All 0 after the first 0
      ∨\                         All 1 after the first 1
      <\                         1 at the first 1
      ≤\                         0 at the first 0
      ≠\                         0 or 1, reversing at each 1
```

⍀ 1st axis scan

\ applies by default to the last dimension, whilst the similar operator, ⍀, applies by default to the first dimension.

```
      TABLE
5 2 3
4 7 6
      ×\ TABLE               (Puts × between all elements of TABLE
5 10  30                      and shows the result of each
4 28 168                      multiplication in row 1 and in row 2.
                              Note that since no dimension was
                              specified, the operation takes place
                              on the last dimension, the columns.

      ×⍀TABLE                (First axis scan. Applies across each
 5  2  3                      row, i.e. Down the columns. Same as
20 14 18                      ×\[1]TABLE)
```

/ Compression, Replication

When used with a simple numeric scalar or vector operand the / operator is used to perform the compression or replication functions. The context in which / is used will make the type of operation apparent.

Compression

The left argument is a vector of 1's and 0's. The right argument must conform in length but can be numbers or characters. With a matrix right argument the dimension on which the operator works must be of the same length as the left argument. For each 1 in the left argument, the corresponding element in the right argument is selected. For each 0, the corresponding element in the right argument is ignored. If a single 1 or 0 is used as the left argument, scalar extension ensures that none (0) or all (1) of the right argument is selected.

```
      0 1 0 1 / 'ABCD'       (The letters in the same positions as
BD                            the 1's are selected)
      1 1 1 1 0/12 14 16 18 20
12 14 16 18                  (20 corresponds with the only 0 and
                              is ignored)
      MARKS←45 60 33 50 66 19
      PASS←MARKS≥50          (Each mark greater than or equal to
      PASS/MARKS              50 puts a 1 in PASS. Those less
60 50 66                      than 50 produce 0's. The numbers
                              corresponding to 1's are selected)
      (MARKS=50)/⍳⍴MARKS     (Which members of MARK were 50?
4                             The fourth)
      1/'FREDERIC'           (The 1 or 0 left argument to /
FREDERIC                      can be used to select whether the
```

```
      0/'FREDERIC'            text is selected or not.)
(empty)
      TABLE←2 3 ρι6
      0 1 0/TABLE            (Select on the last dimension-columns)
2
5
      1 0/[1]TABLE           (Select on the first dimension-rows
1 2 3                         same operation as 1 0⌿TABLE)
```

The form of `/` shown with the text string FREDERIC is often used to control branching within functions. See the Reference section which covers Functions. The compression operation, `/`, applies by default to the last dimension, although it may be used in conjunction with the axis operator, `[]`. First axis compression, `⌿`, applies by default to the first dimension, but again may be used together with the axis operator. Remember that the axis operator is affected by `⎕IO`.

Replicate

This is used to generate multiple copies of elements of the right argument. In addition Replicate can be used either to replace a specified element with one or more instances of that element's prototype or to insert one or more instances of that dimension's prototype. Positive integers in the left argument specify how many copies of each corresponding element in the right argument are wanted.

Negative integers in the left argument are used to insert or substitute prototypes. The two alternative mechanisms for this case are:

(a) Length of left argument the same as the length of the selected dimension of the right argument. In this case, negative elements in the left argument specify that the corresponding element in the right argument should be replaced by the appropriate quantity of its prototype.

(b) If the number of non-negative elements in the left argument is the same as the length of the selected dimension of the right argument, then negative elements in the left argument indicate the position and quantity of prototype elements to insert - the prototype being used being that of the first element of the axis.

As usual, a scalar left argument is extended to match the selected axis. If a replication is carried out along an axis of length 1, that axis will be extended.

```
      2 ¯2 2/TABLE           (Replace second column of TABLE by
1 1 0 0 3 3                   2 columns of 0s - the prototype)
4 4 0 0 6 6
      2 ¯2 2 ¯2 2/TABLE      (Insert two sets of two columns of 0s)
1 1 0 0 2 2 0 0 3 3
4 4 0 0 5 5 0 0 6 6
      VEC←1 2 (2 2ρι4) 3 4
      VEC
 1 2   1 2   3 4
       3 4
       1 1 ¯2 1 1/VEC        (Insert two copies of the prototype of the
 1 2   0 0   0 0   3 4        third element of VEC)
       0 0   0 0
       1 1 ¯2 1 1 1/VEC      (Insert two copies of the prototype of VEC)
 1 2 0 0   1 2   3 4
           3 4
      2 3 2 / 'ABC'
AABBBCC
     2 / 'DEF'                 (With a scalar left argument, the 2 is
DDEEFF                          is extended to each element on the right)
      5 0 5 / 1 2 3
1 1 1 1 1 3 3 3 3 3
      2/TABLE                  (TABLE as above. Replicate on last
1 1 2 2 3 3                     dimension)
4 4 5 5 6 6
      2⌿TABLE                  (Replicate on first dimension. Same as
1 2 3                           2/[1]TABLE)
1 2 3
4 5 6
4 5 6

      2 3/3 1⍴'ABC'            (Last axis, the columns, is extended to
AAAAA                           length 5 to satisfy left argument)
BBBBB
CCCCC
      2 ¯1 2/[2]3 1⍴'ABC'      (Last axis extended and blank column
AA AA                           inserted)
BB BB
CC CC
```

⌿ 1st axis Compress, Replicate

/ applies by default to the last dimension, whilst the similar operator, ⌿, applies by default to the first dimension.

```
     TABLE←2 3 ⍴⍳6
     2/TABLE                   (Replicate on last dimension)
1 1 2 2 3 3
4 4 5 5 6 6
      2⌿TABLE                  (Replicate on first dimension. Same as
1 2 3                           2/[1]TABLE)
1 2 3
4 5 6
4 5 6
```

\ Expand

When used with a simple numeric scalar or vector operand, the `\` operator performs the function known as Expansion. The context in which the symbol is found should make it apparent which operation is being performed.

Two-argument form only

Inserts the array prototype. If the left argument consists of 1's and 0's, each 0 causes a space or 0 to be put in the corresponding position in the right argument.

There must be as many 1's in the left argument as there are elements in the right argument.

```
      1 1 1 0 1 1 1\'PIGDOG'  (The 1's represent the existing
PIG DOG                        characters in the right argument.
                               The 0 shows where a space is to go)
      TABLE
1 2 3 4  5
6 7 8 9 10
      0 1 1 1 1 1 \ TABLE     (Each row is to have a 0 inserted
0  1  2  3  4  5               before the existing numbers. Note
0  6  7  8  9 10               that the last axis is assumed)
```

The expansion function applies by default to the last axis, unless used in conjunction with the axis operator, `[]` (remember this is affected by `⎕IO`). The first axis expansion function, `⍀`, applies by default to the first axis, but otherwise behaves in the same way as the expansion function.

```
      1 0 1 \[1] TABLE        (Using the other axis, the
1  2  3  4  5                  same as 1 0 1⍀TABLE)
0  0  0  0  0
6  7  8  9 10
```

If the left argument includes numbers other than 1 or 0, a positive number specifies how many of the corresponding element to insert, and a negative number specifies the number of prototype elements to insert. There must be as many positive numbers in the left argument as there are numbers in the right argument. (See replicate under `/`).

```
      1 0 3 ¯2 5\ 3 8 2       (1 copy of first element, then 1 prototype,
3 0 8 8 8 0 0 2 2 2 2 2        3 copies of second element, 2 prototypes
                               5 copies of third element.
      VEC←(2 2⍴⍳4) 3 4 5 6    (Prototype is a simple numeric matrix
      1 1 0 1 1 1\VEC          shape 2 2 and is used by expand)
 1 2   3   0 0   4 5 6
 3 4       0 0
```

⍀ 1st axis expand

The expansion function applies by default to the last axis, unless used in conjunction with the axis operator, `[]` (remember this is affected by `⎕IO`). The first axis expansion function, `⍀`, applies by default to the first axis, but otherwise behaves in the same way as the expansion function.

```
      TABLE ← 2 5⍴⍳10
      1 0 1 ⍀ TABLE          (Using the other axis, the
1  2  3  4  5                 same as 1 0 1\[1]TABLE)
0  0  0  0  0
6  7  8  9 10
```

. Inner product

Inner product takes the form:

```
     DATA1  FN1 . FN2  DATA2
```

Where the operands, FN1 and FN2, are both dyadic functions, including user- defined functions. Inner product first combines the data along the last axis of the left argument with the data along the first axis of the right argument in an 'Outer Product' operation with the right operand. Finally a 'reduction' operation is applied to each element of the result.

If the two arguments are vectors of the same size, then the inner product gives the same result as FN2 being applied to the data and then FN1 being applied to the result in a reduction operation. (See `/` for reduction.)

```
      X ← 1 3 5 7
      Y ← 2 3 6 7
      X +.= Y                (This finds and totals the agreements
2                             between X and Y)
```

The above statement is equivalent to `+/X=Y` and involves the following steps:

```
      X=Y                    (Compares X and Y)
0 1 0 1                      (1 means agreement between elements)
      +/0 1 0 1              (Sums the agreements)
2
```

Using the same values of X and Y as above:

```
      X∧.=Y                  (Returns a 1 if all elements in
0                             X equal all elements in Y)
      X∧.=1 3 5 7
1
```

When applied to data of more than one dimension, such as matrices, the operation is more complex. For matrix arguments the shape of the result of the operation is given by deleting the two inner axes and joining the others in order. For example if we have:

```
             TABA of 4 rows and  columns
and          TABB of 5 rows and 6 columns
```

The inner dimensions are used by the inner product operation, and the result will be a 4-row 6-column matrix.

The operations take place between the rows and columns of the two matrices and are therefore the same as inner product operations between vectors as described above.

```
             TABLE1                          TABLE2
             1    2                          6  2  3  4
             5    4                          7  0  1  8
             3    0

             RESULT←TABLE1 +.× TABLE2
             RESULT
       20   2   5   20
       58  10  19   52
       18   6   9   12
```

The first number in RESULT is produced from row 1 of TABLE1 and column 1 of TABLE2.

```
             1 2 +.× 6 7             (Equivalent to +/1 2 × 6 7)
       20
```

Row 1 of TABLE1 is then used with each remaining column in TABLE2 to produce the first row of RESULT. Then row 2 of TABLE1 is used with each column of TABLE2 to produce the second row of RESULT and so on. So the 10 highlighted in row 2 of RESULT is derived from row 2 of TABLE1 and column 2 of TABLE2:

```
             5 4 +.× 2 0             (Equivalent to +/ 5 4 × 2 0)
       10
```

The operation shown above is the Matrix Multiplication operation. The operation can have non-scalar operands:

```
             X
       1 2 3
       4 5 6
             Y
       1 2 3
       4 5 6
       7 8 9
             X+.,Y                   (Columns of Y catenated to rows of X
       18 21 24                       and the results added up)
       27 30 33
             ⍴X+.,Y
       2 3
```

Other useful combinations are:

```
          A∧.=B   Instances of vector B in matrix A
          A∧.≠B   Finds where there is no single match of vector B in
                  in matrix A
          A+.=B   Gives a count of agreements between A and B
          A+.∊B   Give a count of memberships of B in A
```

These may, of course be extended to higher dimensional arguments. The general definition of inner product is given below. For the inner product operation

```
      DATA1 FN1.FN2 DATA2
```

the result is defined as

```
      FN1/¨ (⊂[⍴⍴DATA1]DATA1)∘.FN2 ⊂[1]DATA2
```

∘. Outer product

This involves two data items and a function. The function can be any dyadic function, including user-defined functions. The function operates on pairs of elements, one taken from the left argument and one from the right, till every possible combination of two elements has been used.

```
            X ← 2 3 4
            Y ← 1 2 3 4
            X ∘.× Y                 (Multiplies every number in X by every
      2  4  6  8                     number in Y generating a multiplication
      3  6  9 12                     table:
      4  8 12 16                                Y
                                       |  1    2    3    4
                                 X    2|  2    4    6    8
                                      3|  3    6    9   12
                                      4|  4    8   12   16

            0 1 2 3 4 ∘.! 0 1 2 3 4
       1 1 1 1 1
       0 1 2 3 4                    (Gives all possible combinations. See !)
       0 0 1 3 6
       0 0 0 1 4
       0 0 0 0 1
```

Note that this function always generates a result of one more dimension than the original arguments. Two vectors, for example, generate a matrix.

```
            1 2∘.,⍳3               (Combines each element of the left argument
         1 1  1 2 1 3               with successive elements of the right
         2 1  2 2 2 3               argument using the , function)
            ⍴1 2∘.,⍳3
       2 3                          (Shape of result 2 3)
            2 3∘.↑1 2               (The ↑, 'take', function is applied using
           1 0   2 0                 successive elements of the left argument
         1 0 0 2 0 0                 and right argument)
            ⍴2 3∘.↑1 2
       2 2                          (Shape of result 2 2)
```

The Outer Product will accept arguments of any shape and number of dimensions. The result will be an array whose shape is the shape of the left argument followed by the shape of the right argument. For example:

```
            A ∘.× B
```

where A is a matrix of 4 rows and 3 columns, and B is a matrix of 5 rows and 2 columns, will produce a result of shape 4 3 5 2 - a four dimensional array.

The result is as defined above, namely all possible combinations of the left and right arguments. The rule that shows the layout of the result is that, for

```
            R←A ∘.<FUNCTION> B      (where A and B are shaped as above)
```

The result, R, has a shape 4 3 5 2 and

```
            R[C;D;E;F]  is given by    A[C;D] <FUNCTION> B[E;F]
```

¨ Each

One-argument form

The ¨ ('each') operator applies its operand to each element of its argument. In the case of a scalar operand, or a scalar function, each has no effect.

```
      DAYS←'MONDAY' 'TUESDAY'
      ⍴¨DAYS
6 7
      DATA←(2 2⍴⍳4) (⍳10) 97.3 (3 4⍴'K')
      ⍴DATA                   (Length 4 nested vector)
4
      ⍴¨DATA                  (Shape of each element, note empty vector
   2 2  10    3 4              shape for element 3, the scalar)
      ⍴⍴¨DATA                 (4 shapes returned)
4
      ⍴¨⍴¨DATA                (The shape of each of the shapes - the
   2  1  0 2                   ranks - of each element)
```

Two-argument form

The two-argument form of each applies is left argument and its operand to each element of its right argument. Again, for empty left or right arguments, a fill function is applied.

```
      (1 2 3),¨4 5 6          (Joining successive pairs of elements in
 1 4  2 5  3 6                 the left and right arguments)
      2 3↑¨'MONDAY' 'TUESDAY' (2↑ of first element of right argument
 MO TUE                        3↑ of the second)
      2↑¨'MONDAY' 'TUESDAY'   (Scalar extension results in 2↑ of each
 MO TU                         element of the right argument)
```

```
      2 3ρ¨1 2                (2ρ of first element, 3ρ of second)
   1 1  2 2 2
      4 5ρ¨'THE' 'CAT'
THET CATCA
```

[ ] Axis

The highest dimension of a data item is considered to be the first dimension and the lowest dimension the last . Thus the first dimension of a matrix is the rows and the last dimension is the columns. In the case of a three-dimensional object, the first dimension is the planes followed by the rows and columns.

Axis numbers are governed by the Index Origin, `⎕IO`, and in Index Origin 1, (the default), the first dimension is represented by `[1]`, the second by `[2]` and so on. In Index Origin 0 the first dimension would be `[0]`, the second `[1]` and so on. The number used to represent the axis is always a whole number, except for the ravel and laminate functions.

The primitive functions and operators which will accept an axis specification include the dyadic forms of the primitive scalar functions :

```
+ - × ÷ | ⌈ ⌊ * ⍟ ○ ! ∧ ∨ ⍲ ⍱ < ≤ = ≥ > ≠
```

and some primitive mixed functions :

```
, ⍪       Ravel/Catenate/Laminate       (note first axis variant)
⌽ ⊖       Reverse/Rotate                (note first axis variant)
⊂         Enclose/Partition
⊃         Disclose
↑         Take
↓         Drop
⌷         Index
```

as well as the operators :

```
/ ⌿       Compress/Replicate            (note first axis variant)
/ ⌿       Reduce                        (note first axis variant)
\ ⍀       Scan                          (note first axis variant)
\ ⍀       Expand                        (note first axis variant)
```

Axis with scalar functions

When used with dyadic scalar functions (see above) the axis operator is placed after the function. The axis specified is a scalar or vector of axis numbers such that the number of axes specified is the same as the rank of the argument with the lower rank and all the axes specified must be found in the argument with the higher rank. Thus, for example, if the following expression is typed

```
vector  +[ AXES]   MATRIX
```

the left argument (vector) is rank 1 and the right argument (matrix) is of rank 2. The axes specified (axes) can only be a scalar or vector of length 1 and (in index origin 1) that axis can only be 1 or 2 (one of the two dimensions of matrix).

```
      A←⍳3                        (Vector A)
      B←3 4⍴⍳12                   (Matrix B)
      A+[1 2]B                    (Cannot have two axes specified with a
AXIS ERROR                         vector argument - the left argument)
      A+[1 2]B
      ^
      A+[3]B                      (3 is higher than the highest dimension
AXIS ERROR                         of B - the higher rank argument)
      A+[3]B
      ^
      A+[2]B                      (Axis specification is valid for length
LENGTH ERROR                       and value, but the length of A - the
      A+[2]B                       lower rank argument - does not match the
      ^                            size of the second dimension of B - the
      A+[1]B                       columns)
 2  3  4  5
 7  8  9 10                       (A valid example)
12 13 14 15
      B+[1]A                      (The left or right argument may be of
 2  3  4  5                        higher rank)
 7  8  9 10
12 13 14 15

      MAT←2 3 4⍴⍳24
      MAT
 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 10×[1]MAT                 (Vector multiplied across the first
  1   2   3   4                    dimension of MAT. Result has the same
  5   6   7   8                    shape as MAT)
  9  10  11  12

130 140 150 160
170 180 190 200
210 220 230 240
      TAB←2 3⍴1 5 10 10 50 100
      TAB×[1 2]MAT                (Higher dimension example)
   1    2    3    4
  25   30   35   40
  90  100  110  120

 130  140  150  160
 850  900  950 1000
2100 2200 2300 2400
      TAB×[2 1]MAT                (Order of axes is immaterial)
   1    2    3    4
  25   30   35   40
  90  100  110  120

 130  140  150  160
 850  900  950 1000
2100 2200 2300 2400
```

Multiple axis specifications cannot contain repetitions.

The next condition for axis with a scalar function is that the dimensions of the lower rank argument must be the same as the selected dimensions of the higher rank argument. When multiple axes are specified, they are used in ascending order, irrespective of the order in which they are entered.

Thus, for the example above, if vector is of length 5 and the axis specified is 1 (rows), then matrix must have 5 rows. If the axis specified is 2, matrix must have 5 columns.

Given correctly shaped arguments and valid axis specifications, the lower rank argument is applied across the dimensions of the higher rank argument specified by the axis operator. The result will have the shape of the higher rank argument.

Axis with mixed functions and operators

When an operator or mixed function which accepts the axis operator is applied to data, it works on the last dimension, unless another dimension is specified. Alternatively, you can use the 'first-axis' functions and operators (see Ravel, Catenate, Rotate, Compress, Expand, Scan and Reduce) which are specially defined to apply by default to the first dimension. To specify a different dimension, enclose the number representing the dimension in square brackets, and put it after the operator or function.

```
      TABLE
 1  2  3  4                   (TABLE has 2 rows and 4 columns)
50 60 70 80
      +/ TABLE                (No dimension is specified, so the
10 260                         add takes place on the last
                               dimension, ie across the columns
                               giving the sums of the rows)
      +/[1] TABLE             (The first dimension is specified
51 62 73 84                    so the add is on the rows,
                               giving the sums of the columns.)

      +\TABLE
 1   3   6  10                (The operator acts on the last dimension)
50 110 180 260

      TABLE,13 14             (TABLE is joined with the vector 13 14.
 1  2  3  4 13                 This takes place at the last dimension,
50 60 70 80 14                 the columns making a new column)

      TABLE,[1] 13 14 15 16   (This vector is joined at the
 1  2  3  4                    rows, making a new row.)
50 60 70 80
13 14 15 16

      ⌽ TABLE                 (The rotation is across the
 4  3  2  1                    columns. The same effect could be
80 70 60 50                    achieved by ⌽[2]TABLE)

      ⌽[1]TABLE               (The rotation is across the rows)
50 60 70 80
 1  2  3  4
```

⍬ Zilde

Zilde is a primitive constant, which contains an empty numeric vector. It is equivalent to `⍳0` or `0⍴0`.

```
      X←⍬
      ⍴X
0
      X≡0⍴0
1
      X≡''
0
      ⎕DISPLAY ''      ⍝ Empty character vector
┌⊖┐
│ │
└─┘
      ⎕DISPLAY ⍬       ⍝ Empty numeric vector
┌⊖┐
│0│
└~┘
      3↑⍬
0 0 0
```

⋄ Statement Separator

The ⋄ ("diamond") character acts as a statement separator, which allows you to place multiple statements on a single line. This works either in a function, or in desk-calculator mode. The left-most statement is executed first:

```
      QTY←4 ⋄ PRICE←2.5 ⋄ QTY×PRICE
10
      QTY
4
      PRICE
2.5
```

If an error occurs within one of the statements, execution is abandoned (the remaining statements are not executed).

The statement separator can be used with structured-control keywords:

```
      :Repeat 3 ⋄ "No!" ⋄ :End
No!
No!
No!
```

∇ Line Editor

∇ opens or closes *function definition mode*, a simple line editor (or 'del' editor) for editing functions, operators and classes. Although largely obsolete because APLX offers powerful on-screen editing facilities via the Edit menu or `)EDIT`, it is retained for compatibility with older systems. It is also sometimes useful for creating very small functions.

Editing functions and Operators

(For brevity, we use the word 'function' in this section to denote either user-defined functions or user-defined operators).

∇ followed by a name or function header (for a function which does not already exist in the workspace) opens definition mode. If the function already exists, you should follow it with just the name, not the full header.

The editor prompts you with the next line number, in square brackets. (Note that the function header is line 0.)

To enter a line for the line number which is being shown, just type the line. When you press Enter, the line will be fixed and you will be prompted with the next line number.

By entering line numbers and other characters in square brackets, you can control the editor, as in the following examples:

```
∇NAME[⎕]  Enter editor, open function NAME, list whole function
[⎕]       List function (once you have opened it)
[⎕4]      List from line 4 onwards
[3]       Overwrite line 3
[3] ...   Overwrite line 3 immediately
[5.1]     Insert new line after line 5
[∆2]      Delete line 2 from the function
[4⎕6]     Place cursor at line 4, character position 6
[4⎕0]     Place cursor at end of line 4
[0⎕0]     Place cursor at end of the function header
```

To insert a line, use a fractional line number between the line numbers of the lines on either side of the insertion point. For example, `[3.1]` will insert a line between existing lines 3 and 4 (and you will be prompted with `[3.2]` as the next line).

Note that you can edit the line number itself. This has the effect of copying the line to the new position, either inserting a new line, or overwriting an existing line.

When you have finished editing, type another ∇ character to end the edit session. Lines will be automatically re-numbered in sequence 1 to N, to allow for any insertions or deletions.

Defining or editing a class using the line editor

The line editor can also be used to create or edit a class, in much the same way as it is used to edit a function or operator. To define a new class, open the line editor by entering a line which begins with the del (∇) character, is followed by the header line of the class (optionally including a parent class name and localized names, as per the canonical representation), and which ends with a left curly brace. APLX will open the class definition, and prompt you with the name of the class in curly braces as a reminder that you are in class-edit mode. For example, we can create a new class `Sphere` which inherits from `Point`:

```
      ∇Sphere : Point {
{Sphere}:
```

You can then define properties by entering lines in the same format as the canonical representation of a class. After each line, APLX prompts again with the class name enclosed in curly braces:

```
      Radius←0
{Sphere}:
```

You can also enter methods by using the del editor in the normal way (you will be prompted with the line number until you finish editing the method, then return to class-definition mode and again be prompted with the class name):

```
      ∇R←Volume
[1]   R←1.333333333333×(○1)×Radius*3
[2]   ∇
{Sphere}:
```

Finally, exit from class-definition mode by entering a single right curly brace:

```
      }
```

The canonical representation of the class defined in this way would then be as follows:

```
      ⎕CR 'Sphere'
Sphere : Point {
Radius←0

∇R←Volume
R←1.333333333333×(○1)×Radius*3
∇
}
```

⍫ Lock

You can lock a user-defined function, operator or method by entering or leaving the line editor using the 'del-tilde' character (⍫) rather than 'del' (∇).

Once a function has been locked, it can be run, but cannot be edited or displayed. If you try to edit it, a DEFN ERROR will be reported.

If execution of a locked function is stopped because of an error or interrupt, the function is never suspended, but instead is abandoned. Any error within a locked function will cause a DOMAIN ERROR to be signalled to the caller.

Section 3: Errors

Overview of error handling and the State Indicator

Errors in calculator mode

If you enter a statement containing an error in calculator mode, APL responds with an error message. For example, if you attempt an operation on unsuitable data, you normally get a domain error:

```
      1 1 0 11 ∨ 1 1 0 0
DOMAIN ERROR
      1 1 0 11 ∨ 1 1 0 0
      ^
```

This error has occurred because the OR primitive function operates only on values 0 and 1, not 11 as supplied in the left argument. As the example shows, the statement containing the error is displayed with an error indicator (^) marking the point at which the APL interpreter detected the error. Depending on the version of APLX you are running, and your system preference settings, error messages are usually displayed in red, as shown above.

To correct an error in calculator mode, simply retype the statement correctly, or alternatively use the recall-line key (usually Ctrl-Up Arrow, or Cmd-Up Arrow on the Macintosh) to recal1 the statement, then edit it and re-enter it. In most versions of APLX, you can also correct it directly in the window, and then press Return or Enter to re-evaluate it.

Errors in user-defined functions or operators

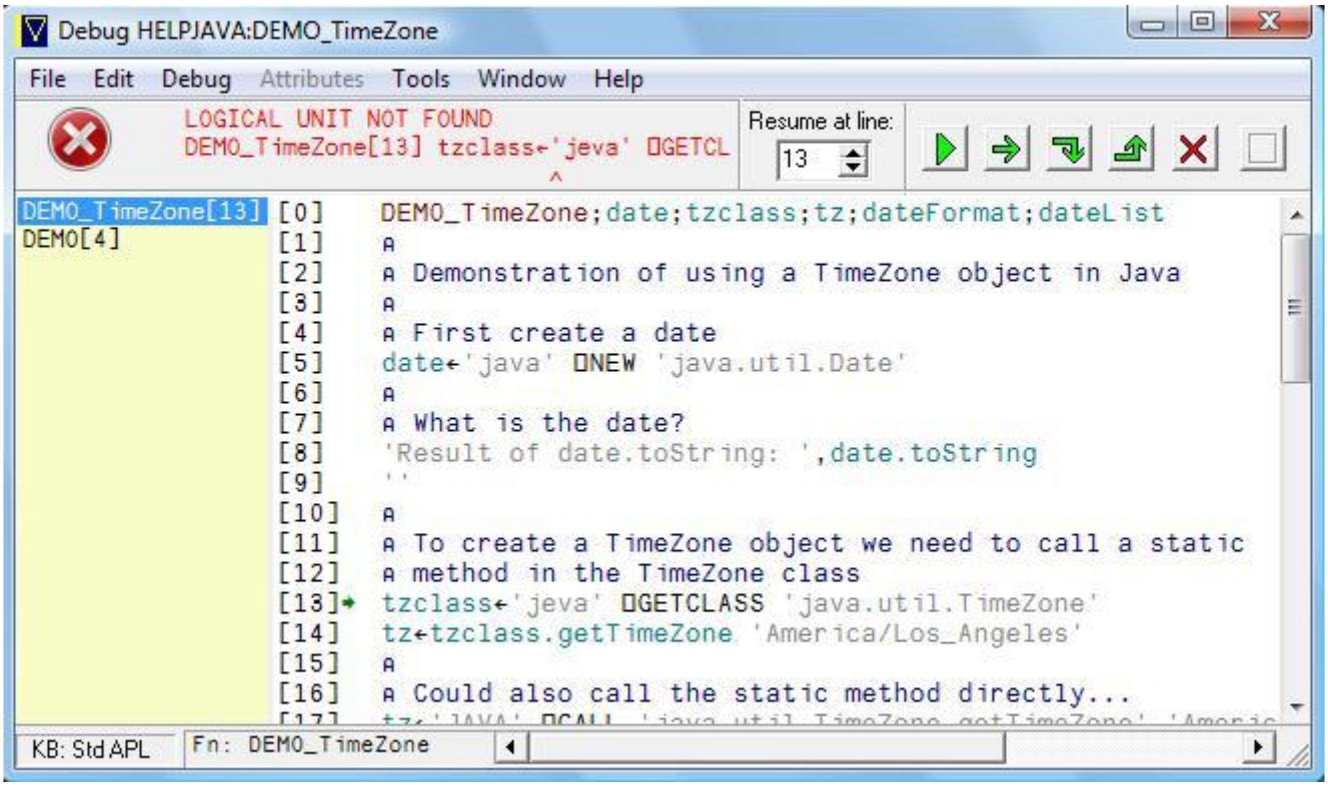
If an error is encountered during execution of a user-defined function or operator, execution stops at that point. The appropriate error message is displayed in the session window, followed on a separate line, by the name of the function containing the error, the line number at which execution stopped and the statement itself:

```
LENGTH ERROR
CALC[2] R←(X,Y)-1 2 3
          ^
```

The above example shows that execution stopped in function `CALC` at line 2.

The Debug Window

As well as displaying the error in the Session Window, desktop editions of APLX will normally display the Debug Window if an error occurs in a user-defined function, operator, or class method. This shows the function or operator in which the error occurred, and allows you to edit the line immediately and continue:


Debug HELPJAVA:DEMO_TimeZone
File Edit Debug Attributes Tools Window Help
Resume at line:
13
KB: Std APL

```
LOGICAL UNIT NOT FOUND
DEMO_TimeZone[13] tzclass←'jeva' ⎕GETCL
                                ^
```

```
DEMO_TimeZone[13]
DEMO[4]
```

```
[0]    DEMO_TimeZone;date;tzclass;tz;dateFormat;dateList
[1]    ⍝
[2]    ⍝ Demonstration of using a TimeZone object in Java
[3]    ⍝
[4]    ⍝ First create a date
[5]    date←'java' ⎕NEW 'java.util.Date'
[6]    ⍝
[7]    ⍝ What is the date?
[8]    'Result of date.toString: ',date.toString
[9]    ''
[10]   ⍝
[11]   ⍝ To create a TimeZone object we need to call a static
[12]   ⍝ method in the TimeZone class
[13]→  tzclass←'jeva' ⎕GETCLASS 'java.util.TimeZone'
[14]   tz←tzclass.getTimeZone 'America/Los_Angeles'
[15]   ⍝
[16]   ⍝ Could also call the static method directly...
[17]   tz←'JAVA' ⎕CALL 'java.util.TimeZone.getTimeZone' 'Americ
```

```
Fn: DEMO_TimeZone
```

In this example, an error has occurred on line 13 of the function, so execution has stopped there. Normally you would edit the incorrect line in situ (in this case correcting the spelling mistake 'jeva' instead of 'java'), and then press the Run button (the solid triangular arrow) to continue execution. You can also resume at a different line (by dragging the small green position indicator, currently on line 13, or by using the 'Resume at line' control), or abandon the function by pressing the Quit (red cross) button.

Interrupts

A function or operator can also be halted by the user hitting the interrupt key (usually Ctrl-Break on Windows, Cmd-Period on the Macintosh, or Ctrl-C under Linux). A single interrupt causes APLX to complete the line of code it is executing before stopping. Two interrupts in quick succession cause it to stop as soon as it can, even if it is executing a single calculation which takes a long time (for example inverting a matrix with `⌹`). The `⎕CONF` system function allows interrupts to be disabled.

Again, on desktop editions of APLX, the Debug window will appear if you interrupt a user-defined function, operator or method.

The State Indicator

It may be that the function at which execution halted was called by another function. You can inspect a system variable called `⎕SI`, the **State Indicator**, or use the system command `)SI`, to see the state of play:

```
      ⎕SI
C[2] *
B[8]
A[5]
```

This display (often referred to as the 'SI Stack') tells you that function `C` was called from line 8 of function `B` which was itself called from line 5 of function `A`.

The asterisk on the first line means that the function named on that line is '**suspended**'. The other functions are '**pendent**'; their execution cannot be resumed till execution of function C is completed.

If at this point you executed another function, `D`, which called function `E`, and at line 3 of `E` a further error occurred, the state indicator would look like this;

```
E[3] *
D[6]
C[2] *
B[8]
A[5]
```

Effectively it contains records of two separate sequences of events;

```
E[3] *
D[6]
--------------------
C[2] *
B[8]
A[5]
```

You can clear the top level of the state indicator (i.e. the record of the most recent sequence) by entering the branch symbol `→` on its own;

```
      →
      ⎕SI
C[2] *
B[8]
A[5]
```

In this example, another `→` would clear the remaining level (now the top level) and restore the state indicator to its original (empty) state.

Alternatively, you can clear the entire state indicator at any stage by using the system command `)SICL`.

Action after suspended execution

If you want to resume execution at the point where it stopped you can do so from the Debug Window as described above, or by using the symbol `→` followed by the line number. If, for example, execution halted at line 3 of `E`, to resume at that point you could type:

```
      →3
```

A system variable `⎕CL` contains the current line number, so you could achieve the same effect by typing:

```
      →⎕CL
```

You don't have to continue from the point where execution was suspended. You can specify a line other than the current line:

```
      →4
```

or

```
      →⎕CL+1
```

Equally, you can specify execution of a different function.

Editing suspended and pendent functions

What's perhaps most likely after an error in execution of a function is that you'll want to edit the function containing the error. (It's marked with `*` in the SI display and, as you may remember, is described as a *suspended* function.) This is done in the normal way by using `)EDIT` (or using `∇` and the function name to enter the del editor), and then making the required correction, or directly in the Debug Window.

It is possible that after editing the function you may get this message:

```
      SI DAMAGE
```

This indicates that you've done something which makes it impossible for the original sequence of execution to be resumed. No action is necessary other than to use the system command `)SICL` to clear the state indicator.

What you cannot do after a halt in execution is to edit any of the pendent functions. They are the functions in the state indicator display that are **not** marked with an asterisk:

```
      ⎕SI
E[3] *
D[6]
C[2] *
B[8]
A[5]
```

An attempt to edit a pendent function using the Del editor will produce a `DEFN ERROR`:

```
      ∇A
DEFN ERROR
      ∇A
       ^
```

Similarly, you can edit the function using `)EDIT A` but APLX will not let you save the changes because the function is pendent. You will get the error message "Cannot fix object - Function is on)SI stack"

If you want to edit a pendent function, simply clear the state indicator using `)SICL`.

Error trapping

You can specify in advance what should happen if an error occurs during execution, in which case that error will not cause execution to stop. For example, if you wrote a function which invited the user to type in some numeric data, you might foresee the possibility that he or she would type non-numeric data instead. This would cause an error. APLX allows you to 'trap' the error at runtime. There are two main ways of doing this:

- A block of code (including any functions called from within the block) can be executed under error-trapped conditions using `:Try..:EndTry`. If an error occurs, control passes to the `:CatchIf` or `:CatchAll` sections.
- Simple error trapping on a single line or expression can be achieved using `⎕EA`, which allows an alternate line of code to be executed in the event of an error, or `⎕EC`, which executes code under error trapped conditions and returns a series of result codes. These are compatible with IBM's APL2.

APLX also implements the older `⎕ERX` style of error-trapping, which specifies a line to be branched to if an error occurs. Use of `⎕ERX` is not recommended for new applications.

In general, it is probably best not to mix different styles of error-trapping in a single function. However, if you do, and an error occurs in a line where more than one error trap is live, then the error trap which will take effect is the first of:

1. `⎕EA`
2. `⎕EC`
3. `:Try... :EndTry`
4. `⎕ERX`

Error-related system functions

A number of system functions are available for finding out where an error occurred and why, or for simulating an error. These include:

- `⎕ERS` which can be used to signal an error (see also the APL2-compatible equivalent `⎕ES`).
- `⎕ERM` which displays the current error message (see also the APL2-compatible equivalent `⎕EM`).
- `⎕LER` which contains the error code and line number for the most recent error. Each kind of event that can be trapped has an error code. A DOMAIN ERROR, for example, is number 11. (See also `⎕ET` which holds the last error code in a format compatible with APL2).

Other debugging aids

- `⎕STOP` allows you to set 'breakpoints', i.e. specify that a function should stop at a given line. (Normally, the Debug Window will then be invoked). On desktop editions of APLX, you can also set or clear breakpoints by clicking in the line-number area of an Edit, Debug or WS Explorer window.
- `⎕TRACE` can be used to display a record of the results when certain 'traced' lines are executed.

Error trapping using `:Try..:EndTry`

Syntax:

```
:Try
...
:CatchIf <boolean expression>
...
:CatchAll
...
:EndTry
```

The block of code following the `:Try` keyword is executed, until either an error occurs, or a `:CatchIf`, `:CatchAll`, `:End` or `:EndTry` is encountered.

If no error has occurred within the `:Try` block, execution transfers to the statement after the `:End` or `:EndTry`.

If an error occurs in the `:Try` block (either in the statements in this function, or in any functions called from it), control transfers to the first `:CatchIf` statement (if any), and the boolean expression is evaluated. If it is true, the block of code following the `:CatchIf` is executed, and execution then resumes after the `:EndTry` or `:End`. If the expression is false, the same procedure is followed for any further `:CatchIf` blocks in the sequence. If none of the tests is true, the `:CatchAll` block (if any) is executed. It is permissible to have as many `:CatchIf` sections as you like.

Once an error has been trapped and control passed to a `:CatchIf` or `:CatchAll` statement, the error trap is disabled. Thus, if a second error occurs, it will not be trapped, and the function will stop in the normal way (unless the whole `:Try... :EndTry` sequence is itself executed under another error trap). `:Try..:EndTry` blocks can be nested.

Typically, you use the `:CatchIf` statement to catch specific types of error, by looking at `⎕LER` or `⎕ET`.

For example:

```
 ∇R←A DIVIDE B
⍝ Protected divide
 :Try
   ⍝ Do division under error-trapped conditions
   R←A÷B
 :CatchIf 11=↑⎕LER
   ⍝ DOMAIN ERROR occurred
   R←0
 :CatchAll
   ⍝ Some other error occurred
   'Unexpected error. The message was:'
   ' ',⎕EM
   →
 :EndTry
 ∇
```

```
      4 DIVIDE 3
1.333333333
      4 DIVIDE 0
0
      DIVIDE 4
Unexpected error. The message was:
 VALUE ERROR
 DIVIDE[4]   R←A÷B
               ^
```

Error Trapping (⎕EA, ⎕EC)

Note: The use of ⎕EA *is now deprecated, unless you need to retain compatibility with IBM's APL2. For most cases, we recommend that you use the structured-control error trapping mechanism* (`:Try :CatchIf :CatchAll :EndTry`) *instead.*

⎕EA and ⎕EC provide statement-level error trapping, using a syntax which is compatible with IBM's APL2.

⎕EA allows an APL expression to be executed under error trapped conditions. The right argument is a character vector containing an expression to be executed. The left argument is a character vector containing the APL expression to be executed if the right argument encounters an error or is interrupted.

If an error occurs in the alternate code of a ⎕EA call this is not trapped but is handled in the default (or non-trapped) manner.

⎕EC allows an APL expression to be executed under error trapped conditions. The right argument is a character vector containing the line of code. If the expression contains an error or is interrupted then ⎕EC returns a return code plus the error code given by ⎕ET.

⎕ET is a numeric vector containing the error code associated with the last error that occurred. The first integer indicates the general class of the error. The second integer indicates the specific nature of the error. ⎕ET can be used to identify the possible source of an error.

⎕EM is a character matrix containing the error message associated with the last error which occurred. The message contains the error description, the function name and line number where the exception occurred, the line of APL code where execution stopped, with a caret (^) pointing to the last character interpreted.

⎕ES is a function which simulates an error and causes execution of the active function or operator to stop. In the monadic form, the right argument is a two element numeric vector containing the error code. If the code is defined then the appropriate error description is displayed. If the code is undefined then no error description is displayed in the error message. If the right argument is zero or empty then no error is signalled. If the right argument is a character vector then that vector is displayed as the error description. In the dyadic form the left argument is the character vector error description and the right argument is the integer error vector.

If an error occurs in a locked function then the error message just gives the name of the function, with no internal details. The error description will usually be DOMAIN ERROR (sometimes WS FULL or INTERRUPT). ⎕ET similarly gives no indication of the true nature of the error. The same is true if a locked function calls an unlocked function which encounters an error. No internal details of the error are given. If a function containing a ⎕EA or ⎕EC statement is locked this does not affect the behaviour of the error handling internal to the function.

In the first example, `⎕EA` is used to handle the error:

```
      ∇R←A DIVIDE B
[1]   R←'A' ⎕EA 'A÷B'
      ∇
      3 DIVIDE 2
1.5
      3 DIVIDE 0               (Alternate execution invoked - returns
3                               left argument)
```

As an alternative, `⎕ES` is used:

```
      ∇R←A DIVIDE B
[1]   'ATTEMPT TO DIVIDE BY ZERO ERROR' ⎕ES(B=0)/5 4
[2]   R←A÷B
      ∇
      3 DIVIDE 4
0.75
      3 DIVIDE 0               (Error signalled)
ATTEMPT TO DIVIDE BY ZERO ERROR
      3 DIVIDE 0
      ^
      ⎕EM                      (⎕EM contains the message matrix)
ATTEMPT TO DIVIDE BY ZERO ERROR
      3 DIVIDE 0
      ^
      ⍴⎕EM
3 31
      ⎕ET                      (⎕ET contains the appropriate codes)
5 4
```

Finally, controlled execution allows the results and error messages (if any) to be studied:

```
      ⎕EC '3÷4'
1   0 0  0.75
      ⎕EC '3÷0'
0   5 4  DOMAIN ERROR          (Three element nested vector result)
              3÷0
              ^
      ⍴⎕EC '3÷0'
3
```

Error Trapping (⎕ERX)

Note: The use of ⎕ERX *is now deprecated. We recommend that you use the structured-control error trapping mechanism* (`:Try :CatchIf :CatchAll :EndTry`) *instead.*

The system function ⎕ERX allows you to set an error trap which will cause control to pass to a given line in a function, if an error occurs:

```
      ∇FOO;Z
[1]   Z←⎕ERX LABEL        Z will contain the previous value of ⎕ERX
```

Control will pass to LABEL when an error occurs in this function, (or a called function which does not have error trapping set). ⎕ERX returns the previous trap value. When an error occurs the normal error display of error message and line number is suppressed. A right argument of 0 to ⎕ERX suppresses the trap.

A non-error trapped function, called by an error trapped function, will behave as if it is locked. A branch will again take place to the designated line in the calling function.

Having transferred execution to an error handling routine, it is important to know the type of error that has occurred and also where it occurred. Sometimes the APL function can attempt some sort of corrective action, but often the error is logged and some message passed to the user.

The function ⎕LER returns the error code number (see below) and the line where the error took place, as a two element vector. If, when error trapping is active, an error occurs, the line number will refer to the most recent function to which the error has been propagated (i.e. the error trapped function).

To read the error message, the system function ⎕ERM shows the character vector that APLX would normally print with a Carriage Return (⎕R) between lines. Inside a locked function, ⎕ERM will show the error message that would be displayed if the function were unlocked. Outside a locked function, however, ⎕ERM is set to be an empty vector for security reasons.

```
      ∇FOO;Z;ER
[1]   Z←⎕ERX ERR
[2]   100×'A'
[3]   'THIS LINE WILL NOT BE REACHED'
[4]   ERR: 'INTERNAL PROGRAM ERROR'
[5]   ER←⎕R ⎕BOX ⎕ERM ⍝ FORM THE ⎕ERM VECTOR INTO A MATRIX
[6]   'ERROR MESSAGE:  ',(ER[1;]),' ON FUNCTION LINE ',⍕⎕LER[2]
      ∇
```

Example of Error Trapping

It is possible to encounter the error message WS FULL if you try to carry out operations using large arrays. Rather than have your function stop, you might like to check for this error state and undertake corrective action.

```
      ∇ADDUP;X;DATA
[1]   ⍝ADDS UP ALL THE NUMBERS UP TO THAT ENTERED
[2]   X←⎕ERX ERR ⍝              X IS USED TO HIDE THE RETURN FROM ⎕ERX
[3]   START:'ENTER A NUMBER'
[4]   DATA←⌊⎕  ⍝                MAKE IT THE NEXT LOWEST INTEGER
[5]   'THE SUM OF THE FIRST ',(⍕DATA),' NUMBERS IS: '
[6]   +/⍳DATA
[7]   →0   ⍝                    END OF THE FUNCTION
[8]   ERR:→(1≠1↑⎕LER)/REALERR ⍝ ERR CODE 1 IS WS FULL
[9]   'THE NUMBER YOU ENTERED WAS TOO BIG TO USE, TRY AGAIN'
[10]  →START
[11]  REALERR: ⍝                AN ERROR HAS OCCURRED WHICH IS NOT WS FULL
[12]  'ERROR TYPE ',(⍕1↑⎕LER),' ON LINE ',⍕1↓⎕LER
[13]  'MESSAGE IS:'
[14]  ⎕ERM
[15]  ∇
```

Make sure that you have some escape route from the error trap routine, otherwise any error within that section of the function will cause an uninterruptible loop. (The Interrupt key or menu item also causes an error - type 13)

Error Signalling

If some error report is to be made to the user of the system, it is useful to be able to modify the usual APL error messages, which may not be very meaningful to the end user. This can be carried out by the function `⎕ERS`. `⎕ERS` can be used to force a standard APL error report, or, if used with a character left argument, it will display those characters and assign the error code in its right argument to `⎕LER`.

Here is an example where `⎕ERS` is used to make sure that the user hasn't hit the interrupt key accidentally:

```
      ∇FOO;DATA
[1]   ⎕ERX ERR ⍝                SET ERROR TRAP
[2]   L:'ENTER YOUR EXPRESSION'
[3]   DATA←⍞
[4]   'THE RESULT IS:'
[5]   ⍎ DATA
[6]   →L
[7]   ERR: →(13≠1↑⎕LER)/NOTINT⍝   NOT INTERRUPT
[8]   'DID YOU MEAN TO HIT INTERRUPT? (Y/N)'
[9]   →('Y'≠1↑⍞)/L ⋄ ⎕ERS 13 ⍝ SIGNAL IF CONFIRMED
[10]  NOTINT: ⎕ERS 1↑⎕LER ⍝       SIGNAL OTHER ERRORS
      ∇
```

If the right argument is a number in the range 1 to 51, `⎕ERS` will display the standard APLX error message. As you will see later in the chapter, some error numbers are undefined, and in these cases `⎕ERS` will display UNKNOWN ERROR TYPE SIGNALLED. If the right argument is an empty vector, no error is signalled. Used with a character left argument, the error message may be altered. An empty vector left argument ('') will suppress the error message.

```
      ∇R←AV B
[1]   'NUMERIC ARGUMENT PLEASE' ⎕ERS (4=⎕DR B)/8
[2]   R←(+/B)÷⍴B
      ∇
      AV 1 2 3
2
      AV 'ABC'
NUMERIC ARGUMENT PLEASE
      AV
      ^
      ⎕LER
8 0
      ⎕SI
(empty response)              (the function has been halted)
```

Error Codes (⎕ET)

Error types reported by `⎕ET` are listed below. Note that some codes are unassigned.

0 0	NO ERROR	
0 1	UNKNOWN ERROR	
0 2	DEFN ERROR	
1 1	INTERRUPT	
1 2	SYSTEM ERROR	
1 3	WS FULL	
1 4	SYSTEM LIMIT	SYMBOL TABLE
1 8	SYSTEM LIMIT	ARRAY RANK
1 9	SYSTEM LIMIT	ARRAY SIZE
1 10	SYSTEM LIMIT	ARRAY DEPTH
1 11	SYSTEM LIMIT	PROMPT LENGTH
1 12	SYSTEM LIMIT	UNASSIGNED
1 13	SYSTEM LIMIT	TOKEN LIST LIMIT
2 1	SYNTAX ERROR	OPERAND OR RIGHT ARGUMENT OMITTED
2 2	SYNTAX ERROR	ILL FORMED LINE
2 3	SYNTAX ERROR	NAME CLASS
2 4	SYNTAX ERROR	INVALID IN CONTEXT/NONCE ERROR
2 5	SYNTAX ERROR	COMPATIBILITY SETTING PREVENTS THIS
2 10	SYNTAX ERROR	STRUCTURED-CONTROL ERROR
3 1	VALUE ERROR	NAME WITH NO VALUE
3 2	VALUE ERROR	FUNCTION WITH NO RESULT
5 1	VALENCE ERROR	
5 2	RANK ERROR	
5 3	LENGTH ERROR	
5 4	DOMAIN ERROR	
5 5	INDEX ERROR	
5 6	AXIS ERROR	
6 1	FILE ERROR	UNAUTHORISED FILE ACCESS
6 2	FILE ERROR	FILE NOT IN SYSTEM
6 3	FILE ERROR	COMPONENT NOT IN FILE
6 4	FILE ERROR	FILE ALLOCAION EXCEEDED
6 5	FILE ERROR	FILE OR COMPONENT HELD
6 6	FILE ERROR	FILE MAINTENANCE IN PROGRESS
6 7	FILE ERROR	USER ALLOCATION EXCEEDED
6 8	FILE ERROR	FILE IN EXISTENCE
6 9	FILE ERROR	FILE I/O ERROR
6 10	FILE ERROR	DISK FULL
6 11	FILE ERROR	USER NOT IN SYSTEM
6 12	FILE ERROR	DATA DAMAGED
6 13	FILE ERROR	FILE LOCKED
6 14	FILE ERROR	LOGICAL UNIT NOT FOUND

Error Codes (⎕LER)

Errors reported by ⎕LER are allocated integer Error Codes. Some error codes are unassigned, but these codes may still be used by the ⎕ERS function.

0	NO ERROR, OR ERROR RESET
¯1	OUT OF RANGE ARGUMENT FOR ⎕ES

1	WS FULL	17	FILE I/O ERROR
2	SYNTAX ERROR	18	FILE NOT IN SYSTEM
3	INDEX ERROR	19	UNAUTHORISED FILE ACCESS
4	RANK ERROR	20	COMPONENT NOT IN FILE
5	LENGTH ERROR	21	FILE ALLOCATION EXCEEDED
6	VALUE ERROR	23	FILE MAINTENANCE IN PROGRESS
7	VALENCE ERROR	24	FILE OR COMPONENT HELD
8	AXIS ERROR	25	INCORRECT COMMAND
9	SYSTEM ERROR	26	DATA DAMAGED
10	SYSTEM LIMIT	27	USER NOT IN SYSTEM
11	DOMAIN ERROR	28	USER ALLOCATION EXCEEDED
12	SYMBOL TABLE FULL	29	FILE IN EXISTENCE
13	INTERRUPT	40	DISC FULL
14	DEFN ERROR	41	FILE LOCKED
15	UNKNOWN ERROR	42	LOGICAL UNIT NOT FOUND
16	NONCE ERROR	43	STRUCTURED CONTROL ERROR

Error Messages

The various error messages that APLX will generate are shown below:

Message	Problem and corrective action
AXIS ERROR	The axis used is incorrect or the operator is not defined with axis or the axis specification contains invalid characters.
BUFFER FULL	Input line too long. Action: interrupt the display and shorten the line.
COMPONENT NOT IN FILE	The file does not contain the specified component, or the function was not found in the shared library.
COPY BUFFER FULL	Name list of)COPY command is too long. Action: shorten name list.
DATA DAMAGED	Error detected in the internal format of a variable.
DEFN ERROR	When using ∇ to create function: - function name duplicates name of an object already in the workspace, invalid header Action: change name of either, or erase object, correct the header. - the name you have used is invalid or locked. When using ∇ to edit a function: - you've included the argument names with the function name when attempting to edit an existing function. - the function is locked. - the function is pendant. (see the section on Error Handling) When editing body of function: - improper attempt at function line editing, for example a [, a number, but no closing].
DISK FULL	File dataspace is full
DOMAIN ERROR	You've used an APL function, but the arguments you have supplied are outside the domain of that function. For example: - You've tried to divide by zero. - You've tried to use one of the arithmetic functions (+ - × ÷) with characters - You've used fractional numbers with functions which require whole numbers (e.g. monadic ι or ?)
FILE ALLOCATION EXCEEDED	The file has reached its maximum allowed size

FILE IN EXISTENCE	Attempt to rename a file to an I.D. which already exists
FILE I/O ERROR	The host operating system has signalled to APL an error in some disc-related operation
FILE LOCKED	An incorrect file password has been used
FILE NOT IN SYSTEM	The file that is being accessed does not exist
FILE OR COMPONENT HELD	The operation cannot be performed due to an outstanding file or component hold by another user
INCORRECT COMMAND	You've typed a command starting with), but the remainder of the command is not correct or not recognised.
INDEX ERROR	When carrying out an indexing operation, you have used an invalid index. For example: - You have asked for element [5] of a 4 element vector
INTERRUPT	User interrupt.
I/O ERROR	APL encountered an error during input/output. Probably hardware failure or illegal ⎕MOUNT table.
LENGTH ERROR	Arguments are of unequal lengths, or the axes where the lengths of the arguments must match are unequal. For example: 2 3 + 3 4 5
LOGICAL UNIT NOT FOUND	The logical unit requested does not exist, or the shared library was not found, or the external class was not found.
NONCE ERROR	The expression you have typed is syntactically correct, but the interpreter does not support it at the moment.
NO SPACE	Not enough disc space to perform)COPY or)SAVE command
NOT COPIED....	Attempt to)PCOPY an object which exists in the active workspace.
NOT ERASED....	The objects shown were not found by the)ERASE operation.
NOT FOUND.....	Workspace does not contain the object. Action: check spelling of workspace or object name.
NOT GROUPED,NAME IN USE	Variable or function already has the name; Action:change name of group or erase conflicting object
NOT SAVED: THIS WS IS WSID	Normally occurs on attempt to save a workspace using a name which is not that of

	the active workspace and which duplicates the name of a workspace in the library. You have used the)SAVE command in the form `)SAVE NAME` to rename and SAVE the workspace. Action: rename the active workspace using)WSID, then save.
NOT SAVED, WS LOCKED	Occurs on attempt to save an active workspace with the same name as, but a different password from, a workspace already in the library. A locked workspace cannot be loaded, dropped, copied or saved-over without knowing the correct password. To change a password on a saved workspace,)LOAD the workspace,)DROP the workspace, then resave with a new password. Action: change the password of the active workspace using)WSID.
NOT WITH OPEN DEFINITION	The command cannot be processed while you are editing a function. Action: close the definition with ∇ and re-issue the command.
RANK ERROR	Function not defined for data of this structure or arguments are of incompatible rank. Action: provide argument of correct structure (single number/character, vector, matrix, etc)
SI DAMAGE	A pendant or suspended function has been replaced or removed by)COPY or)ERASE. Label lines of a suspended function have been edited. A function not at the top of the SI list has been edited, erased or copied. A function on the SI list has had its header edited. Action: clear the state indicator by)SICLEAR.
STRUCTURED CONTROL ERROR	A Structured Control keyword has been encountered but the context is wrong. For example, an :End may have been encountered but there is no current block active, or you have branched into an indented block. Action: Check the block structure keywords match up.
SYMBOL TABLE FULL	Too many names in use for the current symbol table size. Action:)SAVE the current workspace,)CLEAR the active workspace, increase the size of the symbol table using)SYMBOLS,)COPY the saved workspace back into the active workspace.
SYNTAX ERROR	Ill-formed expression, or incorrect number of arguments for a function. For example: - You have used a one argument function without a right argument. - You have unbalanced parentheses
SYSTEM ERROR	An internal hardware or software problem such as a memory fault. After this error a

	clear workspace is loaded automatically.
SYSTEM LIMIT	One of the system limits has been exceeded, for example the rank of an array.
UNAUTHORISED FILE ACCESS	The file's access matrix does not allow the operation from this user number. Action: modify access matrix
USER ALLOCATION EXCEEDED	User has too many files or the aggregate size of the file exceeds the user's quota.
VALENCE ERROR	A function has been used with too many or too few arguments - for example a left argument for a monadic function or a right argument only for a dyadic function.
VALUE ERROR	The name you've asked for does not exist, or you have referred to the result of a function which does not return a result.
WS FULL	Insufficient workspace. The active workspace cannot contain all the objects requested. During a)COPY command no objects are copied. Action: erase some variables or functions to make more space. Clear the state indicator using)SICLEAR.
WS LOCKED	You have used an incorrect password for a workspace that was saved with a password.
WS NOT FOUND	The workspace requested is not in the specified library or logical unit. Action: check the location of the workspace and the spelling of the workspace name.

Section 4: Component File Systems

Component files are APL files in which you can store arbitrary APL arrays or overlays. APLX supports two different component file systems. The first of these is based on the file-access primitives `⍇ ⍈ ⍐ ⍗` (as implemented in APL.68000), and the second is based on system functions such as `⎕FTIE`.

⍇ based File System

The APLX ⍇ based File System uses four primitive functions to transfer APL data between the active workspace and a file space located on a disk storage device:

⍇	File Read	('Quad-Read')
⍈	File Write	('Quad-Write')
⍐	File Hold	('Quad-Hold')
⍗	File Drop	('Quad-Drop')

APL files are identified by a file number, and components are accessed by component number. A file may be kept secure from other users by passwords, or by one of two methods of access control: control of access by user number, and a file or component hold facility.

The file system has been designed to facilitate casual use of the system without reducing the security features which may be required by more complex applications. Files are created automatically when the first write operation is performed.

Individual components may be any valid APL data, including overlays. The components keep their type and shape when stored and retrieved. Components may be added or inserted at any point in a file and any component may be deleted, even if it is located in the middle of a file.

APL files are located in a 'data space'. There may be several 'data spaces' throughout the system. Each 'data space' is independent of all other 'data spaces'. A utility program is supplied with APLX to create and maintain these 'data spaces', and this will be detailed in the system dependent notes. All file operations allow subscripts to select, via a logical unit number set in the `⎕MOUNT` function, which 'data space' is to be used in a given operation.

Basic File Operations

A file consists of a set of sequentially numbered components, each of which may be any APL variable. The components are referred to by their position within the file. Deletion of components or insertion of components within the file automatically renumbers the file in a manner similar to the renumbering of APL function lines during function editing. Files are created by the first valid write operation. Extensions to the built-in file functions allow information about the files, and their components to be read.

For each 'data space', the file system keeps tabs on the number of files each user owns and the size of those files. Each user has quotas which limit the number (if any) of files he may own, and the aggregate size of those files. In addition to the limits on a user, there are quota restrictions on the size of each individual file. The user is free to alter the default file size (which varies from system to system) upwards or downwards - subject, of course, to his overall quota.

Advanced File Operations

Each user of APLX can be allocated a user number (shown by `1↑⎕AI`), which allows each user in a multi-user environment to assume a unique identity. Individual APLX files are tagged with a User Number, and have an associated File Access Matrix which indicates which users can access the file and what operations they may perform. Users will be allocated their user number by the logon procedure adopted by their system. Each user can thus 'own' a number of files and the user can grant or deny access to these files.

The Access Matrix is two columns wide. The first column is a list of user numbers `-` with 0 being taken to mean ALL users. The second column is a list of integers which indicate the access privileges for the indicated user. When a file is created, the default access matrix allows only the owner to access it, and grants to the owner all accesses except File-Delete. An Access matrix may have a maximum of 29 rows.

The access privileges can be given in two ways.

A positive privilege states what the user can do, and a negative privilege states what the user cannot do.

The privilege code is effectively a number generated by adding various powers of 2 (1,2,4,8,16,....), each power of 2 corresponding to a particular privilege. Positive privilege codes are merely the sum of the individual privileges granted, whilst negative privilege codes are generated by adding `¯1` and the result of negating the sum of all the privileges denied.

Power of 2		Operation
0	(1)	Read components
1	(2)	1⍇ 2⍇ 3⍇
2	(4)	Insert Components
3	(8)	Append Components
4	(16)	Replace Components
5	(32)	
6	(64)	Delete a File
7	(128)	Delete a Component
8	(256)	
9	(512)	Set File Allocation
10	(1024)	Rename
11	(2048)	Hold/Release File
12	(4096)	Hold/Release Components
13	(8192)	
14	(16384)	6⍇ 7⍇
15	(32768)	1⍇
16	(65536)	2⍇
17	(131072)	3⍇
18	(262144)	
19	(524288)	Read Access Matrix
20	(1048576)	Write Access Matrix

For example:

Privilege	Meaning
0	No Access
1	Read Only Access
17	Read and Replace Access
¯1	Full Access
¯65	All Operations except Delete allowed

In addition to the access privileges afforded to users by the Access matrix, the Hold mechanism temporarily suspends file access by other users, whilst, for example, an updating operation is being carried out. Hold may be applied and released to whole files or components, and holds may be applied in two strengths - write access restricted and both read and write restricted.

For more information, see the descriptions of the file-access primitives:

```
⍇ Read, Get info, Rename
⍈ Write
⍐ Hold/Release, Change Quota
⍗ Delete
```

⍇ File read

One-argument form

⍇ reads data from a file. The right argument specifies the file and component number of the data required.

The one-argument ⍇ statement takes this form (`{}` means optional):

```
            R←⍇ {[LIBRARY]} FILE, COMPONENT {,USER, PASSWORD}
```

LIBRARY identifies the library volume from which data is to be read. If you omit the library number, library 0 is assumed. If included, the library number is put in square brackets. (see also the section on `⎕MOUNT`).

FILE identifies the file the data is to be read from (a positive integer)

COMPONENT identifies the data item you want to read (an integer). Component number 0 means the last component, and if the component number is omitted, it is assumed to be 0.

USER and PASSWORD are used for file security in shared file applications; the defaults are `1↑⎕AI` and 0 respectively if omitted (both integers). User number 0 also means the owner.

```
            ⍇ 3 300                (Component 300 is read from file 3 on
      2.6 7.1 3.3                   on library 0. The data is displayed)

            A←⍇ 2 40               (The data in component 40 of file 2 is
                                    read and is assigned to A)

            A←⍇[3] 10 19 1000 99   (Data  is  read from component 19  of  file
                                    10 on library 3, belonging to user 1000
                                    and with password 99)
```

A variant of the one-argument form is used for reading the file access matrix. Here the negative of the file number is used and the value returned is the present access matrix. (See the discussion of file access matrices above)

```
            A←⍇¯3                  (The file access matrix of file 3 on
                                    library 0 is assigned to A)
```

Two-argument form

The two argument form of ⍇ provides information about the files and components. The general syntax is:

```
            R ← A ⍇ {[LIBRARY]} FILE {,COMP,USER,PASSWORD}
```

The action of ⍇ in this form is governed by the value of A, as the next table shows (library, file, component ,`user` and password are as defined for the one argument form). The file number is required when information is sought about a given file and the component number is only used when information about a given component is sought. If the file number is omitted it defaults to 0, as does the component number.

A	File no.	Comp. no.	Result of ⍇
1	FILE	0	The number of components in the file
2	FILE	0	A nine-element file description vector
2	0	0	An eight-element user quota vector
3	FILE	COMP	A six-element component description vector
5	0	0	A vector of the file numbers belonging to the user, in ascending order
6	FILE	0	A three-element vector of file hold information
7	FILE	COMP	A three-element vector of component hold information

The file description vector comprises:

```
1 - File number
2 - Maximum allowed size in bytes
3 - Actual size
4 - Number of components
5 - Date file was created, MMDDYY
6 - Time file was created, HHMMSS
7 - Date file was last updated, MMDDYY
8 - Time file was last updated, HHMMSS
9 - Bytes attributable to file overhead
```

The user quota vector comprises:

```
1 - User number
2 - Aggregate file allocation quota
3 - Current aggregate file size
4 - Number of files quota
5 - Number of files in existence
6 - Allocation assigned to a new file
7 - Free space remaining in dataspace
8 - Largest contiguous space left in dataspace
```

The component description vector comprises:

```
1 - File number
2 - Component number
3 - User number
4 - Date component was written, MMDDYY
5 - Time component was written, HHMMSS
6 - Size of component in bytes
```

The file hold information request returns the following:

```
1 - Number of components held
2 - User holding the file (or 0)
3 - Hold restriction (0, 1 or 2)
```

And finally, the component hold vector:

```
     1 - Component number
     2 - User holding the component (or 0)
     3 - Hold restriction (0, 1 or 2)
```

For example:

```
             5⍇[1] 0                (A vector of all files on volume 1)
       1 117 10923478

             2⍇[1] 117 0            (The file description vector of file 117 on
                                     library 1)
       117 500000 388864 60 10383 2429 22384 223357 56
```

File Rename

The file may be renamed with a variant of the two argument ⍇ function. The general form of the operation is ({} means optional) :

```
            R←(NEWFILE,OVER,USER {,NEWPASS})⍇ {[LIBRARY]}OLDFILE,0,USER,{OLDPASS}
```

NEWFILE means the new file number

OLDFILE means the original file number

NEWPASS means the new password (if a password is to be changed it must be specified in both arguments)

OLDPASS means the original password

R, the result, is 1 for a successful rename; 0 if the operation failed

OVER means whether or not the rename may overwrite an existing file (except when changing the password).

```
      OVER = 0 means that overwriting is not allowed
      OVER = 64 means that overwriting is allowed
```

When overwriting, the file being overwritten must have delete access set, and the password (if any) must be correct. Otherwise a file locked error is shown.

⍈ File write

⍈ writes data to a file. The left argument is the data. The right argument identifies where it's to go. If the file specified as the destination already exists, the data is put in it. If it doesn't exist, ⍈ causes it to be created first. A component may only be written within the range of existing components (either by replacement or insertion), or be appended to the end of the file.

The full form of ⍈ is as follows (`{}` means optional)

```
R ← A ⍈ {[LIBRARY]} FILE,COMPONENT {,USER,PASSWORD}
```

A is any APL variable.

R (the result) is an empty vector with display potential off.

LIBRARY number identifies the library volume number to which the file is to be written. If you omit the library number, library 0 is assumed. If included, the library number is put in square brackets. (See also `⎕MOUNT` for a discussion on alteration of library numbers.)

FILE number identifies the file the data is to be written to (an integer).

COMPONENT number is the identifying number which shows the way in which the variable is to be put into the file:

`C=0` Append a new component to the end of the file

`C=integer` Replace an existing component, unless the number is 1 past the end of the file when it is appended

`C=fraction` Insert the component between the two integers on either side of C. Again append if C is less than 1 after the end of file. If C is omitted, it defaults to 0.

USER number and PASSWORD are used for file security in shared file applications; the defaults are `1↑⎕AI` and 0 respectively if omitted.

```
1 3 7 ⍈ 6 2            (The vector 1 3 7 becomes component 2 of
                         file 6 on library 0. If component 2
                         already exists it's overwritten.)

VAR⍈[2]506 3 1075       (VAR is written to component 3 of file
                         506 belonging to user 1075 on library 2)

(0 ⎕OV ⎕NL 3)⍈12 5      (All the functions in the workspace are
                         filed as an overlay, into file 12
                         component 5 - see also ⎕NL, ⎕OV)
```

A variant of ⍈ is used for updating the file access matrix. Here the negative of the file number is used and the left argument is the new access matrix.

Note:

Writing an access matrix to a non-existent file is a way to create an empty file.

```
            A ⍈ {[LIBRARY]} -FILE {,COMPONENT,USER,PASSWORD}

            (1 2⍴1000 ¯1)⍈[3]¯9     (File 9 on library 3 is set to FULL access
                                     for the owner)
```

⍐ File hold

One-argument form

⍐ in one-argument form alters the file allocation quota (how much the file may 'hold'). When a file is first created it is restricted to a given size. (This size will vary from system to system). The file allocation quota may be examined via `2⍇`. A file may be created by reading the file allocation quota. A file so created will have no components.

The general form is (`{}` means optional):

```
            R ← ⍐ {[LIBRARY]} FILE,ALLOCATION {,USER,PASSWORD}
```

A file number of 0 means change the default allocation given to all new files. ⍐ returns the old value of the allocation quota.

```
            ⍐[1] 120 200000         (The file allocation quota of file 120 on
      50000                          library 1 is to be raised to 200000 bytes)

            ⍐[2] 0 100000           (The default file allocation on library 2 is
      50000                          to be increased to 100000 bytes)
```

Two-argument form

This more common form of ⍐ allows file access by other users in a shared file system temporarily to be suspended. While a hold is in effect at the component or file level, the user issuing the successful hold is granted exclusive access to the held component or file to perform his file update(s). The general form of the command is:

```
            R←A ⍐ {[LIBRARY]} FILE,COMPONENT {,USER NO,PASSWORD}
```

For a component number of 0 the entire file is held. The left argument A determines the strength of the hold:

0	means release the component or file, removing a previous hold
1	means restrict write access by others
2	means restrict read and write access by others

```
      1 ⍐[1] 120 3           (Restrict write access to component 3
 1                            of file 120 on library 1)
      2 ⍐ 98 0               (Restrict read/write access to whole
 1                            of file 98 on library 0 (default))
      0 ⍐[1] 120 3           (Release component 3 of file 120
 1                            on library 1)
      0 ⍐ 98 0 1000          (Release file 98 on library 0, file
 1                            belongs to user 1000)
```

In two-argument form, ⍐ returns a 1 if the operation was successful, 0 otherwise.

Effect of Access Matrix on Hold Operation

Some file operations are not affected by the ⍐ function, and some others are blocked even to the holder. The following table illustrates:

Operation	Effect of Hold	
	File Hold	Component Hold
Read components	2	2
1⍇ 2⍇	2	0
3⍇	2	2
Insert Components	1+2	N
Append Components	1+2	0
Replace Components	1+2	1+2
Delete a File	1+2	N
Delete a Component	1+2	N
Set File Allocation	1+2	N
Rename	1+2	N
Hold/Release File	1+2	N
Hold/Release Components	1+2	1+2
6⍇ 7⍇	0	0
Read Access Matrix	2	0
Write Access Matrix	1+2	N

where:

0	means the operation is not affected by a hold
2	means that when the hold strength is 2 (read and write held), only the holder may perform the operation. For components, the block is only on the held components
1+2	means that when the hold strength is 1 or 2, only the holder may carry out the operation. Again, for components, the block is only on the held components
N	means that when the hold strength is 1 or 2, no one may carry out the operation

⍗ File drop

One-argument form

Deletes a component within a file. The right argument identifies the file or component to be dropped. A file is identified by its file number and a component is identified by the number of the file it's in, and its own number within that file.

The full form of ⍗ is ({} means optional):

```
R ← ⍗ {[LIBRARY]} FILE,COMPONENT {,USER,PASSWORD}
```

R (the result) is 1 if the operation was successful, and 0 if not.

LIBRARY number identifies the library volume which holds the file to be accessed. If you omit the library number, library 0 is assumed. If included, the library number is put in square brackets. (see also the entry on ⎕MOUNT for discussion of library numbers)

FILE number is the number of the file to be accessed.

COMPONENT number is the number identifying the component to be deleted. If 0 the last component is deleted. If the component is not specified, the number will be assumed to be 0.

USER number is the number of the owner of the file. If omitted, defaults to 1↑⎕AI, i.e. your own user number. PASSWORD is the optional number designated as a security password 0 is assumed if the password is omitted.

APLX will return a code 1 to indicate that it has successfully carried out the operation, otherwise a 0 is returned.

```
      ⍗ 2 100                (Delete component 100 in file 2)
1
      ⍗[1]4 222              (Delete component 222 of file 4 which is
1                             on library volume 1)
```

Two-argument form

Deletes an entire file from the system. The form is ({} means optional):

```
R←USER ⍗ {[LIBRARY]} FILE,0, {USER,PASSWORD}
```

where R, USER, LIBRARY, FILE, PASSWORD are as defined above.

Note: By default, a user is denied the privilege of deleting an entire file, even his own. In order to delete a file, the owner must first grant himself the deletion privilege by adjusting the Access matrix (see section on ⍃).

⎕Fxxx Component File System

As an alternative to the powerful multi-user component file system accessed using the file primitives ⍇ ⍈ ⍐ ⍗, APLX also implements a second component file system similar to that used in many other APL interpreters, with important extensions. This is based on the system functions ⎕FCREATE ⎕FTIE ⎕FREAD and so on.

⎕Fxxx files are identified by a file name, and created using ⎕FCREATE. For each APL component file, a separate operating-system file will be created. When you want to use an existing file, you first 'tie' (open) it using ⎕FTIE or ⎕FSTIE, and then you refer to the file by the tie number which you have specified or which has been automatically allocated by APLX. (This is in contrast to the ⍈ ⍇-based system, where a single 'dataspace' holds multiple APL component files, component files are always identified by number, and there is no need to 'tie' a file to use it.) Once the file has been tied, components are accessed by component number. When you have finished using a file, you must close it using ⎕FUNTIE. (They are untied automatically when the APL task ends, but they are not untied automatically when you)CLEAR the workspace or)LOAD another workspace).

Components within a file are numbered sequentially, initially from 1 to N, where N is the number of components in the file. You read components from an existing file using ⎕FREAD. You can write a component to the file using the ⎕FAPPEND and ⎕FREPLACE facilities implemented by other APL interpreters; these allow you to append to the end of the file, or to replace an existing component respectively. You can also delete components using ⎕FDROP, but only from the start or the end of the file. Components are not re-numbered, so if you drop components from the start of the file, the first component will no longer be number 1.

APLX retains upwards compatibility with this simple model, but in addition provides the more general functions ⎕FWRITE (which allows you to insert components anywhere within the range of existing components, or immediately before or after them), and ⎕FDELETE (which allows you to delete a component anywhere in the file). When you use these extensions, components are automatically re-numbered so that they always comprise sequential integers from the first component M to the last component 1+M-N, where N is the number of components in the file.

Individual components may be any valid APL data, including nested arrays and overlays created using ⎕OV (which can contain multiple functions and variables). The components keep their type and shape when stored and retrieved. When you replace a component, the new component does not have to be the same size as the original; the file system automatically expands the file if necessary to accommodate a larger component, and if possible releases space when you replace an existing component with a smaller one.

When using the file system in a multi-user or multi-tasking environment, you can optionally tie a file for exclusive use (⎕FTIE), or for shared access (⎕FSTIE). A file may be kept secure from other users by a pass number, and you can set an *access matrix* which determines what operations other users can perform. To facilitate concurrent use of shared files whilst maintaining data integrity, the file hold facility ⎕FHOLD allows you to hold one or more files temporarily for exclusive use.

Special considerations for Client-Server implementations of APLX

See `⎕FCREATE` for details on how component files can be located on either the Client or Server machine.

Mixing 32-bit and 64-bit Component Files

If you are running both 32-bit and 64-bit versions of APLX, then it is possible to share component files between the two architectures, but there are some special points you should be aware of. The rules are as follows:

- If the file has been created from a 32-bit version of APLX, then it will always remain as a 32-bit component file. It can be accessed from 64-bit APLX64 systems, but all components will be held in 32-bit form. If you write a component from APLX64, then the data is converted to 32-bit form before it is written. This means no component can be bigger than 2GB, nor have more than 2,147,483,647 elements. It also means that any 64-bit integer data will be converted to floating-point form if it contains integers of magnitude bigger than 2*31. If it contains integers of magnitude bigger than 2*53, the data conversion will involve loss of precision. The maximum size of the file is currently 2GB.
- If the file has been created from a 64-bit APLX64 interpreter, it will be a 64-bit component file. It cannot be accessed from 32-bit APLX systems. Data can be of any type or size, subject only to an overall size limit for a single component file of 1024GB.

Component File Functions

For more information, see the descriptions of the `⎕Fxxx` system functions:

⎕FAPPEND	Append component to file
⎕FCREATE	Create a new component file
⎕FCSIZE	Read component size information
⎕FDELETE	Delete component from file
⎕FDROP	Drop components from start or end of file
⎕FDUP	Duplicate component file, reclaiming wasted space
⎕FERASE	Erase component file
⎕FERROR	Return operating-system error
⎕FHOLD	Hold/Release component files for exclusive access
⎕FLIB	Return names of component files in directory
⎕FNAMES	Return names of currently-tied files
⎕FNUMS	Return tie numbers in use
⎕FRDAC	Read component-file access matrix
⎕FRDCI	Read component information
⎕FRDFI	Read file information
⎕FREAD	Read component
⎕FRENAME	Rename component file
⎕FREPLACE	Replace existing component
⎕FRESIZE	Set maximum file size
⎕FSIZE	Read file-size and component-range information
⎕FSTAC	Set component-file access matrix

⎕FSTIE	Tie file for shared use
⎕FTIE	Tie file for exclusive use
⎕FUNTIE	Untie component file(s)
⎕FWRITE	Append, replace or insert component

Section 5: Native File Functions

APLX Native File Support

APLX provides a full set of system functions which let you access the native file system on your host machine.

In many cases, the easiest way to read or write data in files is to use the ⎕IMPORT and ⎕EXPORT functions. These allow you to read or write the entire contents of a file in a single call, in a number of common formats, for example in formats which spreadsheets can access. However, for more detailed control of the contents of a file, or to access files which are too big to read into a variable in the workspace, you will need to use the native file functions described below.

See also ⎕SQL, which allows you to read and write data held in relational databases.

Native file functions using tie numbers

Most of the APLX native file functions refer to a host file through a file *tie number*, a non-zero integer value used to identify the file once it has been opened. You can specify the tie number yourself as an argument to the ⎕NTIE or ⎕NCREATE functions. Alternatively, you can provide an argument of 0 and let APLX choose a unique tie number for you (in this case it is returned as the explicit result of the function). The name of the file to tie is supplied to the ⎕NTIE or ⎕NCREATE call as a character vector and may be a file name or a full host path name. If the full path is omitted the current working directory is assumed. Case is significant in host file names under Linux or AIX, but not under Windows and MacOS.

Files may be accessed totally randomly, that is you can read and write data as an arbitrary stream of bytes anywhere in the file. The ⎕NREAD and ⎕NWRITE functions also allow you to specify an optional conversion to apply to the file data. For example you can read data as raw bytes or translate text files into the internal representation used by APLX. Unicode text is also supported. You can read numeric data as 2 or 4-byte integers, or as booleans or 8-byte floats. In addition you can specify that data is byte-swapped for transfer between machines with different byte-ordering conventions.

When you have finished using a file it must be untied using the ⎕NUNTIE function. This will close the file and release any file locks that have been set by the ⎕NLOCK function. Files are also untied automatically by a)CLEAR or an)OFF. Tied files are not affected by a)LOAD operation.

Errors may arise using the native file system for a number of reasons, for example an attempt to tie a non-existent file or to read beyond the end of a file. In the event of an error of this type, the system function will return a FILE I/O ERROR. In addition, if error trapping is not enabled, a short informative message is displayed:

```
      'TEST.DATA' ⎕NTIE 1
A file or directory in the path name does not exist.
FILE I/O ERROR
      'TEST.DATA' ⎕NTIE 1
      ^
```

The text of the specific error message is also available using the `⎕NERROR` function. This returns the error message for the last native file system function to give rise to a FILE I/O ERROR.

File size limits

In 32-bit versions of APLX, the maximum integer is 2147483647. Because file sizes and positions are expressed as integers, this effectively puts a limit of 2GB on the size of native files which you can directly access in the 32-bit versions of APLX.

In APLX64, the maximum integer is 9223372036854775807, making it possible to directly access files of up to 8,589,934,592 GB.

Special considerations for Client-Server versions of APLX

In Client-Server implementations of APLX, you can specify whether the native file access should take place on the Client or Server machine. See the description of `⎕NCREATE` for more information.

Native File System Functions

For details on using the native-file functions, see the following entries in the section on *System Functions and Variables*:

`⎕NAPPEND`	Append data to file
`⎕NCREATE`	Create file
`⎕NERASE`	Erase file
`⎕NERROR`	Get last file error
`⎕NLOCK`	Lock/Unlock file
`⎕NNAMES`	List names of tied files
`⎕NNUMS`	List tie numbers
`⎕NREAD`	Read data from file
`⎕NRENAME`	Rename file
`⎕NREPLACE`	Replace data in file
`⎕NRESIZE`	Resize file
`⎕NSIZE`	Get size of file
`⎕NTIE`	Open file
`⎕NUNTIE`	Close file
`⎕NWRITE`	Write data to file

Section 6: System Commands

)CLASSES (first (last))

Lists the names of the user-defined classes in the current workspace. If the command is followed by a character or group of characters, the list gives the names of all functions beginning with that character or group of characters onwards (the parameter `first`, used on its own). A second character or group of characters after the command (the parameter `last`) is used to end the list of names. Names are shown in alphabetic order, fully sorted.

```
      )CLASSES
Point   Polygon Rectangle  Sphere    Triangle
      )CLASSES S
Sphere  Triangle
      )CLASSES Pol R
Polygon Rectangle
```

See also `⎕CLASSES`, which returns a list of user-defined classes and references to external classes.

)CLEAR (wssize)

Clears the current workspace. All objects in the workspace are erased, most system variables revert to their default settings, and the name of the workspace reverts to CLEAR WS.

```
      )CLEAR
CLEAR WS
```

`)CLEAR` also optionally changes the workspace size. You can specify a parameter which is the workspace size you want. It must be an integer, and can be specified in bytes, or followed by K or KB for kilobytes, M or MB for megabytes, or G or GB for gigabytes. The valid range is 50 KB to 2 GB (for 32-bit versions of APLX), or up to a theoretical maximum of 8580934592 GB for APLX64.

Depending on the operating system and its configuration, and the amount of memory already in use by APLX tasks, you are likely to be limited in the maximum size of workspace which you can allocate. Thus you may not get the full size requested. In practice also, if the workspace you allocate is larger than the physical RAM in your system, then performance may become be very poor.

For example:

```
      )CLEAR 1000000
WS Size = 976KB
CLEAR WS

      )CLEAR 1024KB
WS Size = 1.0MB
CLEAR WS
```

```
      )CLEAR 100M
WS Size = 100MB
CLEAR WS

      )CLEAR 2G
WS Size = 484MB
CLEAR WS
```

Note that in the last example, the user requested 2GB but the operating system allocated only 484MB.

Example valid only in APLX64:

```
      )CLEAR 16G
WS Size = 16GB
CLEAR WS
```

)CONTINUE

The `)CONTINUE` command is implementation dependent, but when implemented this command will change the name of your active workspace to CONTINUE and `)SAVE` it, then leave APL. On re-entering APL, the CONTINUE workspace is automatically loaded. Use of this command is not recommended, as it can quite easily lead to confusion on a multi-user system.

```
      )CONTINUE
11.15.54 04/29/89 CONTINUE
```

)COPY (lib) name (:pass) (name(s)

Copies into the currently-active workspace named items from a saved workspace. For example, to copy functions FRED and JOE from a workspace called MYWS in library 3, you would enter:

```
      )COPY 3 MYWS FRED JOE
SAVED  1991-06-13 23.24.06
```

If just the workspace name is used, the entire contents are copied:

```
      )COPY MYWS
 SAVED  1991-06-13 23.28.17
```

If the name of an object to be copied matches the name of an object already in the active workspace, the copy will overwrite the object already in the workspace. See also `)PCOPY` Protected copy, `)SCOPY` Silent copy, `)SPCOPY` Silent protected copy). If a WORKSPACE FULL or SYMBOL TABLE FULL error is encountered, the active workspace is left unchanged. You should note that the `)COPY` operation works by temporarily `)SAVE`ing the active workspace in the logical unit from which objects are being copied (or in a disc defined for temporary objects), extracting the required objects from the workspace identified in the `)COPY` command and then merging the active workspace and the objects to be copied. It is thus possible to see a DISC FULL message during a copy operation.

Copying classes and objects

`)COPY` can be used to copy classes and objects from a saved workspace. However, some special considerations arise:

- If a class is copied, and in the original workspace it had a parent, then the `)COPY` will fail unless a parent class of the same name exists in the destination workspace, or is copied at the same time. APLX will report an error "Class XXX not copied, missing parent class YYY"
- If a variable containing an object reference is copied, APLX will attempt to copy both the object reference, and the object itself together with its saved property values. However, a class of the same name as that of the original object must exist in the destination workspace (or be copied in at the same time). If this is not the case, the `)COPY` will proceed, but the object reference will be set to refer to the Null object. APLX will print a warning "At least one object reference set to NULL (class does not exist)".
- When an object instance is copied in, it is possible for data to be lost. This will happen if the original version of the object (in the saved workspace) had a non-default property which is no longer valid in the current destination workspace (because the version of the class is different). If this happens, the `)COPY` will proceed, but APLX will print a warning "At least one object property not copied (not valid for class)".

If you `)COPY` a list of classes and/or objects, or an entire workspace, APLX will first copy any top-level classes (classes with no parent), then classes of the first generation (children of top-level classes), and so on. It will then `)COPY` object instances and other items. This guarantees that no object properties or class hierarchy information is unnecessarily lost.

Library specification and path names

There are two different ways in which you can specify where APLX should look for the saved workspace:

- You can specify the workspace name as just the base name of the workspace, for example `MYWS` or `Budget03`, optionally preceded by a library number. In this case, APLX appends any default file-extension to the name (`.aws` for Windows, AIX or Linux), and searches in the directory corresponding to the specified library number. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the `⎕MOUNT` system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.
- You can specify a full operating-system path name, including directory separation characters, such as `/usr/workspaces/Budget03.aws` *(Linux)*, `C:\workspaces\Budget03.aws` *(Windows)*, or `MacHD::workspaces:Budget03` *(MacOS)*. APLX uses the path name exactly as supplied, so under Linux, Windows and AIX you usually need to provide the `.aws` file extension.

See the description of the `)LOAD` system command for more detail on libraries and path names.

Indirect copy

If one or more of the names following the `)COPY` command is enclosed in parentheses and is the name of a variable in the workspace to be copied from which is a simple character scalar, vector or matrix, then the contents of the variable are interpreted as the name or names of objects to be copied. The alternative forms of `)COPY` (i.e. `)PCOPY`, `)SCOPY`, and `)SPCOPY`) will also accept name arrays as part of the name list of the command.

```
      (THIS THAT THE_OTHER)←⊂'DATA'
      NAMES←⎕BOX 'THIS THAT THE_OTHER'
      )WSID TEST
WAS TEST
      )VARS
NAMES   THAT    THE_OTHER      THIS
      )SAVE
1991-06-13 19.14.26 TEST
      )CLEAR
CLEAR WS
      )COPY TEST (NAMES)
SAVED  1991-06-13 19.14.26
      )VARS
THAT    THE_OTHER      THIS
```

`)CS` (number)

`)CS` followed by an integer from 0 to 7 establishes APL.68000 Level I or Level II mode. The Compatibility Setting is a workspace parameter and defaults to 0. See `⎕CS` for details of the parameters. For example,

```
      )CS 0
      1 2 3[2]
```

generates a RANK ERROR whilst Compatibility Setting 1 (APL.68000 Level I mode) will return the result 2 for the same expression. The Compatibility Setting is a workspace parameter and defaults to 0

`)DIGITS` number

Followed by a whole number between 1 and 15, this command sets the maximum number of significant digits displayed after the decimal point in results. On its own (without any following number) it asks the current setting of DIGITS. The default setting is 10. (See also the system variable `⎕PP, Print precision`)

```
      )DIGITS
IS 10
      )DIGITS 8
WAS 10
```

```
      )DIGITS
IS 8
            ⎕PP
8
```

)DISPLAY name

Displays the structure of a variable, in the same form as that returned by the `⎕DISPLAY` system function.

```
      DATA←(2 2⍴⍳4) 'HELLO'
      )DISPLAY DATA
┌→───────────────┐
│ ┌→──┐  ┌→────┐ │
│ ↓1 2│  │HELLO│ │
│ │3 4│  └─────┘ │
│ └~──┘          │
└∊───────────────┘

      X←⊂(2 3⍴⍳6) (1 1⍴1) (1 2 2⍴⍳4)
      )DISPLAY X
┌─────────────────────────┐
│ ┌→────────────────────┐ │
│ │ ┌→────┐ ┌→┐ ┌┌→──┐ │ │
│ │ ↓1 2 3│ ↓1│ ↓↓1 2│ │ │
│ │ │4 5 6│ └~┘ ││3 4│ │ │
│ │ └~────┘     └└~──┘ │ │
│ └∊────────────────────┘ │
└∊────────────────────────┘
```

See the description of `⎕DISPLAY` for details of the display format.

)DROP (lib) name (:pass)

Drops (erases) a named workspace from disk. If the saved workspace has been saved with a password (see `)WSID` and `)SAVE`), then the `)DROP` command must include the correct password. For example:

```
      )DROP MYWS
      )DROP 1 MYWS
      )DROP MYWS:SECRET
      )DROP /usr/workspaces/MYWS.aws
```

Library specification and path names

There are two different ways in which you can specify where APLX should look for the workspace to be erased:

- You can specify the workspace name as just the base name of the workspace, for example `MYWS` or `Budget03`, optionally preceded by a library number. In this case, APLX appends any default file-extension to the name (`.aws` for Windows, AIX or Linux), and searches in the directory corresponding to the specified library number. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the `⎕MOUNT` system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.

- You can specify a full operating-system path name, including directory separation characters, such as `/usr/workspaces/Budget03.aws` *(Linux)*, `C:\workspaces\Budget03.aws` *(Windows)*, or `MacHD::workspaces:Budget03` *(MacOS)*. APLX uses the path name exactly as supplied, so under Linux, Windows and AIX you usually need to provide the `.aws` file extension. (Note: In Client-Server implementations of APLX, you can specify that the path refers to the Client or Server machine by preceding the file name with an Up Arrow ↑ or Down Arrow ↓).

See the description of the `)LOAD` system command for more detail on libraries and path names.

`)EDIT` (type) name

APLX includes a 'full screen' editor for functions, operators, variables and classes. This editor may be accessed via:

```
            )EDIT
```

The editor is entered thus:

```
      )EDIT NAME          ⍝ EDIT EXISTING OBJECT <NAME>
                          ⍝ EDIT NEW FUNCTION OR OPERATOR <NAME>
      )EDIT 0 NAME        ⍝ EDIT NEW OR EXISTING FUNCTION OR OPERATOR <NAME>
      )EDIT 1 NAME        ⍝ EDIT NEW OR EXISTING VAR <NAME>
      )EDIT 2 NAME        ⍝ EDIT NEW OR EXISTING CLASS <NAME>
```

`)EDIT` can also be used to edit individual class members, rather than the whole class. In this case, you specify the fully-qualified name in the form ClassName.MemberName.

See also `⎕EDIT`

)ERASE name(s)

Erases named global variables, functions, operators and classes from the active workspace. The command is followed by the name, or names, of the objects to be erased. If an item cannot be erased, a message to that effect is displayed. Local variables are not erased - use ⎕EX if you wish to do this.

```
      )ERASE ∆CC NEMO
NOT FOUND: NEMO
```

Indirect erase

If one or more of the names following the)ERASE command is enclosed in parentheses and is the name of a variable which is a simple character scalar, vector or matrix, then the contents of the variable are interpreted as the name or names of items to be erased.

```
      )VARS
A       B       C       DATA    MAT
      )ERASE A C
      )VARS
B       DATA    MAT
      NAMES←⎕BOX 'B DATA'
      NAMES
B
DATA
      )ERASE (NAMES)      (rows of NAMES interpreted as object to erase)
      )VARS
MAT     NAMES
      )ERASE NAMES
      )VARS
MAT
      DATA←'A_NAME'
      )VARS
      DATA    MAT
      )ERASE (DATA)       (same error message as direct erase)
NOT FOUND: A_NAME
```

Erasing individual class members

)ERASE can be used to erase a member (a method of property) from a class definition, using dot notation in the form ClassName.MemberName to specify which member should be deleted. The change will immediately be reflected in any existing instances of the class:

```
      PT←⎕NEW COLOR_POINT
      PT.⎕NL 2     ⍝ List properties of object PT
COLOR
X
Y
Z
      )ERASE COLOR_POINT.Z
      PT.⎕NL 2     ⍝ Object PT now has one less property
COLOR
X
Y
```

Erasing whole classes

)ERASE can also be used to erase a class definition (and all the methods and properties defined in it). Any instances of the class will become instances of the erased class's parent, if there is one, or of the NULL class, if the erased class did not have a parent. Similarly, any classes which inherited from the erased class will be re-parented so that they now inherit from the erased class's parent.

In this example, class POINT3D inherits from COLOR_POINT which in turn inherits from POINT. PT is an instance of COLOR_POINT:

```
      )CLASSES
COLOR_POINT    POINT   POINT3D
      ⎕CLASS POINT3D
{POINT3D} {COLOR_POINT} {POINT}
      PT←⎕NEW COLOR_POINT
      PT.⎕CLASSNAME
COLOR_POINT
```

If we erase the class COLOR_POINT, its child class POINT3D is re-parented. The instance PT becomes an instance of the original parent:

```
      )ERASE COLOR_POINT
      ⎕CLASS POINT3D
{POINT3D} {POINT}
      PT.⎕CLASSNAME
POINT
```

If we now erase the class POINT, POINT3D will now have no parent, and the instance PT becomes an instance of the NULL class:

```
      )ERASE POINT
      PT.⎕CLASSNAME
NULL
      ⎕CLASS POINT3D
{POINT3D}
```

)FNS (first (last))

Lists the names of all the functions in the current workspace. If the command is followed by a character or group of characters, the list gives the names of all functions beginning with that character or group of characters onwards (the parameter first, used on its own). A second character or group of characters after the command (the parameter last) is used to end the list of names. Names are shown in alphabetic order, fully sorted.

```
      )FNS
AFE     CONTINUE        HELP    INFO    SCLOSE  SLOG    SMOUNT  SOPEN
SREAD   SRET    SUNMOUNT        TRANSLATE       ∆MERR   ∆SNAME
      )FNS T
TRANSLATE       ∆MERR   ∆SNAME
      )FNS SM
SMOUNT   SOPEN    SREAD    SRET     SUNMOUNT          TRANSLATE
∆MERR  ∆SNAME
      )FNS  SM  T
SMOUNT   SOPEN    SREAD    SRET     SUNMOUNT          TRANSLATE
```

)GROUP name(s)

Gathers functions and variables into a group. The first name given will be the name of a group, and the subsequent names are those of variables, functions and operators to be placed in the group. If only the group name is supplied, the effect is to disband that group.

Groups can be used with the commands)ERASE and)COPY to deal with a set of objects in a single operation. However, they are generally considered obsolete, because APLX support 'indirect')ERASE and)COPY, where the list of names is contained in an APL variable.

```
      )FNS
COVARIANCE      MEAN    MEDIAN  MODE    STANDARD_DEV    VARIANCE
      )GROUP AVERAGES MEAN MEDIAN MODE
      )GRPS
AVERAGES
      )GRP AVERAGES
MEAN    MEDIAN  MODE
      )ERASE AVERAGES
      )FNS
COVARIANCE      STANDARD_DEV    VARIANCE
```

)GRP name(s)

Lists the names of objects in group *name*.

)GRPS (first (last))

Lists the names of all the groups in the current workspace. If the command is followed by a character or group of characters, the list gives the names of all groups beginning with that character or group of characters onwards (the parameter first, used on its own). A second character or group of characters after the command (the parameter last) is used to end the list of names. Names are shown in alphabetic order, fully sorted.

```
            )GROUP FILEFNS SOPEN SREAD SRET
            )GRP FILEFNS
      SOPEN    SREAD    SRET
            )GRPS
      FILEFNS
```

)HOST (command)

The)HOST command allows the user to issue a command directly to the host environment and display the result without leaving the APL workspace.

When used without a command, it displays the operating system under which you are working:

```
      )HOST
IS AIX
```

When used with a command specified, the command is passed to the operating system and executed. For example:

```
      )HOST pwd
/usr/apl/aplx
```

Control-C or Break in the Interrupt menu will end the command and return to APL. Otherwise, control returns to APL when the command terminates, or after a timeout value of 10 seconds. (For finer control of the timeout, see the ⎕HOST system function)

The)HOST command is highly implementation-specific, and some operating system commands may not be allowed. Points to note are:

- **AIX** and **Linux:** Interactive commands can be executed if required.
- **MacOS:**)HOST is not implemented under MacOS 8 and 9 except to report the OS name. Under MacOS X,)HOST is implemented. It invokes the BSD terminal shell to run Unix-style programs such as 'ls' or shell scripts. However, interactive programs are not supported.
- **Windows:**)HOST is implemented under Windows, although interactive programs are not supported. Note that, under Windows, many common commands are 'built-in' to the command-line shell, rather than being separate executable programs. Under Windows NT, 2000, XP and Vista, you can run these using the 'CMD' program with the '/C' option. (Under Windows 95, 98 and ME, use 'COMMAND.COM /C'). For example:

```
      )HOST CMD /C DIR C:\PROG*.*
 Volume in drive C has no label.
 Volume Serial Number is 07D0-0B11

 Directory of C:\

17/11/2000  21:05       <DIR>          Program Files
               0 File(s)              0 bytes
               1 Dir(s)  14,522,580,992 bytes free
```

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine. The two parts of the application communicate via a TCP/IP network. Typically, the Client will be the APLX front-end built as a 32-bit Windows application running on a desktop PC, and the Server will be a 64-bit APLX64 interpreter running on a 64-bit Linux or Windows server.

In such systems, `)HOST` allows you to specify whether the command should be executed on the Client or the Server machine. You do this by preceding the command string with either an Up Arrow ↑ to indicate that the command should be executed on the Client, or a Down Arrow ↓ to indicate that it should run on the Server. If you do not specify, the default is that the call should take place on the Client.

In this example, the Client is running under Windows 2000, and the Server under Linux x86_64:

```
      )HOST ↑cmd /c ver
Microsoft Windows 2000 [Version 5.00.2195]

      )HOST ↓uname -nsp
Linux nx6125 x86_64
```

`)IN` (lib) filename (name(s))

Imports a Transfer File into the active workspace. Transfer Files are text versions of APL objects that are created by the `)OUT` command, or equivalent APL functions. They may be created by APLX or by another APL interpreter such as IBM's APL2. The Transfer File format is fully explained under the `)OUT` and `⎕TF` commands. The default file extension for Transfer Files is `.atf`.

You can import either the whole Transfer File, or just selected items as specified by the 'names' parameter of the command. For example:

```
      )FNS
      )IN DISPLAY            (Read the whole Transfer File)
      )FNS
ABSTRACT        DISPLAY DESCRIBE
      )CLEAR
CLEAR WS
      )IN 3 DISPLAY DESCRIBE  (Read specified objects from Transfer
                               File in Library 3)
      )FNS
DESCRIBE
      )IN 2 DODO             (Transfer File not found)
WS NOT FOUND
```

Library specification and path names

There are two different ways in which you can specify where APLX should look for the Transfer File:

- You can specify the just the base name of the file, for example `MYWS` or `Budget03`, optionally preceded by a library number. In this case, APLX appends the default file-extension `.atf` to the name, and searches for the file in the directory corresponding to the specified library number. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the `⎕MOUNT` system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.

- You can specify a full operating-system path name, including directory separation characters, such as `/usr/transfer/Budget03.atf` *(Linux)*, `C:\transfer\Budget03.atf` *(Windows)*, or `MacHD::transfer:Budget03.atf` *(MacOS)*. APLX uses the path name exactly as supplied, so you usually need to provide the `.atf` file extension explicitly. (Note: In Client-Server implementations of APLX, you can specify that the path refers to the Client or Server machine by preceding the file name with an Up Arrow ↑ or Down Arrow ↓).

See the description of the `)LOAD` system command for more detail on libraries and path names.

`)LIB` (lib)

Lists the names of the workspaces in the library or explicit path specified (or Library 0 by default). If the command (and library number, if used) are followed by a letter, only workspaces beginning with that letter are listed.

```
      )LIB
COAL        CONSOLE       FORMAT      MIRSEQ    NEWGRAF
PSYS        SYSFNS
      )LIB C
COAL        CONSOLE
      )LIB 3
CALCULATE   DISPLAY
      )LIB C:\workspaces\budgets
Budget02    Budget03    BudgetDraft
```

Note that only APLX workspaces are shown in the list, not other files. Under Windows, Linux and AIX, these will have the file extension `.aws` (any workspaces you save using a full pathname without this extension will not be listed). The file extension is not shown in the `)LIB` display.

Library specification and path names

There are two different ways in which you can specify the directory where APLX should look for the workspaces:

- You can specify a numeric library number, as shown in the first three examples above. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the `⎕MOUNT` system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.

- As shown in the last example above, you can specify a full operating-system directory name, including directory separation characters, such as `/usr/workspaces/` *(Linux)*, C:\workspaces\ *(Windows)*, or MacHD::workspaces:Budget03 *(MacOS)*. APLX uses the directory name exactly as supplied.

Note also that, in Client-Server implementations of APLX, you can precede the path name with an Up or Down arrow to specify on which machine the directory should be searched.

See the description of the)LOAD system command for more detail on libraries and path names.

)LOAD (lib) name (:pass)

Loads a named workspace. The workspace is loaded into memory and overwrites any workspace already associated with the current APL task. If the workspace has a password, it must be given. Otherwise the message WS LOCKED appears and the workspace is not loaded.

```
      )LOAD MYWS
SAVED   2002-09-11 10.32.16
```

Library specification and path names

There are two different ways in which you can specify where APLX should look for the saved workspace:

- You can specify the workspace name as just the base name of the workspace, for example MYWS or Budget03, optionally preceded by a library number. In this case, APLX appends any default file-extension to the name (.aws for Windows, AIX or Linux), and searches in the directory corresponding to the specified library number. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the ⎕MOUNT system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.
- You can specify a full operating-system path name, including directory separation characters, such as /usr/workspaces/Budget03.aws *(Linux)*, C:\workspaces\Budget03.aws *(Windows)*, or MacHD::workspaces:Budget03 *(MacOS)*. APLX uses the path name exactly as supplied, so under Linux, Windows and AIX you usually need to provide the .aws file extension.

How APLX interprets the name and library specification

The rules which APLX applies when interpreting a name and/or library specification in a system command are as follows:

1. If the name begins with a number followed by one or more spaces, the number is taken to be a library number and the text after the space(s) is the file name. The directory in which the file is located is taken from the appropriate row of the ⎕MOUNT table, or the installed workspace directory for library 10. If the directory is blank, the user's home directory is assumed. Under AIX, Linux or Windows, the file extension (.aws) is automatically appended to the supplied name.

2. If the name does not begin with a number followed by spaces, APLX looks at the name to see if it contains at least one directory separator character (/ under Linux, \ under Windows, / or : under MacOS). If it does not, then it is treated as a simple filename in Library 0, and (under AIX, Linux or Windows) the file extension is automatically appended to the supplied name.
3. If the name does contain a directory-separator character, it is assumed to be a full pathname, **including the file extension if any**. APLX uses the name exactly as supplied.

In each case, the name is normally assumed to end at the first blank character. If you want to include blanks in the name, you can enclose the whole file name in single quotes.

If you use full pathnames under AIX, Windows or Linux, you should normally supply the `.aws` file extension when saving workspaces, otherwise the workspace will not show up in the `)LIB` listing. This is not true under MacOS, which keeps a file type separate from any file extension.

Some system commands (including `)LOAD`) can take an optional colon and password. Under MacOS, this might be confused with a full pathname, so you must include a space before the colon to terminate the name.

Examples *(Windows):*

Suppose the first three rows of your `⎕MOUNT` table are set up as follows:

```
      3 40↑⎕MOUNT ''
c:\temp

G:\apl\historic\aplx
```

Note that the second row, library 1, is blank, so will correspond to the user's home directory. This might be something like: `C:\Documents and Settings\Jim\My Documents`.

Library of directory 0:

```
      )LIB 0
IF          JIM         PICTUREDEMO PLUSFNS     SC          TESTDISPLAY
```

Library of directory 0, implicit library number:

```
      )LIB
IF          JIM         PICTUREDEMO PLUSFNS     SC          TESTDISPLAY
```

Library of the same directory, specifying a full library path:

```
      )LIB c:\temp
IF          JIM         PICTUREDEMO PLUSFNS     SC          TESTDISPLAY
```

Library of directory 1. Because `⎕MOUNT` *table entry is blank, this is the user's home directory:*

```
      )LIB 1
ANOTHER     BERT        FRED
```

Load a workspace from library 0 (the 0 could be omitted). Full path is `c:\temp\PICTUREDEMO.aws`

```
      )LOAD 0 PICTUREDEMO
SAVED  2002-07-05 15.51.31
```

Load a workspace from library 1. Full path is `ANOTHER.aws` *in user's home directory:*

```
      )LOAD 1 ANOTHER
SAVED  2003-11-18 10.49.51
```

Load a workspace using full explicit path, ***including file extension*** *(Note that Windows file names are not case-sensitive):*

```
      )LOAD C:\TEMP\PICTUREDEMO.AWS
SAVED  2002-07-05 15.51.31
      )WSID
C:\TEMP\PICTUREDEMO.AWS
```

Save under a name containing spaces - we need to enclose the name in quotes:

```
      )SAVE 'A nice name with spaces'
2003-12-10 13.44.16
      )LIB
A nice name with spaces IF          JIM        PICTUREDEMO PLUSFNS
SC          TESTDISPLAY
```

Re-load using a full pathname - again we need to enclose the name in quotes, and supply the file extension because we are using a full path:

```
      )LOAD 'c:\temp\A nice name with spaces.aws'
SAVED  2003-12-10 13.44.16
```

Paths in MacOS X

Under MacOS X, you can enter file paths either using in traditional Macintosh style, using colon as a directory separator (`MacHD::workspaces:Budget03`), or in a Unix style, with slash as the separator (`/Volumes/MacHD/workspaces/Budget03`).

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine. The two parts of the application communicate via a TCP/IP network. Typically, the Client will be the APLX front-end built as a 32-bit Windows application running on a desktop PC, and the Server will be a 64-bit APLX64 interpreter running on a 64-bit Linux or Windows server.

In such systems, you can specify whether the file should be accessed on the Client or the Server machine. You do this by preceding the file name with either an Up Arrow ↑ to indicate that the file should be accessed on the Client, or a Down Arrow ↓ to indicate that it should be accessed on the Server. If you do not specify, the default is that the access takes place on the Client. This is true either if you specify the full path name in the system command, or via the `⎕MOUNT` table.

In this example, we load a workspace from the Client machine (under Windows), and save it on the Server machine (in this case, a Linux system):

```
      )LOAD ↑C:\workspaces\mailfilter.aws
SAVED  2006-08-15 8.50.04
      )SAVE ↓/usr/local/wsrelease/mailfilter.aws
2006-08-24 15.50.10
```

)NMS (first (last))

Lists the names of all the global symbols in the current workspace. If the command is followed by a character or group of characters, the list gives the names of all symbols beginning with that character or group of characters onwards (the parameter `first`, used on its own). A second character or group of characters after the command (the parameter `last`) is used to end the list of names. Names are shown in alphabetic order, fully sorted.

Each name is followed by a dot and a number which indicates the type of symbol:

Code	*Type*
2	Variable
3	Function
4	Operator
9	Class

```
      )VARS
A       B       C       D
      )FNS
FRED    JOE
      )CLASSES
Stats
      )NMS
A.2     B.2     C.2     D.2     FRED.3  JOE.3   Stats.9
      )NMS C
C.2     D.2     FRED.3  JOE.3   Stats.9
      )NMS C FO
C.2     D.2
```

)OPS (first (last))

Lists the names of all the defined operators in the current workspace. If the command is followed by a character or group of characters, the list gives the names of all operators beginning with that character or group of characters onwards (the parameter `first`, used on its own). A second character or group of characters after the command (the parameter `last`) is used to end the list of names. Names are shown in alphabetic order, fully sorted.

```
      )OPS
A       B       C       D
      )OPS C
C       D
      )OPS B Z
B       C       D
```

)OFF

Ends an APL session and returns the user to the operating system.

)ORIGIN (number)

Followed by 0 or 1 it sets the index origin to 0 or 1. The index origin affects the origin (or base of the range) used for generating random numbers and indexes.

The index origin is normally set to 1.

On its own, without any following number, this command displays the current value of the index origin. (See also the system variable, `⎕IO`, for a full discussion of Index Origin.)

```
      )ORIGIN
IS 1
      )ORIGIN 0
WAS 1
      ⎕IO
0
```

`)OUT` (lib) filename (name(s))

Creates a Transfer File in the library specified. If no individual objects are named then the Transfer File will include all functions, operators and variables in the workspace as well as `⎕CT`, `⎕FC`, `⎕IO`, `⎕LX`, `⎕PP`, `⎕PR` and `⎕RL`. Alternatively, specified objects may be written to the Transfer File together with an optional selection of system variables.

```
      )WSID                    (Sample workspace)
TEST
      )VARS
A       B       C       D
      )FNS
FRED    JOE
      )OUT TESTWS              (Export everything, including standard
                                system variables)
      )OUT PARTWS A C FRED ⎕PP ⎕PW
                               (Partial export only)
```

The Transfer File is written in a format which can be read using the system command `)IN` by APLX or by another APL interpreter such as IBM's APL2. Before it is written, the data is translated to the form similar to that used by APL2/PC. Not all APLX characters can be represented in this form, and variables containing control codes should not be transferred.

Library specification and path names

There are two different ways in which you can specify where APLX should save the transfer file:

- You can specify the workspace name as just the base name of the file, for example `MYWS` or `Budget03`, optionally preceded by a library number. In this case, APLX appends the default file-extension `.atf` to the name, and saves the file in the directory corresponding to the specified library number. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the `⎕MOUNT` system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.

- You can specify a full operating-system path name, including directory separation characters, such as `/usr/transfer/Budget03.atf` *(Linux)*, `C:\transfer\Budget03.atf` *(Windows)*, or `MacHD::transfer:Budget03.atf` *(MacOS)*. APLX uses the path name exactly as supplied, so you usually need to provide the `.atf` file extension explicitly. (Note: In Client-Server implementations of APLX, you can specify that the path refers to the Client or Server machine by preceding the file name with an Up Arrow `↑` or Down Arrow `↓`).

See the description of the `)LOAD` system command for more detail on libraries and path names.

Transfer file format

The Transfer File format is based on an ancient layout known as punched card layout and is a series of length 80 character records. Each record is delimited by an 'end of record' character (Hex F8) and a Carriage Return - Line Feed pair - making 83 characters in all. The first character of each record is reserved for format information, which will be:

```
       Character                      Meaning

          *                           Timestamp for the function or operator
          X                           Last record for the object (or only record)
              (blank)                 All other records
```

Remaining columns of the record are occupied by the Transfer Form of the objects selected. The Transfer Form is described under `⎕TF` Thus for the sample workspace shown above:

```
      XN⎕PP 0 10
X
      XN⎕IO 0 1
X
      XN⎕CT 0 1E¯13
X
      XC⎕FC 1 6 .,*0-
X
      XN⎕RL 0 16807
X
      XC⎕PR 1 1
X
      XC⎕LX 1 0
X
      XNA 0 1
X
      XNB 0 2
X
      XNC 0 3
X
      XND 0 4
X
      XFFRED ⎕FX 'FRED' '2'
X
      XFJOE ⎕FX 'JOE' '4' 'L:''SAMPLE FN''' '80'
X
```

and for the partial export:

```
      XNA 0 1
X
      XNC 0 3
X
      XFFRED ⎕FX 'FRED' '2'
X
      XN⎕PP 0 10
X
      XN⎕PW 0 80
X
```

)PCOPY (lib) name (:pass) name(s)

Copies into memory named objects from a named workspace without overwriting existing objects.

```
      )PCOPY MYWS FRED JOE
```

If just the workspace name is used, the entire contents are copied:

```
      )PCOPY MYWS
```

If the name of an object to be copied matches the name of an object already in the workspace in memory, the copy does not take place and a message is displayed. (See also)COPY).

```
      )PCOPY MYWS NAME
SAVED   1.21.33 05/29/89
NOT COPIED: NAME
```

)REPARENT class parent

)REPARENT allows you to change the parent of an internal (user-defined) class.

The first argument is the name of the class which you want to re-parent. The second is the name of the class which will be the new parent.

Other than the restriction that the new parent class must not be descended from the class you are modifying, there is nothing to prevent you from re-parenting a class arbitrarily. However, the main use for)REPARENT is for inserting an extra level into the class hierarchy. For example, if you have a class Car which inherits from Vehicle, you might want to create a new class MotorVehicle which inherits from Vehicle, and re-parent Car so that it now inherits from MotorVehicle:

```
      )CLASSES
Car     MotorVehicle    Vehicle
      ⎕CLASS Car
{Car} {Vehicle}
      ⎕CLASS MotorVehicle
{MotorVehicle} {Vehicle}
      )REPARENT Car MotorVehicle
      ⎕CLASS Car
{Car} {MotorVehicle} {Vehicle}
```

Any existing instances of the class will be unaffected except that any properties which are not valid in the new version of the class (because they were inherited from the old parent but are not inherited from the new parent) will be lost.

You can also re-parent a class using the `⎕REPARENT` system function, or by using the Class Editor (select Reparent Class.. from the File menu).

`)RESET` (number)

Clears all of the State Indicator (as `)SICLEAR`) or the appropriate number of entries from the State Indicator. `⎕ERM`, `⎕EM` and `⎕ET` are adjusted appropriately. If the State Indicator is cleared to an entry that was pendent, `⎕ERM`, `⎕EM` are set to be empty and the error type is set to an interrupt (`⎕ET` is 1 1).

See also the `→` (branch) symbol which, when used on its own, clears the top function marked with an asterisk, and its calling function(s), from the state indicator.

`)SAVE` (lib) (name (:pass))

Saves the current workspace to disk. It is given the same name as the workspace in memory, unless a different name is specified after the command.

If a password is specified, the workspace can be subsequently accessed only if the password is used.

```
      )SAVE
2002-09-11 10.32.16 MYWS
      )SAVE 3 MYWS:SECRET
2002-09-11 10.35.25 MYWS
      )SAVE C:\Workspaces\Budget.aws
2002-09-11 10.37.25 C:\Workspaces\Budget.aws
      )WSID
C:\Workspaces\Budget.aws
```

When attempting to `)SAVE` a workspace with the same name as a workspace that has already been saved to disk with a password, the operation will only succeed if the password of the active workspace matches that of the workspace on disk. In order to change the password of a workspace that is already saved to disc, you must first `)DROP` the copy on disk and then `)SAVE` with the new password.

Library specification and path names

There are two different ways in which you can specify where APLX should save the workspace:

- You can specify the workspace name as just the base name of the workspace, for example `MYWS` or `Budget03`, optionally preceded by a library number. In this case, APLX appends any default file-extension to the name (`.aws` for Windows, AIX or Linux), and saves the workspace in the directory corresponding to the specified library number. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the `⎕MOUNT` system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.

- You can specify a full operating-system path name, including directory separation characters, such as `/usr/workspaces/Budget03.aws` *(Linux)*, `C:\workspaces\Budget03.aws` *(Windows)*, or `MacHD::workspaces:Budget03` *(MacOS)*. APLX uses the path name exactly as supplied, so under Linux, Windows and AIX you usually need to provide the `.aws` file extension.

See the description of the `)LOAD` system command for more detail on libraries and path names.

`)SCOPY` (lib) name (:pass) (name(s))

Silent copy. The same as `)COPY` except that no message is displayed on completion, unless there is an error.

`)SDROP` (lib) name (:pass)

Silent drop. The same as `)DROP` except that no message is displayed on completion, unless there is an error.

`)SI` (number)

Displays the contents of the State Indicator. This contains the names of all halted user functions and operators. If no number is specified in the command, each line of the State Indicator is shown. If a number is specified, that many lines of the State Indicator will be displayed. (See also the system function, `⎕SI`). Certain lines will have a `*` displayed against them, these are suspended functions (or operators) which have stopped execution because of an error (or an interruption).

If a function or operator which is on the State Indicator is altered or replaced by the `)COPY` command such that the SI DAMAGE error report is given, then that entry on the State Indicator will have its line number altered to `¯1`.

```
      ∇TOP
[1]   'TOP LEVEL'
[2]   MIDDLE
      ∇

      ∇MIDDLE
[1]   'MIDDLE LEVEL'
[2]   TEST
      ∇
```

```
          ∇TEST;A;B;C
     [1]    A←⍳10
     [2]    B←⎕A
     [3]    C←A×B
            ∇

            TOP
     TOP LEVEL
     MIDDLE LEVEL
     DOMAIN ERROR
     TEST[3] C←A×B
                 ^
            )SI
     TEST[3] *
     MIDDLE[2]
     TOP[2]
            )SI 1
     TEST[3] *
            )SI 2
     TEST[3] *
     MIDDLE[2]
```

)SIC (number)

Clears all of the State Indicator (as)RESET) or the appropriate number of entries from the State Indicator. ⎕ERM, ⎕EM and ⎕ET are adjusted appropriately. If the State Indicator is cleared to an entry that was pendent, ⎕ERM, ⎕EM are set to be empty and the error type is set to an interrupt (⎕ET is 1 1).

See also the → (branch) symbol which, when used on its own, clears the top function marked with an asterisk, and its calling function(s), from the state indicator.

)SICL (number)

Clears all of the State Indicator (as)RESET) or the appropriate number of entries from the State Indicator. ⎕ERM, ⎕EM and ⎕ET are adjusted appropriately. If the State Indicator is cleared to an entry that was pendent, ⎕ERM, ⎕EM are set to be empty and the error type is set to an interrupt (⎕ET is 1 1).

See also the → (branch) symbol which, when used on its own, clears the top function marked with an asterisk, and its calling function(s), from the state indicator.

`)SINL`

Lists the names of all halted user functions and associated local variables.

```
      ∇TEST;A;B;C
[1]   A←⍳10
[2]   B←⎕A
[3]   C←A×B
[4]   ∇
      TEST
DOMAIN ERROR
TEST[3] C←A×B
          ^
      )SI
TEST[3] *
      )SIV
TEST[3] *       A       B       C
```

`)SIS` (number)

Displays the contents of the State Indicator (see `)SI` above), showing the line number and the statement that was being executed. A carat mark indicates where execution was interrupted. If no number is specified in the command, each line of the State Indicator is shown. If a number is specified, that many lines of the State Indicator will be displayed.

```
      )SIS                (using the sample functions shown with )SI)
TEST[3]         C←A×B
                  ^
MIDDLE[2]       TEST
                ^
TOP[2]  MIDDLE
        ^
*       TOP                   (The asterisk indicates that the function
        ^                      executed from desk-calculator mode)

      )SIS 1
TEST[3]         C←A×B
                  ^
      )SIS 2
TEST[3]         C←A×B
                  ^
MIDDLE[2]       TEST
                ^
```

`)SIV` (number) or `)SINL` (number)

Lists the names of all halted user functions and operators and associated local variables. If no number is specified in the command, each line of the State Indicator is shown. If a number is specified, that many lines of the State Indicator will be displayed.

```
      ∇TEST;A;B;C
[1]   A←⍳10
[2]   B←⎕A
[3]   C←A×B
[4]   ∇
      TEST
DOMAIN ERROR
TEST[3] C←A×B
          ^
      )SI
TEST[3] *
      )SIV
TEST[3] *       A       B       C
```

`)SLOAD` (lib) name (:pass)

Silent load. Same as `)LOAD` except that no message is displayed on execution, unless there is an error.

`)SPCOPY` (lib) name (name(s))

Silent copy. Same as `)PCOPY` except that no message is displayed on execution, unless there is an error.

)SSAVE (lib) (name (:pass))

Silent save. Same as)SAVE except that no message is displayed on execution, unless there is an error.

)SWSID (lib) name (:pass)

Silent workspace identification. Same as)WSID except that no message is displayed on execution.

)SYMBOLS (number)

The symbol table is a list maintained by APL of all variable names, function names, labels and other names used in the current workspace. To discover how much space is available in this table, use the)SYMBOLS command on its own. The default symbol table size will vary from system to system, but in most implementations of APLX it is 1026.

To alter the number of symbols that you can use, you can alter the symbol table size. The system will allocate a symbol table size at least as big as you have requested, up to the maximum. This command can only be used in a CLEAR WS and must be the first command carried out.

```
      )SYMBOLS
IS 270, USED 5
      )SYMBOLS 500
INCORRECT COMMAND
      )CLEAR
CLEAR WS
      )SYMBOLS 4000
WAS 1026
      )SYMBOLS
IS 4023, USED 0
```

Exceptionally you may get the error message SYMBOL TABLE FULL. If this happens, you must save the current workspace, clear the workspace and reset the size of the symbol table by typing the)SYMBOLS command followed by a new limit. You can then use)COPY to call back your functions and variables. There is little point in specifying a very large size as the symbol table uses up memory. The maximum symbol table size is 32022.

)TABS (number)

Use the)TABS command on its own to list the current tab positions. Use with a vector of numbers from 1 to 160 to set the logical tab positions for automatic tabbing of input and output. Alternatively, a scalar argument X will set the tab stops every X columns. These must coincide with the physical tab settings on the terminal.

```
      )TABS
ARE NOT SET
      )TABS 10 25 60
WERE NOT SET
      )TABS
ARE 10 25 60
      )TABS 10
WERE 10 25 60
      )TABS
ARE 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160
```

This command is implementation dependent, and generally applies only for versions of APLX which output to a 'dumb' terminal.

)TIME

Displays the current date and time in the format currently set by the ⎕CONF function. Various formats are allowed, for example:

```
      )TIME
1990-05-24 21.15.55           (ISO)
      )TIME
24/5/1990  21.15.55           (European)
      )TIME
5/24/1990  21.15.55           (US)
```

)VARS (first (last))

Lists the names of all global variables in the current workspace. If the command is followed by a character or group of characters, the list gives the names of all variables beginning with that character or group of characters onwards (the parameter `first`, used on its own). A second character or group of characters after the command (the parameter `last`) is used to end the list of names. Names are shown in alphabetic order, fully sorted.

```
      )VARS
A       A_NAME  AA      AAA     ANT     ART     B       BAT     CAT
CAULDRON        EVENT   GOOD    GREAT   GREATEST        MANY    MORE
MOST  MUCH    THE¯END THE_VERY_END    ∆       ⍙
      )VARS T
THE¯END THE_VERY_END    ∆        ⍙
      )VARS A D
A       A_NAME  AA      AAA     ANT     ART     B       BAT     CAT
CAULDRON
      )VARS ANT GREATEST
ANT     ART     B       BAT     CAT     CAULDRON        EVENT   GOOD
GREAT   GREATEST
```

)WIDTH (number)

Lists the current WIDTH setting. This determines the maximum number of character positions in a line of output. This is normally 80. You can change it to any number between 40 and 390 by typing the command followed by the number. (See also ⎕PW.)

```
      )WIDTH
IS 80
      )WIDTH 65
WAS 80
      )WIDTH
IS 65
      ⎕PW
65
```

)WSID (lib) (name (:pass))

Workspace identification. Used on its own, it tells you the name of the current workspace. Followed by a name (optionally preceded by a library number) it names the current workspace. If you include a password (preceded by a colon), you will have to use the password when you subsequently access the workspace.

```
      )WSID FRED
WAS CLEAR WS
      )WSID 3 FRED:SECRET     (password is SECRET, library is library 3)
WAS FRED
      )WSID /usr/workspaces/FRED.aws           (full pathname supplied)
WAS 3 FRED
```

Library specification and path names

There are two different ways in which you can specify the full path associated with the workspace name:

- As shown in the first two examples above, you can specify the workspace name as just the base name of the workspace, for example `MYWS` or `Budget03`, optionally preceded by a library number. In this case, APLX appends any default file-extension to the name (`.aws` for Windows, AIX or Linux), and makes the full path for the workspace by prepending directory corresponding to the specified library number. Library numbers 0 to 9 are set up either using the Preferences dialog, or by using the `⎕MOUNT` system function. Library 10 contains the utility and demonstration workspaces supplied with APLX. If you omit the library number, library 0 is assumed.

- As shown in the third example above, you can specify a full operating-system path name, including directory separation characters, such as `/usr/workspaces/Budget03.aws` *(Linux)*, `C:\workspaces\Budget03.aws` *(Windows)*, or `MacHD::workspaces:Budget03` *(MacOS)*. APLX uses the path name exactly as supplied, so under Linux, Windows and AIX you usually need to provide the `.aws` file extension.

See the description of the `)LOAD` system command for more detail on libraries and path names.

`)XLOAD` (lib) (name (:pass))

Loads a named workspace, without executing the latent expression. The workspace is loaded into memory and overwrites the workspace already there. If the workspace has a password it must be given. Upon loading the Latent Expression (see `⎕LX)` is not executed unless attention checking is set to OFF for the workspaces (see 5 `⎕CONF)`.

See the description of the `)LOAD` system command for details on specifying workspace names and library paths.

Section 7: System Functions & Variables

⎕A Alphabet, Upper Case

The niladic system function ⎕A returns a result which is a 26 character vector of the letters in the alphabet in upper case only.

```
      ⎕A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
      ⎕A⍳'APL'
1 16 12
```

⎕a Alphabet, Lower Case

The niladic system function ⎕a returns a result which is the 26 characters of the lower-case alphabet.

```
      ⎕a
abcdefghijklmnopqrstuvwxyz
```

⎕AF Atomic Function

The monadic function ⎕AF is used to convert characters into integers and vice versa. If the right argument is a character array, the result is an integer array of the same shape, with each element being the position in ⎕AV of the corresponding character. If the right argument is an integer array, the result is a character array of the same shape, with each element being the character at the corresponding position in ⎕AV.

Exceptionally for an APL function it is origin independent, always assuming origin 0. It is the same as ⎕AV⍳ or ⎕AV[..] (in origin 0). Integer arguments must be in the range 0 to 255 and the right argument must always be a simple array.

```
      ⎕AF 'LEVEL II'
108 101 118 101 108 32 105 105
      ⎕AF 108 101 118 101 108 32 105 105
LEVEL II
```

⎕AI Account Information

The niladic system function ⎕AI returns a result which gives a statement of the computer facilities the current APL task has used during the current APL session. It consists of 7 numbers representing respectively:

```
User  number
CPU time used by this APL task
Connect time
Keyboard  unlock time
APL  version number
Month to-date CPU  time          *
Month to-date connect time       *
```

* (or this session's CPU and connect time, depending on the implementation.)

All times are in milliseconds. By default, the user number is 1000.

⎕AT Object Attributes

The dyadic system function ⎕AT returns a variety of attributes for objects named in its right argument. The right argument should be a character scalar, vector or matrix of names and the left argument an integer in the range 1 - 4. The information available is:

Left Argument	Length of result	Attributes Returned
1	3	Valences
2	7	Fix timestamp
3	4	Execution properties
4	2	Object size

and the attributes are defined as follows (using the sample workspace defined below):

```
      )FNS                        (Sample workspace)
GO      STOP    WAIT
      )VARS
NUMS
```

Valences (Left argument 1)

The first element of the valence vector is 1 if the object is a variable or function with an explicit result and 0 for other cases. The second element is 0 1 for monadic functions, 2 for dyadic or nomadic functions, and 0 otherwise. The third element is 0 if the object is not an operator, 1 for monadic operators, 2 for dyadic operators.

```
      (⎕NL 2 3),1 ⎕AT ⎕NL 2 3
GO   1 0 0                      (Niladic function, explicit result)
NUMS 1 0 0                      (Variable)
STOP 0 0 0                      (Niladic function, no result)
WAIT 0 2 0                      (Dyadic/nomadic function, no result)
```

Fix timestamp (Left argument 2)

If the named object is a function/operator, the timestamp of the latest fix of the function is returned.

```
      (⎕NL 2 3),2 ⎕AT ⎕NL 2 3
GO   1990 5 25 0 19 14 703      (Latest fix time)
NUMS    0 0  0 0  0  0   0      (Not a function)
STOP 1990 5 25 0 19 10 906
WAIT 1990 5 25 0 19 30 640
```

Execution properties (Left argument 3)

A four element vector detailing the four execution properties:

```
Nondisplayable; Nonsuspendable; Ignores weak interrupts; Converts APL
errors to DOMAIN ERROR
```

Thus the vector is 0 0 0 0 for unlocked functions and 1 1 1 1 for locked functions.

```
      (⎕NL 2 3),3 ⎕AT ⎕NL 2 3
GO   0 0 0 0
NUMS 0 0 0 0
STOP 0 0 0 0
WAIT 0 0 0 0
```

Object size (Left argument 4)

A two element vector is returned for each object named, showing first the size of the object as a whole, and secondly either size of the data portion (for variables) or repeating the object size.

```
      (⎕NL 2 3),4 ⎕AT ⎕NL 2 3
GO   100 100                    (100 byte function)
NUMS 56  40                     (56 byte variable, 40 byte data portion)
STOP 68  68
WAIT 104 104
```

⎕AV Atomic Vector

The niladic system function ⎕AV returns a result which contains a vector of 256 characters which correspond to all the possible characters in the character set. The order in which they appear is the order of internal representation, as described in the section on the APLX Character Set.

⎕B Backspace

The niladic system function ⎕B returns the Backspace character.

```
      ⎕AV⍳⎕B
9
```

⎕BOX Vector to/from Matrix

Converts a vector to a matrix, or a matrix to a vector using optional fill characters and line delimiters. ⎕BOX will only accept simple arguments.

One-argument form

In creating a matrix from a vector, ⎕BOX uses the space characters in the vector to determine the line breaks. It also uses the space character as a filler to make up the lines to the same length.

```
      ⎕BOX 'LEE PRENDERGAST PSMITH'
LEE
PRENDERGAST
PSMITH
```

When ⎕BOX is used to convert a matrix into a vector it will treat the space character(s) at the end of each row as fillers, and will separate the original rows by spaces in the resultant vector:

```
      TAB
APL
LISP
PASCAL
      ⎕BOX TAB
APL LISP PASCAL
```

Two-argument form

The two argument form of `⎕BOX` accepts one or two character left argument. The first character is used as the delimiter and the second the filler. If no filler is specified, the space character is used. In this example, `⎕BOX` has a left argument of `'*'` and it uses `'*'` to determine breaks between lines:

```
      '*' ⎕BOX 'JAN FEB MAR* 1   2    3'
JAN FEB MAR
 1   2   3
```

Here's an example with '.' used as the filler:

```
      '/.' ⎕BOX 'LEE/PRENDERGAST/PSMITH'
LEE........
PRENDERGAST
PSMITH.....
```

In this next example, $ is treated as the delimiting character and the space character as the filler in the matrix TAB (defined above):

```
      '$' ⎕BOX TAB
APL$LISP$PASCAL
```

If the matrix contains 'filler' symbols, these can be removed:

```
      NAMETAB
LEE........
PRENDERGAST
PSMITH.....
      '$.' ⎕BOX NAMETAB
LEE$PRENDERGAST$PSMITH
```

`⎕BOX` can also use numeric left and right arguments in the same way as character arguments. By default the delimiter and fill numbers are both 0.

```
      ⎕BOX 1 2 3 0 1 2
1 2 3
1 2 0
      ¯1 ¯6 ⎕BOX 2 3 4 ¯1 6 ¯1 ¯1 8 2
 2  3  4
 6 ¯6 ¯6
¯6 ¯6 ¯6
 8  2 ¯6
```

⎕C Control Characters

The niladic system function ⎕C returns a vector of the 32 ASCII control characters and rubout. It is mainly useful for sending escape sequences to dumb terminals or other low-level devices. For example, with ⎕IO (index origin) set to 1 (the default).

```
⎕C[1]       is   NUL
⎕C[8]       is   BELL
⎕C[9]       is   BACKSPACE
⎕C[10]      is   TAB
⎕C[11]      is   LINE FEED
⎕C[13]      is   FORM FEED
⎕C[14]      is   CARRIAGE RETURN
⎕C[28]      is   ESC
⎕C[33]      is   RUBOUT
```

Note that some of these character positions have been assigned to line-drawing characters in some versions of APLX.

See also ⎕TC.

⎕CALL Call external static method

The dyadic system function ⎕CALL allows you to call a 'static' method (or access a static property) in an external environment such as .Net or Java. A 'static' method is a member of a class which can be called without needing to create an instance of the class.

The left argument is a character vector which specifies the external environment in which you want to make the call, in the same format as for ⎕NEW. The right argument is either a character vector containing the name of the method (if there are no arguments or this is a property), or a nested vector where the first element is the name of the method and subsequent elements are the arguments to the method. The explicit result is whatever the external environment returns as the result of the call; it may be an ordinary array of data, or a reference to an object in the external environment.

For example, the .Net System.DateTime class contains a static property `Now` which returns the current date and time as an instance of the `DateTime` class:

```
      '.net' ⎕CALL 'System.DateTime.Now'
[.net:DateTime]
      TS←'.net' ⎕CALL 'System.DateTime.Now'
      TS.ToString
10/10/2007 12:06:42
```

It also contains a static method `IsLeapYear` which takes an integer argument representing a year, and returns a Boolean value indicating whether the year is a leap year:

```
      '.net' ⎕CALL 'System.DateTime.IsLeapYear' 1994
0
      '.net' ⎕CALL 'System.DateTime.IsLeapYear' 1996
1
```

Similarly, in Java, to create a `TimeZone` object we need to call a static method in the `TimeZone` class:

```
      tz←java ⎕CALL 'java.util.TimeZone.getTimeZone' 'America/Los_Angeles'

      ⍝ Does this time zone use daylight savings time?
      tz.useDaylightTime
1

      ⍝ What is the time zone called with and without daylight savings time?
      tz.getDisplayName 1 (tz.LONG)
Pacific Daylight Time
      tz.getDisplayName 0 (tz.LONG)
Pacific Standard Time
```

Using a class reference to call static methods

Another way of calling a static method of an external class is to get a reference to the class itself (usually by calling `⎕GETCLASS`), and use that to access the method using dot notation. For example, in the above Java example we could have called the `getTimeZone` method as follows:

```
      tzclass←'java' ⎕GETCLASS 'java.util.TimeZone'
      tz←tzclass.getTimeZone 'America/Los_Angeles'
```

⎕CC Console Control

The monadic system function `⎕CC` provides terminal-control facilities. With the exception of arbitrary output each `⎕CC` operation is identified by a numeric code or code list which is included as the right argument to `⎕CC`. The typical form of a `⎕CC` expression is:

```
            RESULT←⎕CC OPERATION
```

where RESULT is the result of the operation and OPERATION is the appropriate code or code list. The numbers used as the right argument to `⎕CC` must be integers (whole numbers). The `⎕CC` operation specified is carried out, and in most cases an empty vector is returned as the result.

System Dependent

`⎕CC` is highly system dependent. On console-mode (dumb-terminal) implementations of APLX `⎕CC` controls the user's terminal display. On windowing implementations (*APLX for Windows* and *APLX for MacOS*), it controls the Session window.

Single key input with ⎕CC

```
            RESULT←⎕CC ¯1
```

The code, `¯1`, specifies that the next character entered on the keyboard is to be assigned to the named variable. No input translation takes place.

If the next key hit after the above statement has been executed is 'Z', then RESULT will contain 'Z'.

Arbitrary output with ⎕CC

```
            RESULT←⎕CC TEXT
```

The characters in the right argument are output to the screen without output translation. RESULT is an empty vector with display potential off.

Cursor addressing with ⎕CC

```
            RESULT←⎕CC 0 R C
```

You can move the cursor to a particular line and character position, by using the code 0, followed by the numbers of the row and column you require. Rows and columns start at 0. Invalid cursor positions are ignored.

The result of `⎕CC` is an empty vector with display potential off.

```
            [3]  ⎕CC 0 10 20 ⋄ 'HELLO'
```

The cursor moves to row 10 column 20 and the text string 'HELLO' is printed.

Basic screen control with ⎕CC

Each of the following operations can be invoked by a statement of the form:

```
            RESULT←⎕CC code(s)
```

The required operation is carried out. The result returned by the operation is in most cases an empty vector with display potential off. (An exception is `⎕CC 20` which returns a numeric vector consisting of three elements.) Multiple codes can be output at one time (up to a maximum of 50), but all must be integers.

```
            RESULT←⎕CC 20
```

Returns a three element vector which contains the row and column where the cursor is positioned and the screen width.

The complete list of `⎕CC` codes is as follows. Codes marked with an asterisk `*` are NOT implemented on the Windows, MOTIF or MacOS systems.

Operations Performed by ⎕CC			
Code	Operation	Code	Operation
1	Clear Screen	19	Insert Character
2	Home cursor	20	Read Curs. Pos, Screen Width
3	Cursor to beginning of line	21	Insert Mode On
4	Move Cursor Up One Line	22	Insert Mode Off
5	Move Cursor Down One Line	23 *	Half Intensity On
6	Move Cursor Left One Position	24 *	Half Intensity Off
7	Move Cursor Right One Position	29 *	Underlined characters On
8	APL character set	30 *	Underlined Characters Off
9	ASCII character set	31 *	Reverse Screen Video
10	Erase from Cursor to End of Line	32 *	Normal Screen Video
11	Erase from Cursor to End of Screen	33 *	Select 80 Columns
12 *	Protect Characters On	34 *	Select 132 Columns
13 *	Protect Characters Off	35	Bell (Audible Alarm)
14 *	Protection On	36	Bold Text On
15 *	Protection Off	37	Bold Text Off
16	Delete Line	38	Italic Text On
17	Insert Line	39	Italic Text Off
18	Delete Character		

On implementations which support color display, the following codes are also implemented:

101	Black text
102	White text
103	Red text
104	Green text
105	Blue text
106	Cyan text
107	Magenta text
108	Yellow text

Other code numbers are reserved.

The operations which ⎕CC can perform depend on the capabilities of the terminal and/or system, but the following operations are almost always available:

```
          1 2 3 4 5 6 7 10 11 20

          ∇ TEST
    [1]   ⍝ EXAMPLE OF ⎕CC USAGE
    [2]   ⎕CC 1 2 ⍝        CLEAR SCREEN AND HOME TO SET ⎕CC (IF NEEDED)
    [3]   ⎕CC 0 10 10
    [4]   POSITION ←⎕CC 20
    [5]   ⎕CC 29 ⍝         UNDERLINE ON
    [6]   'I AM AT POSITION ',POSITION
    [7]   ⎕CC 30   ⍝       UNDERLINE OFF
    [8]   ∇
```

⎕CHART Draw Chart of Data

Implemented on desktop editions of APLX only

`⎕CHART` provides a quick way of drawing a chart (graph) of data in an APL array. The right argument is the data to be plotted, which can comprise one or more data series. The optional left argument is a set of keyword/value pairs which allow you to customize the appearance of the chart. The chart appears in a new window. Once it has appeared, you can use the menu bar to alter the way in which it is displayed (for example, to switch to a logarithmic Y scale).

`⎕CHART` is designed to provide automatic graph-drawing facilities with very little programming effort. If you wish to have full control over how the chart is drawn, you should instead use the `Chart` object under `⎕WI`. You can also chart data by right-clicking over the name of a variable and selecting 'Display As Chart' from the pop-up menu, or by choosing 'Chart Variable..' from the Edit menu of the session window.

Data to be Charted

The right argument to `⎕CHART` is the data to be charted. It can be any of the following:

- A numeric vector (or one-row or one-column matrix). In this case, the values in the array are regarded as the Y values, and they are plotted against implicit X values 0, 1, 2...

- A numeric matrix of at least two rows and columns. In this case, APLX by default assumes that the longer dimension represents the points of each series, and the shorter dimension represents the number of series to be plotted. For example, for an array of 20 rows and 3 columns, it is assumed that there are three data sets, each containing 20 values, rather than 20 data sets, each containing 3 values.

  Once APLX has decided which dimension represents the data sets, it then tries to decide which data set (if any) represents the X values. APLX first looks to see whether the first or the last data set comprises a regularly incrementing series (such as 10, 14, 18, 22); if so, it is assumed to be the series of X values against which the other data sets are plotted as Y values. If neither data set comprises a regularly-increasing series (i.e. with a constant interval), APLX next looks to see if either data set comprises an irregularly incrementing series (such as 10, 12, 16, 24); if so, it is taken to represent the X values. Finally, if neither data set satisfies the criteria, it is assumed that all the data sets represent Y values, which are then plotted against implicit X values 0, 1, 2...

  If any of these assumptions are wrong, you can change them using the menu bar once the window has opened. You can also specify explicitly where the data is by supplying a left argument to `⎕CHART`, as discussed below.

- A nested matrix of either two rows or two columns, where one of the rows (or columns) comprises a series of numbers, and the other a set of character strings. In this case, the numbers are assumed to represent data values, and the strings labels. The data is initially plotted as a Bar chart, but using the Chart menu you can change it to a different type such as a Pie chart.

Customizing the Chart

The optional left argument to `⎕CHART` allows you to the determine how the chart is displayed. (If you omit it, APLX uses the above rules initially, and then the parameters can be adjusted using the Chart menu.)

It comprises a nested vector of one or more phrases of the form 'Keyword=Value'. (If there is only one such phrase, the argument can be a simple character vector). The options are as follows (case is ignored in checking the keywords):

`title` - Sets the title for the whole chart, for example `'title=Absorbtion of Calcium'`

`type` - Sets the type of graph, for example `'type=area'`. The value part of this phrase should be one of the following: `line scatter area bar stair horizbar` or `pie`. If you omit this phrase, a Line chart is drawn initially, unless the right argument is nested, in which case a Bar chart is drawn. Once the chart has appeared, you can change the type using the Chart menu.

`data` - Specifies how the data is laid out in the right argument, for example `'data=rows'`. The value part of this phrase should be one of the following: `rows` (the data points are along the rows of the right argument, or `columns` or `cols` (the data points are laid out down the columns of the right argument). If you omit this phrase, APLX uses the rules described above. Again you can adjust the choice once the chart has been drawn.

`x` - Specifies which data set contains the X values. This can one of `first` (the X values are in the first row or column of the right argument), `last` (the X values are in the last row or column of the right argument), or `implicit` (all the rows/columns contain Y values, which should be plotted against 0, 1, 2...). If you omit this phrase, APLX tries to guess the most likely layout using the rules described above.

`seriesname` - Allows you to specify a label to be attached to each series of the graph. You will normally want to repeat this phrase several times, once for each series in the chart. For example, if you wanted a chart showing sales for three regions against an implicit X axis of 0,1 2, you could provide an N by 3 array as the right argument, and for the left argument specify: `'seriesname=America' 'seriesname=Europe' 'seriesname=Asia' 'x=implicit'`. The data in the first row would be labelled 'America', the second 'Europe', and the third 'Asia'.

Live Charting

The `'id'` keyword can be used to tell `⎕CHART` to re-use an existing chart window to graph new data. This can be useful if you want to do simple animations, for example display a graph of changing data acquired from an external measuring device in real time. The keyword takes the form `'id=N'`, where `N` is a positive integer. When the `id` keyword is specified, `⎕CHART` will check whether there is already a chart window with the same id, creating a new window only if one is not found (See examples).

Tip: If you want to chart a variable, and have the chart change when the variable changes (in desk calculator mode), you can use an expression like this in a Watch window:

```
      'id=1' ⎕CHART X
```

Examples

Draw a sine wave. Because no X values are specified, implicit X values of 0, 1, ... 99 are used:

```
      ⎕CHART 1○0.1×⍳100
```

Draw a sine wave with the X values specified in degrees. We provide a two-column matrix; APLX assumes the second column is the X axis, because it increments regularly:

```
      ⎕IO←1
      ⎕CHART (1○(10×0,⍳36)×○2÷360),[1.5]0,10×⍳36
```

Draw a bar chart:

```
      ⎕CHART 2 4⍴'Apples' 'Pears' 'Oranges' 'Kumquats' 445 323 345 765
```

Draw the same data as a pie chart:

```
      'type=pie' ⎕CHART 2 4⍴'Apples' 'Pears' 'Oranges' 'Kumquats' 445 323 345 765
```

Draw some random data as two, named series, against implicit X values 0, 1, 2..:

```
      'x=implicit' 'seriesname=Bicycles' 'seriesname=Tricycles' ⎕CHART ?2 8⍴100
```

Draw an animated graph

```
       ∇AnimatePulse;pulse;X
[1]   pulse←(*-X÷10)×1○4×X←⍳100      ⍝ Create some fake data
[2]   pulse←(⌽pulse),pulse
[3]   :Repeat
[4]     'id=1' ⎕CHART pulse
[5]     pulse←2⌽pulse
[6]   :EndRepeat
      ∇
```

See also the Chart and Series system classes, which give you much more detailed control over the layout and appearance of graphs.

⎕CL Current Line

The result of the niladic system function `⎕CL` is a single number which identifies the line currently being executed in a user-defined function, or that which was being executed when the function was suspended. It is equivalent to `1↑⎕LC`.

⎕CLASS Class hierarchy for object or class

Not implemented for System classes

The monadic system function `⎕CLASS` returns a vector of references to the class hierarchy for an object (or a class). The right argument is an object reference or a class reference. The result is a vector of class references. The first element is a reference to the class of the object (or the class itself, if the right argument was a class reference). Subsequent elements are references to successive parent classes, if any.

For example, if the class `Poem` inherits from the class `LiteraryWork`, and `Sonnet` inherits from `Poem`:

```
      )CLASSES
LiteraryWork    Poem    Sonnet

       TwoLoves←⎕NEW Sonnet

       ⎕CLASS Poem           ⍝ Argument is a Class reference
{Poem} {LiteraryWork}
       ⎕CLASS Sonnet
{Sonnet} {Poem} {LiteraryWork}

       ⎕CLASS TwoLoves       ⍝ Argument is an Object reference
{Sonnet} {Poem} {LiteraryWork}
       (⎕CLASS TwoLoves).⎕CLASSNAME
 Sonnet Poem LiteraryWork
```

`⎕CLASS` can also be used for external classes. For example, the Ruby `DateTime` class inherits from `Date` which inherits from `Object`:

```
      DT←'ruby' ⎕SETUP 'require' 'Date'
      DT←'ruby' ⎕NEW 'DateTime'
      ⎕CLASS DT
{ruby:DateTime} {ruby:Date} {ruby:Object}
```

⎕CLASSES References to user-defined and external classes

The niladic system function ⎕CLASSES returns a vector of references to all the user-defined classes in the workspace, plus any references to external classes which have been retained in the workspace (typically by a call to the system function ⎕GETCLASS or the system method ⎕CLASSREF)

In this example, we have three user-defined classes in the workspace, and we also create a reference to the .Net DateTime class:

```
       )CLASSES
LiteraryWork    Poem    Sonnet
      DT←'.net' ⎕NEW 'System.DateTime' 2007 5 30
      CLASSDT←DT.⎕CLASSREF
      CLASSDT
{.net:DateTime}
     ⎕CLASSES
{LiteraryWork} {Poem} {Sonnet} {.net:DateTime}
```

⎕CONF Configure APL

The user can change certain parameters of the APL system via ⎕CONF. The extent to which ⎕CONF is implemented varies from system to system. The current range of options is:

```
     5    ⎕CONF     0 = disable interrupts, 1 = enable interrupts
     6    ⎕CONF     (date format) (time format) (display order)
                     date     0   =   US 8 character          mm/dd/yy
                              1   =   US 10 character         mm/dd/yyyy
                              2   =   European 8 character    dd/mm/yy
                              3   =   European 10 character   dd/mm/yyyy
                              4   =   ISO                     yyyy-mm-dd
                     time     0   =   24 hour US              hh.mm.ss
                              1   =   24 hour European        hh:mm:ss
                              2   =   12 hour US              hh.mm.ss am
                              3   =   12 hour European        hh:mm:ss am
                     order    0   =   time before date
                              1   =   date before time
     7    ⎕CONF     Preferred workspace size for this workspace (in bytes),
                    or 0 for the default value, -1 for 'as much as possible'
     8    ⎕CONF     APL component file-system name, as a character vector
```

In each case the result returned by ⎕CONF is the previous value.

The Interrupt Checking Flag is a workspace parameter. A workspace can be saved with attention checking off, thereby preventing the latent expression being interrupted.)CLEAR will enable interrupts.

The Date/Time Format controls the display format adopted by)TIME)SAVE)LOAD and the ∇ editor.

The Preferred Workspace Size parameter applies to the current workspace if you `)SAVE` it, and then it is loaded by double-clicking on the workspace icon or by dragging the workspace on to the APLX program icon. When APLX starts up, it will be allocated the workspace size if possible. This value does not affect the workspace size if you `)LOAD` the workspace once APLX is already running.

The APL Component File-System Name is the filename used for component-file operations using the ⍇ ⍈ ⍐ ⍗ primitives. It defaults to `'aplsfile.afl'`. If you change the name using `8 ⎕CONF`, the change applies only for the current APL session; when you restart APLX, the name reverts to the name stored in the APLX preferences file.

⎕CR Canonical Representation

Converts a user-defined function, operator, method, or class into a character matrix or vector. The name of the item to be converted to text form is the right argument of `⎕CR`. `⎕CR` can be used monadically, or optionally with a left argument. The value of the left argument determines the shape of the result.

Left Argument	***Result***
Omitted or 0	Character matrix
1	Character vector with embedded carriage-returns separating lines
2	Vector with each item a vector of characters corresponding to one line of the text form

The matrix or vector shows the lines of the function, operator, method or class, with line numbers omitted. It has the characteristics of any object composed of characters. You can, for example, use indexing to alter its elements, or use the ↓ and ↑ functions on it. And, if you want, you can arrange for another function to make the alterations for you by specifying within that function the alterations you require. To convert a function called DT to character data

```
            MATRIX←⎕CR 'DT'         (for a matrix result)
            VECTOR←1 ⎕CR 'DT'       (for a simple delimited vector result)
            VECTOR←2 ⎕CR 'DT'       (for a nested vector result)
```

See also `⎕FX` for the reverse operation, which takes the text form and fixes it as a function, operator or class.

Canonical representation of a class

`⎕CR` can also be used to convert an entire class to text form. In this case, the right argument should be a class name. The format of the result is as follows:

- The first line is the class header. This comprises the name of the class, followed (if the class inherits from another class) by a colon and the name of the parent class. Any private members of the class (i.e. names which are local to the class) are then listed, separated by semi-colons. The header line ends with a left curly brace '{' character.

- The properties of the class are then listed, one per line. The name of the property is listed first. If it has a default value, an assignment arrow follows, and then the transfer form of the expression which initializes the property. If the property is read-only, two assignment arrows are used. If the property is class-wide (i.e. there is only a single copy shared between all instances in the workspace, then whole line is enclosed in curly braces.

- The methods of the class (and the constructor, if any) are then listed, with a del character starting and ending each method.
- Finally, the class ends with a closing right curly brace, on a line by itself.

For example:

```
      ⎕CR 'Point'
Point; IncSerial;serial {
Z←0
Y←0
X←0
{serial←0}
{CATEGORY←←'Geometric'}

∇IncSerial
⍝ Increment serial number
serial←serial+1
∇

∇R←Mag
⍝ Return magnitude (distance from origin)
R←+/(X,Y,Z)*2
R←R*0.5
∇

∇Point B
⍝ Constructor for Class B.  Optionally set up X Y Z
:If 0≠⍴B
  (X Y Z)←B
:EndIf
IncSerial
∇

∇R←GetSerial
R←serial
∇
}
```

In the above example, the class `Point` has three instance properties (`X` `Y` and `Z`), all initialized to 0. It has two class-wide properties `serial` and `CATEGORY`. `CATEGORY` is a read-only property, so is initialized with a double assignment arrow. The `serial` property is private (i.e. not accessible from outside the class), so it is localized in the class header. The class has three ordinary methods (of which one, `IncSerial` is private), and a constructor (identifiable by the fact that it has the same name as the class).

If a second class `MovingPoint` inherits from `Point`, and adds a new property `VELOCITY`, then you might have:

```
      ⎕CR 'MovingPoint'
MovingPoint : Point {
VELOCITY←0
}
```

This shows the use of the colon character in the header, to indicate that `MovingPoint` inherits from `Point`.

Canonical representation of a single method from a class

`⎕CR` can also be used to convert a single method from a given class to text form. In this case, the right argument should be the fully-qualified method name (i.e. the class name, a period, and the method name). For example:

```
      ⎕CR 'Point.Mag'
R←Mag
⍝ Return magnitude (distance from origin)
R←+/(X,Y,Z)*2
R←R*0.5
```

⎕CS Compatibility Setting

The system variable `⎕CS` (Compatibility Setting) can be used to ensure that applications written under APL.68000 Level I work in the same way under APLX. It can be set to a number in the range 0 to 7. It is normally set to 0, but if a Level I workspace is loaded it will automatically be set to 7. The behaviour affected by `⎕CS` is as follows:

`⎕CS←1`	Expressions of the form `5 6 7[2]` will be allowed rather than giving RANK ERROR.
`⎕CS←2`	`⎕NC` and `⎕NL` will use the code 4 rather than ¯1 to indicate an invalid name.
`⎕CS←4`	Default formatting of numeric arrays uses the same width for all columns, rather than determining the width separately for each column.

The three parameters can be set in any combination by adding together the codes. For example:

```
      ⎕CS←0
      1 3 5[2]
RANK ERROR
      1 3 5[2]
           ^
      2 3⍴1 2 3 1 100 1000    (Each column has its own format)
1   2    3
1 100 1000
      ⎕CS←5                   (Set codes 1 and 4)
      1 3 5[2]
3
      2 3⍴1 2 3 1 100 1000    (All columns share the same format)
    1    2    3
    1  100 1000
```

⎕CT Comparison Tolerance

The setting of the system variable ⎕CT (comparison tolerance) determines the accuracy of the comparative and logical functions. Comparison Tolerance will only matter, in practice, if either of the arguments of an affected function is represented internally as a floating-point number.

The following primitive functions are affected by ⎕CT:

```
            ⌈ ⌊ < ≤ = ≥ > ≠  ∊ ⍳  ∣  ⍷  ≡  ≢  ~
```

For equality tests, two numbers are judged to be equal if the magnitude of their difference does not exceed the value of ⎕CT multiplied by the larger of their magnitudes. X≥Y is true if X-Y is greater than or equal to ⎕CT multiplied by larger of the magnitudes of X or Y.   X>Y is true if X≥Y is true and X=Y is not.

The effect on ⌈ and ⌊ is similar. Both of these functions will have no effect on an integer. A value which is close to an integer by proportionately less than ⎕CT returns that integer irrespective of the direction in which it differs from that integer. All other values behave as expected. The residue function ∣ is fuzzy, and thus A∣B will return 0 if B÷A is within ⎕CT of an integer value.

The default in 32-bit implementations of APLX is 1E¯13, and in 64-bit implementation is 3E¯15. It can be reset by assignment to a value between 0 and just less than 1.

```
      ⎕CT                        (⎕CT at normal setting)
1E¯13
      4=3.9
0
```

Compare it with the result produced by the same expression after ⎕CT has been changed:

```
      ⎕CT← .026                  (4×⎕CT is greater than the
      4=3.9                       difference between the two numbers)
1
      ⎕PP←15                     (X is less than 1 by
      ⎕←X←1-.9×⎕CT                proportionately less than ⎕CT)
0.9766
      ⌊X
1
      ⌈X
1
```

Special considerations for 64-bit versions of APLX

In APLX64, integers are represented as 64-bit numbers, and floating-point numbers are represented in 64-bit IEEE floating-point format, with 53 bits of precision. The default value of ⎕CT is 3E¯15. This means that, if you are dealing with numbers larger than around 2*48 (approx 2.8E14), you may get different answers for operations which depend on ⎕CT, according on whether the number is represented internally as an integer or a floating-point number. This is because operations which act on integers are carried out using exact arithmetic, without reference to ⎕CT, whereas operations which involve floating-point numbers do take account of ⎕CT.

Consider this sequence:

```
      X←2*50
      X
1125899906842624
      ⎕DR X
2
      X=X+1
0

      Y←1.0×X
      Y
1125899906842624
      ⎕DR Y
3
      Y=Y+1
1
```

In this example, X is represented internally as a 64-bit integer. It is distinct from the 64-bit integer X+1, because ⎕CT is ignored when APLX compares numbers represented as integers.

In contrast, Y (which has the same value as X) is held internally as a floating-point number. When the comparison with Y+1 is made, APLX reports that they are equal because the relative difference is less than ⎕CT.

Note that, in 32-bit versions of APLX, the behavior is the same, but you do not normally notice it because, with the default value of ⎕CT, no number which could be represented as a 32-bit integer is within comparison tolerance of the adjacent integer values.

Practical implications for 64-bit APLX applications

If the magnitude of the numbers you are dealing with is less than 2*48, then with the default value of ⎕CT there should be no problems. The results of arithmetic and comparison operations will not depend on whether the numbers are represented internally as floating-point or integer.

If the magnitude of the numbers you are dealing with is between 2*48 and 2*53, you might need to reduce ⎕CT to ensure consistent results, or alternatively force the numbers to be represented as integers or as floats before making the comparisons.

If the magnitude of the numbers you are dealing with is greater than 2*53, then numbers represented internally as floating-point cannot be converted to integer because there is not enough precision in the floating-point representation to know which integer is the correct one. If you want exact comparisons, you need to ensure that the numbers remain in integer form (for example, do not place them in the same array as non-integral values).

⎕D Digits

The niladic system function ⎕D returns a 10 element character vector containing the digits 0 to 9.

```
      ⎕D
0123456789
```

⎕DBR Delimited Blank Removal

One-argument form

Used with character data to remove leading and trailing blanks and compress embedded multiple blanks to one blank. The argument must be a simple character vector or scalar.

```
      ⎕DBR '    JAN    FEB   MAR APRIL      MAY          '
JAN FEB MAR APRIL MAY
```

Two-argument form

A left-hand argument can be included. It defines a character or characters to be used as delimiters in the right-hand argument. Leading and trailing blanks, and blanks preceding and following the delimiters are suppressed. Unless the space character is included in the left argument, other blanks remain:

```
      '*'  ⎕DBR  '  X   JAN  * FEB   MAR  *APRIL    * MAY   Y  '
X   JAN*FEB   MAR*APRIL*MAY   Y
      '* ' ⎕DBR  '  X   JAN  * FEB   MAR  *APRIL    * MAY   Y  '
X JAN*FEB MAR*APRIL*MAY Y
```

⎕DISPLAY Display Array Structure

The monadic system function ⎕DISPLAY returns a character matrix which shows the structure of an APL array. It is equivalent to the DISPLAY function supplied with previous versions of APLX and APL.68000, and with many other APL interpreters. It takes any APL expression as its argument, and returns a character matrix which is the formatted array with symbols showing the structure and nesting.

Note: In desktop editions of APLX, you can invoke a Display Window to show array structure, using the pop-up menu which appears when you right-click (or, under MacOS, click-and-hold) over a variable name. For named variables, you can also use the)DISPLAY system command. In both cases the array structure is shown in the same form as shown here.

⎕DISPLAY takes the data or expression whose value is to be examined as a right argument. For example:

```
        ⎕DISPLAY 1 2 3
```

The output from ⎕DISPLAY shows the data with boxes around it. The output can have one or many boxes inside the perimeter box. Also you will see that the characters at the left hand end of the top and bottom lines and those at the top of the left side line will change for different data types. It is these boxes and their embedded characters which inform you of the specific structure of the data being examined.

The symbols indicate the type of variable contained within the box while the number of boxes within boxes shows the depth of the data. Here are some simple examples:

```
      ⎕DISPLAY 3 7 8
┌→────┐
│3 7 8│
└~────┘
```

This first example is a simple 1 dimensional vector. This is indicated by the absence of an arrow on the left hand side of the box showing that there is no organisation along this axis, only along the horizontal axis as shown by the right pointing arrow at the top of the box. The '~' symbol at the bottom of the box indicates numeric data.

```
      ⎕DISPLAY 3 4⍴⎕A
┌→───┐
↓ABCD│
│EFGH│
│IJKL│
└────┘
```

This second example shows a two dimensional (rank 2) character matrix. The right arrow and the down arrow indicate two dimensions, while the absence of the '~' character indicates text or character elements.

Here is a summary of the special characters and their meanings:

	Placement on box	Meaning
-	Beneath a character	Scalar character
→	Left of top edge	Vector or higher-rank array
~	Left of bottom edge	Numeric data
+	Left of bottom edge	Mixed data
⊖	Left of top edge	Empty vector or higher-rank array
↓	Left side of box	Matrix or higher-rank array
⌽	Left side of box	Empty matrix or higher-rank array
∊	Left of bottom edge	Nested array

Where an array is empty, this is shown by the '⌽'character on the left side of the box or the '⊖'character on the top side, and the prototype of the array is shown inside the box.

Examples

```
      ⎕DISPLAY (3 4 5) 6 (9 9)
┌→──────────────────┐
│ ┌→────┐   ┌→──┐  │
│ │3 4 5│ 6 │9 9│  │
│ └~────┘   └~──┘  │
└∊──────────────────┘
```

The '∊' is on the outside box because the data within is nested. The inner boxes however have '~' characters showing that within these smaller boxes only numeric data exists. The depth of the data is two:

```
      ≡ (3 4 5) 6 (9 9)
2
```

This can be read from the display by counting the greatest number of nested boxes, two.

```
      ⎕DISPLAY 'A'                    ⍝ Character scalar
 A
 -

      ⎕DISPLAY 1                      ⍝ Numeric scalar
 1
```

(Note here that there is no need for a complete box with scalars)

```
      ⎕DISPLAY ⍳0                     ⍝ Empty numeric vector
┌⊖┐
│0│
└~┘

      ⎕DISPLAY 'A B C D E F'            ⍝ Character vector
┌→──────────┐
│A B C D E F│
└───────────┘

      ⎕DISPLAY 3 4⍴1 2 3 4 5            ⍝ 2-dimensional numeric matrix
┌→──────┐
↓1 2 3 4│
│5 1 2 3│
│4 5 1 2│
└~──────┘

      ⎕DISPLAY 1 1 1 2 5⍴'MICRO APL '    ⍝ Multi-dimensional character matrix
┌┌┌┌→────┐
↓↓↓↓MICRO│
││││ APL │
└└└└─────┘

      ⎕DISPLAY 2 4 5⍴⍳30                   ⍝ 3-dimensional numeric matrix
┌┌→─────────────┐
↓↓ 1  2  3  4  5│
││ 6  7  8  9 10│
```

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

```
      ⎕DISPLAY 1 2 3 'A' 4 5 6               ⍝ Mixed vector
┌→────────────┐
│1 2 3 A 4 5 6│
└+────────────┘
```

(Note the plus sign in the bottom left denoting mixed data)

```
      ⎕DISPLAY (1 2 3 4) (50 60 70) (89 99 109)  ⍝ Nested vectors
┌→─────────────────────────────────┐
│ ┌→──────┐ ┌→───────┐ ┌→────────┐ │
│ │1 2 3 4│ │50 60 70│ │89 99 109│ │
│ └~──────┘ └~───────┘ └~────────┘ │
└∊─────────────────────────────────┘

      ⎕DISPLAY (2 3⍴⍳6) (1 1⍴1) (1 2 2⍴⍳4)        ⍝ Nested matricies
┌→───────────────────┐
│ ┌→────┐ ┌→┐ ┌┌→──┐ │
│ ↓1 2 3│ ↓1│ ↓↓1 2│ │
│ │4 5 6│ └~┘ ││3 4│ │
│ └~────┘     └└~──┘ │
└∊───────────────────┘

      ⎕DISPLAY ⊂'G H I J',(⊂'K L M N') (3 2⍴⎕D)  ⍝ Nested (⎕D is character)
┌──────────────────────────────────────┐
│ ┌→─────────────────────────────────┐ │
│ │               ┌───────────┐ ┌→─┐ │ │
│ │ G   H   I   J │ ┌→──────┐ │ ↓01│ │ │
│ │ - - - - - - - │ │K L M N│ │ │23│ │ │
│ │               │ └───────┘ │ │45│ │ │
│ │               └∊──────────┘ └──┘ │ │
│ └∊─────────────────────────────────┘ │
└∊─────────────────────────────────────┘


      ⎕DISPLAY ⊂⊂⊂(3 4⍴⎕A) (2 3 7⍴⍳42),'A'        ⍝ Mixed nested matrices
┌──────────────────────────────────────────────┐
│ ┌──────────────────────────────────────────┐ │
│ │ ┌──────────────────────────────────────┐ │ │
│ │ │ ┌→─────────────────────────────────┐ │ │ │
│ │ │ │ ┌→──┐ ┌┌→───────────────────┐   │ │ │ │
│ │ │ │ ↓ABCD│ ↓↓ 1  2  3  4  5  6  7│ A │ │ │ │
│ │ │ │ │EFGH│ ││  8  9 10 11 12 13 14│ - │ │ │ │
│ │ │ │ │IJKL│ ││15 16 17 18 19 20 21│   │ │ │ │
│ │ │ │ └────┘ ││                    │   │ │ │ │
│ │ │ │        ││22 23 24 25 26 27 28│   │ │ │ │
│ │ │ │        ││29 30 31 32 33 34 35│   │ │ │ │
│ │ │ │        ││36 37 38 39 40 41 42│   │ │ │ │
│ │ │ │        └└~───────────────────┘   │ │ │ │
│ │ │ └∊─────────────────────────────────┘ │ │ │
│ │ └∊─────────────────────────────────────┘ │ │
│ └∊─────────────────────────────────────────┘ │
└∊─────────────────────────────────────────────┘
```

```
      ⎕DISPLAY 0⍴(2 2⍴⍳4) ('PERSIMMON')        ⍝ Empty nested vector
┌⊖───────┐
│        │
│ ┌→───┐ │
│ ↓0  0│ │
│ │0  0│ │
│ └~───┘ │
│        │
└∊───────┘
```

(Note that the prototype of the array is shown in the box)

⎕DL Delay

The monadic system function ⎕DL delays execution for not less than the number of seconds specified and then returns a numeric result which is the number of seconds actually delayed. You can specify any positive number including values less than 1, but the time resolution varies from system to system. It is usually around 10ms or better.

```
[2]   .....
[3]   'YOU HAVE 10 SECONDS IN WHICH TO WORK OUT THE ANSWER.'
[4]   A←⎕DL 10
[5]   'YOUR TIME IS UP.'
```

⎕DR Data Representation

The system function ⎕DR is used to examine or alter the internal type of an item of data using the following codes:

```
1   =   Boolean (patterns of 1s and 0s)
2   =   Integer
3   =   Floating point
4   =   Character
5   =   Overlay
6   =   Nested or mixed
7   =   Object or class reference
```

One-argument form

Reports data type of any array.

```
      ⎕DR 2.9
3
      X←1 0 1 1 0 1
      ⎕DR X
1
      ⎕DR 'ABC' 1 2 3
6
      ⎕DR (⍳10) (2 2⍴⍳4)
6
```

APL will choose which type of number format to use, and certain operations will force data to be of a specific type. For example, the result of a comparison `(< ≤ = ≥ > ≠)` is guaranteed to be a 0 or a 1, and the result of these operations is thus Boolean. Internally, the different types of data take up different amounts of space:

```
Code    Type                Space
───────────────────────────────────────────────
  1     Boolean             1 bit per element
  2     Integer             4 bytes per element   (32 bits)
                        or  8 bytes per element   (64 bits) under APLX64
  3     Floating point      8 bytes per element   (64 bits)
  4     Character           1 byte per element    (8 bits)
  5     Overlay             see ⎕OV
  6     Nested/mixed        depends on contents
  7     Object/Class ref.   4 bytes per element   (32 bits)
                        or  8 bytes per element   (64 bits) under APLX64
```

(see also `⎕AT`).

Two-argument form *(scalar left argument in range 1 to 4)*

On occasions it is useful to examine or change the data type, either for encryption purposes, or to combine character and numeric data quickly. When used with a left argument consisting of one of the data-type codes in the range 1 to 4, and a right argument consisting of a data array of type 1 to 4, `⎕DR` converts the item to the representation specified:

```
          1 ⎕DR 5
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
                                    (The 32-bit binary pattern for 5)
```

Note that, under APLX64, integers are represented internally as 64-bit numbers, so converting an integer scalar to binary returns a length-64 binary vector.

The conversion of one type to another changes the number of elements in data, and so `⎕DR` must change the dimensions of its result. The last dimension of the result is adjusted to compensate for the type change. Zero bits are used to pad out a row if necessary.

How the data conversion works

In essence, all that dyadic `⎕DR` does is to change the workspace entry's type, without changing the bit pattern of the data.

Suppose you start with the character vector `'1234'`. This data is held internally as the four bytes, hex `31 32 33 34`. If those same four bytes are used as the data portion of a binary vector, you will get the bit pattern corresponding to those four bytes:

```
      1 ⎕DR '1234'
0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 0 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 0
```

The first 8 bits are the binary pattern for hex 31, i.e. `0 0 1 1 0 0 0 1`, the second 8 bits are the pattern for hex 32, etc.

If you represent this data as an integer (in a 32-bit version of APLX), you will get the 32-bit integer which corresponds to this bit pattern:

```
      2 ⎕DR '1234'
825373492
```

which is the decimal number equivalent to hex 31323334.

Notes

a) For a scalar or vector, the length obviously changes (a 32-bit scalar integer becomes a length four character vector or a length 32 binary vector). For higher dimensional arrays, the last dimension is increased or reduced as necessary.

b) But what if there are not enough elements? For example, suppose we ask for the 8-byte float representation but only give it 4 bytes (this would happen if we ask for `3 ⎕DR '1234'`). The rule APLX applies in this case is that the data is padded on the right with null (zero) bytes to make up the necessary number of whole data elements. So `3 ⎕DR '1234'` is the same as `3 ⎕DR '1234', ⎕AF 0 0 0 0`

c) Changing arbitrary data to float is potentially dangerous because you can produce a bit pattern which is not legal as a 64-bit IEEE floating point number (the internal representation used by APLX for float numbers).

d) If you are running APLX64, the 64-bit version of APLX, you will get different answers for integer arguments, because integers are represented as 8 byte-numbers in APLX64.

In APLX:

```
      1 ⎕DR  825373492
0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 0 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 0
      ⍴1 ⎕DR  825373492
32
```

In APLX64:

```
      1 ⎕DR  825373492
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 0 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 0
      ⍴1 ⎕DR  825373492
64
```

Byte-Ordering Issues

In the case of a big-endian processor (PowerPC, SPARC, 68000, etc), the 32-bit hex number 31323234, if viewed as a series of bytes, is in the order you expect: hex bytes 31 32 33 34. But on a little-endian processor (Intel), it is backwards: 34 33 32 31. So the question arises: on a little-endian processor, should the result of transforming a 32-bit integer to binary/character treat the data as a 32-bit container (hex 31323334), or as a series of four bytes as they would appear in memory (hex 34 33 32 31)? In APLX, by default it is treated as 32-bit container (i.e. effectively swap the bytes for the little-endian case - but see below for changing this default). This design decision was taken for two reasons. Firstly, it means the result is the same on all APLX (32-bit) platforms. Secondly, it means the results are consistent with what you would reasonably expect. For example, on all 32-bit platforms APLX gives the following result when converting an integer to binary:

```
    1 ⎕DR 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
```

(the bit pattern for the integer 2, comprising four bytes)

```
     ⎕AF 4 ⎕DR 2
0 0 0 2
```

(the individual bytes which make up the 4-byte number 2, even though on a little-endian machine they would be backwards if the processor read them as individual bytes)

```
     2 ⎕DR 4 ⎕DR 1 ⎕DR 2
2
```

(Convert the integer number 2 to binary. Convert the resulting binary vector to a length 4 character vector. You get back the number you started with.)

Two-argument form *(vector left argument or compatibility mode)*

As well as the default conversions described above, you can force alternative representations of the converted data by providing a two- or three-element vector as the left argument. You can also use alternative codes for the standard conversions 1 to 4 in the first element, for compatibility with other APL interpreters, as follows:

Code	Type	Space	
11	Boolean	1 bit per element	
83	Integer	1 byte per element	(8 bits)
163	Integer	2 bytes per element	(16 bits, little-endian)
323	Integer	4 bytes per element	(32 bits, little-endian)
7	Integer	8 bytes per element	(64 bits)
643	Integer	8 bytes per element	(64 bits, little-endian)
645	Floating point	8 bytes per element	(64 bits, little-endian)
82	Character	1 byte per element	(8 bits)

The optional second element, if supplied, is the number of bytes per element when converting from character to integer or float, and vice versa. 0 means use the default value implied by the first argument.

The third element, if supplied, is the byte-ordering (endian) flag:

0	Big-endian or as implied by the fist element (default)
1	Little-endian
2	Natural-endian (big on big-endian systems, little on little-endian systems)

Note that you can only specify the element size if you are converting to or from a character representation.

Examples

```
      ⎕AF 4 ⎕DR 2       ⍝ Convert 32-bit integer 2 to four characters
0 0 0 2
      ⎕AF 4 0 1 ⎕DR 2   ⍝ Same, but little-endian byte order
2 0 0 0
      ⎕AF 4 2 1 ⎕DR 2   ⍝ Force integer to be consider a 16-bit value
2 0
      ⎕AF 4 2 1 ⎕DR 200000 ⍝ Too large to fit in 16 bits
DOMAIN ERROR
      ⎕AF 4 2 1 ⎕dr 200000
         ^

      ⎕AF 82 ⎕DR 2      ⍝ Compatibility mode: Int to Char, little-endian
2 0 0 0
      323 ⎕DR 82 ⎕DR 23 ⍝ Round trip
23

      ⎕AF 4 ⎕DR 2.56    ⍝ Convert float to 8 bytes (IEEE Double representation)
64 4 122 225 71 174 20 122
      ⎕AF 4 4 ⎕DR 2.56  ⍝ Convert float to 4 bytes (IEEE Single representation)
64 35 215 10
```

⎕EA Execute Alternate

Note: The use of `⎕EA` *is now deprecated, unless you need to retain compatibility with IBM's APL2. For most cases, we recommend that you use the structured-control error trapping mechanism* (`:Try :CatchIf :CatchAll :EndTry`) *instead.*

The dyadic system function `⎕EA` will attempt to execute its right argument. If an error (or interrupt) occurs it will then attempt to execute its left argument. Errors in the left argument will be handled as they would be normally.

```
      5÷0
DOMAIN ERROR                  (Standard error)
      5÷0
      ^
      '5÷1' ⎕EA '5÷0'         (Alternative expression executed)
5
      '3÷0' ⎕EA '5÷0'         (The alternative will report an error
⍎ DOMAIN ERROR                 in the usual way if it contains an error)
      3÷0
      ^
```

See the section on Error Handling for more information.

⎕EC Execute Controlled

The monadic system function `⎕EC` will execute its argument and return a result which is a three item vector containing a return code, the error type and the result (or `⎕EM`) respectively. The return code is an integer in the range 0 to 5

```
0     Error
1     Expression with a result which would display
2     Expression with a result which would not display
3     Expression with no explicit result
4     Branch to a line
5     Naked branch
```

The second item is the value that would be returned by `⎕ET` (without altering the current value of `⎕ET`). The third item is the result (if one is generated); for return code 3 or 5 the third item is 0 `0⍴0`; for return code 4 the third item is the argument to the branch; for return code 0 the third item is `⎕EM`.

```
      ⎕ET
0 0
      ⎕EC '2×''A'''
 0  5 4  DOMAIN ERROR           (Error, ⎕ET, ⎕EM)
              2×'A'
              ^
      ⎕ET
0 0                            (⎕ET has not changed)
```

`⎕EC` overrides any `⎕STOP` settings

```
      ∇TEST
[1]     2×5
[2]     +/⍳100
[3]     ∇
      1 ⎕STOP 'TEST'           (Stop set on line 1)
1
      ⎕EC 'TEST'               (Execute TEST under ⎕EC)
10
5050
3  0 0                         (No explicit result, ⎕ET, empty vector)
      ⎕STOP 'TEST'             (Stop setting still active)
1
```

⎕EDIT Edit fn/op/var

The APLX editor may be accessed via the system function ⎕EDIT (see also)EDIT).

If you are creating a new object, the optional left argument is 0 if you want to create a function, 1 if you want to create a variable, or 2 if you want to create a class. The default is to create a function. The left argument is ignored if the object already exists.

```
⎕EDIT 'NAME'          ⍝ EDIT EXISTING OBJECT <NAME>
                      ⍝ EDIT NEW FUNCTION OR OPERATOR <NAME>
0 ⎕EDIT 'NAME'        ⍝ EDIT NEW OR EXISTING FUNCTION OR OPERATOR
1 ⎕EDIT 'NAME'        ⍝ EDIT NEW OR EXISTING VAR <NAME>
1 ⎕EDIT 'CLASS'       ⍝ EDIT NEW OR EXISTING CLASS <NAME>
```

The operation of the editor may vary according to the implementation of APLX being used. Under Windows and MacOS, APL execution will be suspended until the editing is complete if you invoke it with ⎕EDIT, but not if you invoke it in any other way.

⎕EM Error Matrix

⎕EM contains the current error message as a character matrix. The initial value of ⎕EM is 3 0⍴' ' and it is reset to this value by)RESET or)SICLEAR. Note that the error message can originate from calculator mode errors or errors within defined functions. ⎕EM is implicitly localised in that it shows the error message which relates to the most currently pendent function on the SI stack - see the earlier chapter on Error Handling for more details.

```
      )RESET
      ⍴⎕EM                    (Default value of ⎕EM)
3 0
      45×'NAME'               (Standard error report)
DOMAIN ERROR
      45×'NAME'
      ^
      ⍴⎕EM                    (Size of ⎕EM after the error)
3 15
      ⎕EM                     (Contents of ⎕EM are the same as the
DOMAIN ERROR                   error message above)
      45×'NAME'
      ^
      )RESET                  (Clear ⎕EM)
      ⍴⎕EM
3 0
```

⎕ERM Error Message Vector

The niladic system function ⎕ERM is the same as ⎕EM except that it returns a character vector with embedded carriage returns between lines.

⎕ERS Error signalling

The nomadic system function ⎕ERS is used to 'signal' errors to a calling function. The calling function is halted unless it is error trapped. ⎕ERS can be used with one or two arguments:

```
            ⎕ERS ERRORNUMBER
            ERRORMESSAGE ⎕ERS ERRORNUMBER
```

The ERRORNUMBER may be a vector of numbers, but the first is used as the argument. The ERRORNUMBER can be any integer, or an empty vector. The error message associated with the first number in the right argument is displayed, the calling function is halted and the first number of ⎕LER (Line Error Report) is set to this number. The exact behavior depends on the value of ERRORNUMBER:

- If ERRORNUMBER is a positive integer associated with a standard APLX error message, the associated error message is signalled. See the section on Error Codes for a list of standard error numbers.
- If the ERRORNUMBER is positive, but is not associated with a standard error message, an UNKNOWN ERROR TYPE is signalled.
- Negative ERRORNUMBERs cause an error to be signalled but no message to be displayed.
- If ERRORNUMBER is 0, no error is signalled and ⎕ER, ⎕ET, ⎕ERM and ⎕EM are reset.
- If ERRORNUMBER is an empty numeric vector, the function has no effect. No error is signalled.

An error message can be included as the left argument. If present, this is displayed instead of the normal error message associated with this number.

For example, an attempt to divide by zero normally generates:

```
      2÷0
DOMAIN ERROR
      2÷0
       ^
```

A function DIVIDE can be written which 'signals' an attempt to divide by zero, rather than stopping:

```
      ∇ R←A DIVIDE B
[1]   'ATTEMPT TO DIVIDE BY ZERO' ⎕ERS (B=0)/8
[2]   R←A÷B
[3]   ∇
      2 DIVIDE 0
DOMAIN ERROR
      2 DIVIDE 0
        ^
```

and this function can itself be used within other error trapped functions:

```
      ∇ TEST;X
[1]   X←⎕ERX ERR
[2]   START:'ENTER TWO NUMBERS '
[3]   DATA←2↑⎕                    ⍝MAKE SURE WE HAVE 2 NUMBERS
[4]   'THE DIVISION IS:'
[5]   DATA[1] DIVIDE DATA[2]
[6]   →0
[7]   ERR:→(8≠1↑⎕LER)/0           ⍝NOT ONE OF OUR  ERRORS
[8]   'ATTEMPT TO DIVIDE BY ZERO, TRY AGAIN'
[9]   →START
[10] ∇

      TEST
ENTER TWO NUMBERS
⎕: 4 2
THE DIVISION IS:
2
      TEST
ENTER TWO NUMBERS
⎕: 4 0
ATTEMPT TO DIVIDE BY ZERO, TRY AGAIN
⎕: 4 4
THE DIVISION IS:
1
```

The use of a negative argument to `⎕ERS` suppresses printing of the error message.

```
      ∇R←A DIVIDE B
[1]    ⎕ERS(B=0)/¯8
[2]    R←A÷B
      ∇   1.53.19 05/28/90
      3 DIVIDE 0
      3 DIVIDE 0                (No error message shown)
        ^
      ⎕LER                      (⎕LER set to ¯8)
¯8 0
      ⎕ET                       (⎕ET set to 0 1)
0 1
      ⎕ERS 0                    (Use of ⎕ERS 0 to reset error numbers)
      ⎕LER
0 0
      ⎕ET
0 0
```

⎕ERX Error trapping

Note: The use of ⎕ERX *is now deprecated. We recommend that you use the structured-control error trapping mechanism* (`:Try :CatchIf :CatchAll :EndTry`) *instead.*

The monadic system function ⎕ERX is followed by a line number. Its effect is to set or clear error trapping. It takes a scalar integer argument which is a line number (or label) in the function. If an error occurs in the error trapped function, or functions called by the error trapped function, control is passed to the specified line. ⎕ERX returns the previous value set.

```
[2]  R←⎕ERX ERRLAB
 .
 .
[20] ERRLAB: 'HALTED, ERROR'
[21] 'ERROR WAS ',⎕R, ⎕ERM
```

To switch off error trapping, use an argument of 0.

⎕ES Error simulate

One-argument form

⎕ES can simulate a predetermined or programmer-defined error, in a form compatible with IBM's APL2. If the right argument is a character vector or scalar, ⎕ET is set to 0 1 and the right argument is used as the error message. A two element integer right argument will generate the appropriate error message (if the argument corresponds to a valid value for ⎕ET) or no error message (if there is no corresponding value for ⎕ET). An empty vector right argument causes no action to be taken, whilst a right argument of 0 0 resets error messages and codes (⎕EM ⎕ERM ⎕ET ⎕LER).

```
      ∇R←A DIVIDE B
[1]   ⎕ES(B=0)/'ATTEMPT TO DIVIDE BY ZERO'
[2]   R←A÷B
      ∇   1.43.05 05/28/90

      3 DIVIDE 0
ATTEMPT TO DIVIDE BY ZERO     (Message displayed)
      3 DIVIDE 0
      ^
      ⎕ET                     (⎕ET set to 0 1)
0 1
      ⎕LER                    (1↑⎕LER set to 15 - unknown error)
15 0

      ∇R←A DIVIDE B
[1]   ⎕ES(B=0)/5 4            (Signal the standard error)
[2]   R←A÷B
      ∇   1.44.32 05/28/90
      5 DIVIDE 0
DOMAIN ERROR                  (Standard message shown)
      5 DIVIDE 0
      ^
      ⎕ET                     (Standard values for ⎕ET, ⎕LER)
5 4
      ⎕LER
11 0
```

Using an argument to `⎕ES` that does not correspond to a error code of `⎕ET` causes no error message to display.

```
      ∇DOIT
[1]   ⎕ES 101 45              (Outside the ⎕ET range)
      ∇   1.45.48 05/28/90
      DOIT
      DOIT                    (No error message displayed)
      ^
      ⎕ET                     (⎕ET takes chosen value)
101 45
      ⎕LER                    (⎕LER takes ¯1 value)
¯1 0
```

Two-argument form

The two-argument form of `⎕ES` can be used to signal both an Error Type and an Error Message. The left argument should be a character scalar or vector which will be used as the error message portion of `⎕EM` and `⎕ERM` and the right argument should be a two element integer vector which will be assigned to `⎕ET.` The left argument overrides the usual error message associated with a given value of `⎕ET.` If the right argument is empty, no error is signalled. If it is 0 0 any left argument is ignored and error messages and reports are reset `(⎕ET ⎕LER ⎕EM ⎕ERM).`

```
      ∇R←A DIVIDE B
[1]   ⎕ES(B=0)/5 4
[2]   R←A÷B
      ∇   1.32.25 05/29/90
      3 DIVIDE 0              (Standard error message)
DOMAIN ERROR
      3 DIVIDE 0
      ^
      ⎕ET                     (Standard error codes)
5 4
      ⎕LER
11 0
      ∇R←A DIVIDE B
[1]   'ATTEMPT TO DIVIDE BY ZERO' ⎕ES(B=0)/5 4
[2]   R←A÷B                   (Redefined error message)
      ∇   1.33.27 05/29/90
      3 DIVIDE 0
ATTEMPT TO DIVIDE BY ZERO     (New error message)
      3 DIVIDE 0
      ^
      ⎕ET                     (Standard ⎕ET)
5 4
      ⎕LER
11 0
```

⎕ET Error Type

⎕ET returns a two element error type code generated by the latest error, either in calculator mode or within a defined function, in a format compatible with IBM's APL2. The earlier section on Error Handling contains a fuller discussion of the usage of ⎕ET as well as a table of the pre-assigned error types that ⎕ET can return.

```
      1000000⍴99
WS FULL
      1000000⍴99
      ^
      ⎕ET
1 3
      ⎕LER
1 0
```

⎕ET is implicitly localised in that it shows the error code which relates to the most currently pendent function on the SI stack.

See also ⎕ES (Error simulate) and ⎕LER (Line Error).

⎕EV Event Record

(Not implemented in APLX Server Editions)

Each callback property you can set corresponds to an APLX event. Just before your callback is called, the niladic system function ⎕EV is set to contain the event record, which gives further information about the event which is being reported. ⎕EV is an integer vector of length 9. The first five elements are common to all events, and the remaining four depend on the event type.

The elements of ⎕EV which are always supplied irrespective of the event type are (in index origin 1):

⎕EV[1]	Object tie number (the same as the tie property of the object)
⎕EV[2]	Event type number
⎕EV[3]	System clock time of event in milliseconds
⎕EV[4]	Mouse vertical position (in pixels) when event occurred
⎕EV[5]	Mouse horizontal position (in pixels) when event occurred

The remaining elements depend on the specific event type.

⎕EVA Event Arguments

The niladic system function ⎕EVA is a synonym for ⎕WARG. It is valid only inside a ⎕WE callback function, run by APLX as the result of an event occurring in one of your windows or other objects. It is used to pass data associated with the event from an external control or server, or another APL task. This data comprises the arguments passed when the event was created in the external or system object.

The exact content of ⎕EVA depends on the source of the event.

Use for APL Multi-Tasking ***(not available in Server Editions)***

When you create Child tasks using the APL object under ⎕WI, the Child and Parent tasks can each trigger events for the other task. ⎕EVA is used to pass data associated with the event, as follows:

- When the Child task is about to execute a command or expression, an onExecute event is triggered in the Parent's task object. When the callback function runs, ⎕EVA contains the command or expression which the Child task is about to execute, as a simple character vector.
- When an untrapped error occurs during function execution in the Child task, an onError event is triggered in the Parent's task object. When the callback function runs, ⎕EVA contains the error message in the same form as ⎕ERM.
- The Child and Parent tasks can each explicitly send an event to the other by using the Signal method. This takes any APLX array or overlay as an argument. When the onSignal callback in the receiving task runs, ⎕EVA contains the array or overlay which the sending task specified.

Use for OCX/ActiveX controls and OLE Automation ***(Windows only)***

⎕EVA allows your APL application to examine the arguments which an external caller (for example an OCX control or OLE server) has passed to APL as part of the underlying event-handling which occurs when an event is triggered. Typically, it is a simple or nested vector, with one element per argument passed by the control. If there are no arguments it is an empty vector. You can use the Events tab of the APLX Control Browser to see what events are associated with a control, and the parameters it passes.

For example, the Formula One spreadsheet-control allows you to specify a callback, to be triggered when the user clicks in a cell. The event name is onXClick, and the parameters are defined as:

```
void Click (long nRow, long nCol);
```

This means that it passes two integer parameters, which are the row and column of the cell in which the user has clicked. In your APLX callback function you can retrieve this information; ⎕EVA will return a two-element vector with these two values.

Use in .Net programming *(Windows only)*

When a .Net event for which you have defined an APL callback is triggered, `⎕EVA` contains a reference to the event argument object. This is an instance of the general .Net class `System.EventArgs`, or to a more specific descendant (such as `System.Windows.Forms.MouseEventArgs`), which provides information about the event. By examining the properties of this object, you can extract information about the event, such as the position of the mouse when the event was triggered.

⎕EVAL Evaluate external expression

The dyadic system function `⎕EVAL` allows you to evaluate an arbitrary expression in an external object-oriented environment, provided the architecture supports it.

The left argument is a character vector which specifies the external environment in which you want to evaluate the expression, in the same format as for `⎕NEW`. The right argument is a character vector containing an expression which is valid in the target environment.

The main use for `⎕EVAL` is for running code is an interpreted language such as Ruby or R, and for setting up variables in the Ruby environment:

```
      'ruby' ⎕EVAL 's=String.new "Hello there"'
Hello there
      'ruby' ⎕EVAL 's.length'
11
      'ruby' ⎕EVAL 'Math.sqrt(9)'
3
```

This example shows in the R environment:

```
      r←'r' ⎕new 'r'
      r.x←2 3ρι6         ⍝ x is an R variable
      r.x
1 2 3
4 5 6

      'r' ⎕eval 'x[2,]'
4 5 6
      'r' ⎕eval 'mean(x[2,])'
5
```

Note that the last line could be executed using the alternative syntax:

```
      r.⎕eval 'mean(x[2,])'
5
```

`⎕EVAL` is not supported for .Net or Java:

```
      '.net' ⎕EVAL '2+2'
Library for architecture '.net' does not support direct evaluation
DOMAIN ERROR
      '.net' ⎕EVAL '2+2'
      ^
```

See also ⎕CALL, which allows you to call arbitrary 'static' methods in external environments, and the system-method form of ⎕EVAL for R.

⎕EVN Event Name

The niladic system function ⎕EVN is valid only when you are running a ⎕WE callback function in response to an event. It contains the name of the event as a character vector. For .Net, this will be one of event property names, such as 'Closed' for a Form, or 'Click' for a Button. For System classes, it will be one of the callback names beginning with 'on'.

⎕EVT Event Target

The niladic system function ⎕EVT is valid only when you are running a ⎕WE callback function in response to an event. It returns a scalar object reference, which is a reference to the object which generated the event. For example, if a callback is running in response to a Click event for a .Net Button object, ⎕EVT will contain a reference to the Button.

If no event target is available, it will return a reference to the Null object.

⎕EX Expunge

The monadic system function ⎕EX causes named objects to be erased. See also)ERASE.

The right argument is a character vector (for a single name) or matrix of names. Expunge returns a vector of numbers in which 1 indicates successful erasure, 0 non-erasure:

```
      )VARS
DATA RESULT
      )FNS
AVERAGE MEAN    SD
      ⎕EX ⎕BOX 'MEAN STANDARD DATA'
1 0 1
      )VARS
RESULT
      )FNS
AVERAGE SD
```

Local objects are expunged if both local and global objects of the same name exist. This contrasts with)ERASE which erases the global version.

⎕EX can be used to erase a class definition (and all the methods and properties defined in it). Any instances of the class will become instances of the erased class's parent, if there is one, or of the NULL class, if the erased class did not have a parent. Similarly, any classes which inherited from the erased class will be re-parented so that they now inherit from the erased class's parent.

In this example, class POINT3D inherits from COLOR_POINT which in turn inherits from POINT. PT is an instance of COLOR_POINT:

```
      )CLASSES
COLOR_POINT    POINT   POINT3D
      ⎕CLASS POINT3D
{POINT3D} {COLOR_POINT} {POINT}
      PT←⎕NEW COLOR_POINT
      PT.⎕CLASSNAME
COLOR_POINT
```

If we erase the class COLOR_POINT, its child class POINT3D is re-parented. The instance PT becomes an instance of the original parent:

```
      ⎕EX 'COLOR_POINT'
1
      ⎕CLASS POINT3D
{POINT3D} {POINT}
      PT.⎕CLASSNAME
POINT
```

If we now erase the class POINT, POINT3D will now have no parent, and the instance PT becomes an instance of the NULL class:

```
      ⎕EX 'POINT'
1
      PT.⎕CLASSNAME
NULL
      ⎕CLASS POINT3D
{POINT3D}
```

⎕EXPORT Export APL array to file in specified format

The dyadic system function ⎕EXPORT exports an APL array to a file, in a format which can be read by other (non-APL) applications. (See also ⎕IMPORT which allows you to import data from a file of specified format).

The left argument is the array of data you want to export. The right argument determines the name of the file to be created, and the format of the file. If the right argument is a character vector, it is interpreted as the name of the file you want to create (including full path if required) and the format of the file is inferred from the file extension. If the right argument is a two element nested vector, the first element is the filename (or full pathname), and the second is a text string specifying the file type. File types are case-insensitive.

For example, the two following statements are equivalent, and will export the contents of the array BUDGET in 'comma-separated variables' ('CSV') format:

```
      BUDGET ⎕EXPORT 'Budget2007.csv'
      BUDGET ⎕EXPORT 'Budget2007.csv' 'csv'
```

The following file formats are supported, with the restrictions shown on the type and ranks of the data which can be exported in that format:

File type/extension	***Description***	***Restrictions***
`'txt'`	Text representation of the array (same as monadic format), with characters represented in 8-bit extended ASCII form. For matrices and higher-rank arrays, lines of text will be separated by the appropriate newline character for the platform (CR-LF on Windows, LF on Linux and MacOS).	All arrays supported, subject to available memory.
`'utf16'` or `'utf-16'`	Same as 'txt', with characters represented in 16-bit UTF-16 Unicode form (2 bytes per character).	All arrays supported, subject to available memory.
`'utf8'` or `'utf-8'`	Same as 'txt', with characters represented in the 8-bit UTF-8 Unicode form (variable number of bytes per character).	All arrays supported, subject to available memory.
`'csv'`	'Comma-separated variables' format, as used by many applications such as spreadsheets for data exchange. The file comprises one line of text per row of the data, with individual elements separated by commas. Numeric elements are expressed in text form. Text elements are surrounded by double-quotation marks; however, when importing CSV, many applications will ignore the quotation marks and treat the element as numeric if the string could be interpreted as a valid number.	The maximum rank of the array is 2. The array can be nested, but any elements other than scalars and character vectors will be represented by their monadic format. The array must not contain object references. Characters are output in 8-bit (extended ASCII) form.
`'tsv'`	'Tab-separated variables' format. Same as CSV, except the fields are separated by tab characters instead of commas.	Same as for CSV.
`'htm'` or `'html'`	HTML format. The result is an HTML page which can be loaded into a Web Browser or imported into another application such as Microsoft Word or Excel. The page will contain an HTML table containing the data, unless the array is a simple character array, in which case it is output as a paragraph with <BR> between lines.	The maximum rank of the array is 2. The array can be nested, but any elements other than scalars and character vectors will be represented by their monadic format. The array must not contain object references. All printable characers in `⎕AV` can be output (non-ASCII characters will be output in escaped Unicode form).

`'slk'`	Symbolic Link (SYLK) format, a Microsoft-specified file format typically used to exchange data between applications such as Excel and other spreadsheets.	The maximum rank of the array is 2. The array can be nested, but any elements other than scalars and character vectors will be represented by their monadic format. The array must not contain object references. Only ASCII characters, plus some other specific characters such as accented letters, can be represented. APL symbols and other non-representable characters will be replaced by `'?'`.

`'xml'`	Extensible Markup Language (XML) format, a format used for saving structured data with markup information.	The left argument must be an APL array with the same specification as `⎕XML`. The data is written as UTF-8 encoded XML text. This conversion is equivalent to the two-stage command:
		
		`      (⎕XML array) ⎕EXPORT 'filename' 'utf8'`
		
		In order to ensure that the XML generated is valid, `⎕EXPORT` will add the following XML prologue if the APL array does not contain one:
		
		`      <?xml version="1.0" encoding="utf-8"?>`

Note that, because the intention is to allow APLX to exchange data with other applications, `⎕EXPORT` translates APL high minus (`¯`) to ASCII or Unicode minus (`-`).

For example, suppose that BUDGET is a 4-row matrix of text vectors and numbers as follows:

```
     BUDGET
             Q1    Q2    Q3    Q4
 Sales    11300 13220 16550 19230
 Expenses 12450 12950 13640 13980
 Profit   ¯1150   270  2910  5250
      ⎕DISPLAY BUDGET
┌→───────────────────────────────────┐
↓ ┌⊖┐        ┌→─┐  ┌→─┐  ┌→─┐  ┌→─┐  │
│ │ │        │Q1│  │Q2│  │Q3│  │Q4│  │
│ └─┘        └──┘  └──┘  └──┘  └──┘  │
│                                    │
│ ┌→────┐                            │
│ │Sales│    11300 13220 16550 19230 │
│ └─────┘                            │
│                                    │
│ ┌→───────┐                         │
│ │Expenses│ 12450 12950 13640 13980 │
│ └────────┘                         │
│                                    │
│ ┌→─────┐                           │
│ │Profit│   ¯1150 270   2910  5250  │
│ └──────┘                           │
│                                    │
└∊───────────────────────────────────┘
```

This array can be exported in CSV format as follows:

```
      BUDGET ⎕EXPORT 'Budget2007.csv'
```

The contents of the file 'Budget2007.csv' will be:

```
"","Q1","Q2","Q3","Q4"
"Sales",11300,13220,16550,19230
"Expenses",12450,12950,13640,13980
"Profit",-1150,270,2910,5250
```

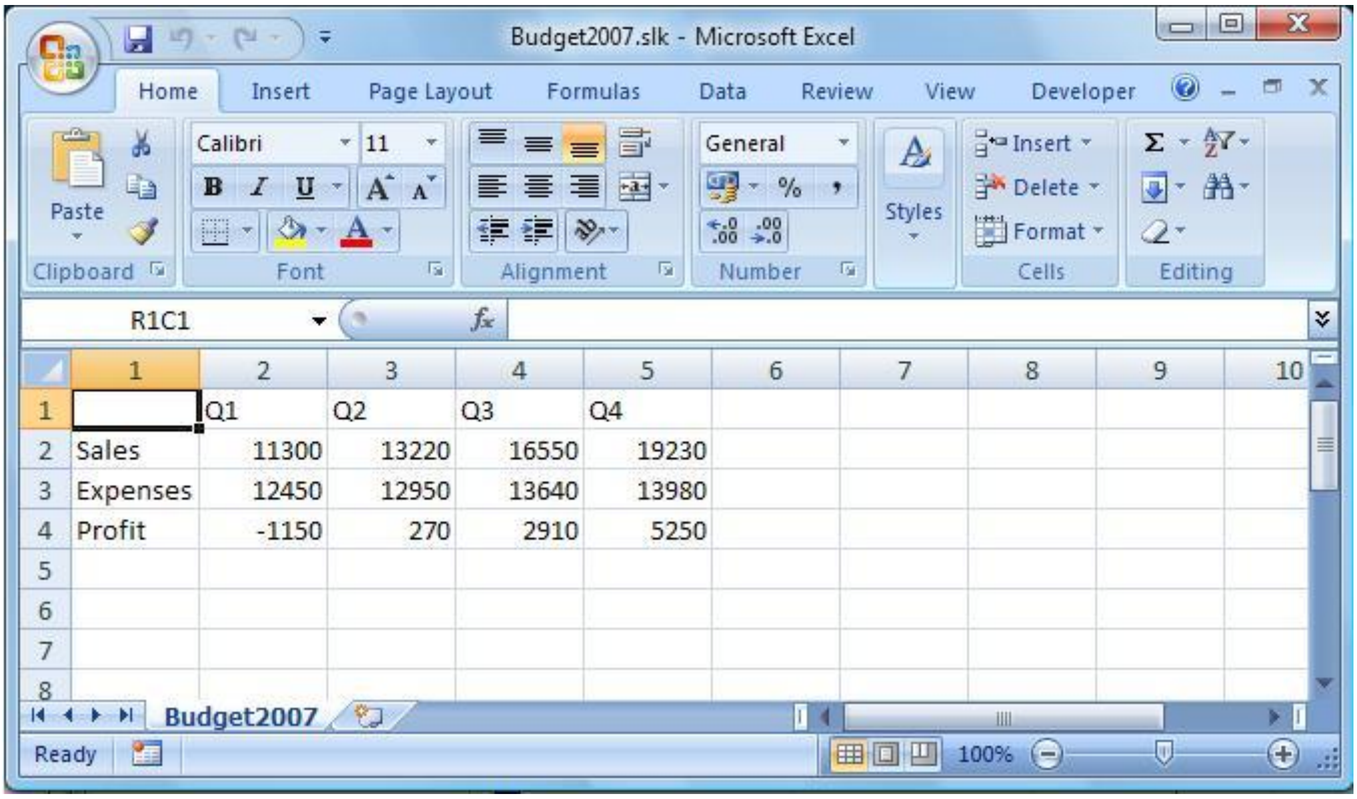
Alternatively, the same data can be exported in SYLK format:

```
      BUDGET ⎕EXPORT 'Budget2007.slk'
```

The contents of the file 'Budget2007.slk' will contain the data in SYLK format:

```
ID;PAPLX;N;E
P;PGeneral
B;Y4;X5
C;Y1;X1;K""
C;X2;K"Q1"
C;X3;K"Q2"
C;X4;K"Q3"
C;X5;K"Q4"
C;Y2;X1;K"Sales"
C;X2;K11300
C;X3;K13220
C;X4;K16550
C;X5;K19230
C;Y3;X1;K"Expenses"
C;X2;K12450
C;X3;K12950
C;X4;K13640
C;X5;K13980
C;Y4;X1;K"Profit"
C;X2;K-1150
C;X3;K270
C;X4;K2910
C;X5;K5250
E
```

Either format can be directly opened by another application such as Excel:



	1	2	3	4	5
1		Q1	Q2	Q3	Q4
2	Sales	11300	13220	16550	19230
3	Expenses	12450	12950	13640	13980
4	Profit	-1150	270	2910	5250

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine. Typically, the Client will be the APLX front-end built as a 32-bit Windows application running on a desktop PC, and the Server will be a 64-bit APLX64 interpreter running on a 64-bit Linux or Windows server.

In such systems, you can specify whether the file should be accessed on the Client or the Server machine. You do this by preceding the file name with either an Up Arrow ↑ to indicate that the file should be accessed on the Client, or a Down Arrow ↓ to indicate that it should be accessed on the Server. If you do not specify, the default is that the access takes place on the Client.

⎕FAPPEND Append component to file

The ⎕FAPPEND function appends a new component to the file, returning the component number used. The syntax is:

```
        R ← DATA ⎕FAPPEND TIENO {PASS}
```

DATA is any APL array or an overlay created using ⎕OV. TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS.

The explicit result is the component number to which the data was written (i.e. the highest existing component number plus 1).

The effect of ⎕FAPPEND is similar to ⎕FWRITE with a component number of 0 (or omitted).

⎕FC Format Control

The system variable ⎕FC contains six characters used by the primitive format function ⍕. The characters are used as follows:

```
Character     Default          Usage
    1            .             Decimal point character
    2            ,             Thousands indicator
    3            *             Fill for blanks
    4            0             Fill  for overflows (the default 0  causes a
                               DOMAIN ERROR)
    5            _             Print as blank (cannot be ,.0123456789)
    6            ¯             Negative number indicator
```

For details of the way in which ⎕FC is used, see the entries for ⍕ (Format by Specification and Format by example).

⎕FCREATE Create a new component file

The ⎕FCREATE function creates a new component file, and leaves it tied. The syntax is:

```
FILENAME ⎕FCREATE TIENO
```

FILENAME is a character vector specifying the name of the file to create. The name may be specified in either of two ways. If the name contains a directory-separator character (: / or \), it is treated as a full host path name, and you need to specify the file extension explicitly, usually .aqf. Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set using the preferences dialog or ⎕MOUNT), and the file extension .aqf is added automatically.

TIENO is an arbitrary non-zero integer to be used in subsequent read/write operations to identify the file (the tie is exclusive). The tie number must not currently be in use to tie another file. Alternatively, you can provide a tie number of 0, in which case APLX automatically allocates the next available unused tie number, and returns it as the explicit result of the function.

For example, suppose APLX is running on a Windows machine and the ⎕MOUNT table is set (either under program control or using the Preferences dialog) so that library 0 is in `c:\temp` and library 1 is in `m:\budget\current`:

```
      2 30↑⎕MOUNT ''
c:\temp
m:\budget\current
```

You could create a new file called `RUN3` in `c:\temp` as follows (no directory separator character appears in the name, so it is taken as a simple file name in library 0):

```
      'RUN3' ⎕FCREATE 2
```

(Using a left argument of `'0 RUN3'` would be equivalent). The file is created, and then exclusive-tied on tie number 2, so you can write to it immediately:

```
      ⎕FNUMS
2
      ⎕TS ⎕FWRITE 2
```

The full operating-system path would be `c:\temp\RUN3.aqf`.

In this second example, the user has specified 0 as the tie number, so APLX allocates and returns the next available tie number. The file is created in library 1, so the full operating-system path would be `m:\budget\current\RUN4.aqf`:

```
      '1 RUN4' ⎕FCREATE 0
3
      ⎕FNUMS
2 3
```

In this third example, a full path name has been supplied, with an explicit tie number of 8, so we now have three files tied:

```
      'c:\temp\RUN4.aqf' ⎕FCREATE 8
      ⎕FNUMS
2 3 8
      ⎕FNAMES
RUN3
1 RUN4
C:\TEMP\RUN4.aqf
```

(Under Linux or AIX, the full path name might be something like `/usr/tmp/RUN4.aqf`. Under MacOS, it might be something like `Macintosh HD:temp:RUN4.aqf`).

Note that, if you specify a full file name, you can use any file extension (or none). However, we recommend that you always use `.aqf` for APLX `⎕Fxxx` component files. If you do not use the `.aqf` extension, your component files will not show up in `⎕FLIB`, and you will not be able to access them using the library-relative syntax.

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine. The two parts of the application communicate via a TCP/IP network. Typically, the Client will be the APLX front-end built as a 32-bit Windows application running on a desktop PC, and the Server will be a 64-bit APLX64 interpreter running on a 64-bit Linux or Windows server.

In such systems, you can specify whether the file should be accessed on the Client or the Server machine. You do this by preceding the file name with either an Up Arrow ↑ to indicate that the file should be accessed on the Client, or a Down Arrow ↓ to indicate that it should be accessed on the Server. If you do not specify, the default is that the access takes place on the Client. This is true either if you specify the full path name in the `⎕FCREATE` call, or via the `⎕MOUNT` table.

Mixing 32-bit and 64-bit Component Files

If you are running both 32-bit and 64-bit versions of APLX, then it is possible to share component files between the two architectures, but there are some special points you should be aware of. See the introduction to the `⎕Fxxx Component File System` for details.

`⎕FCSIZE` Read component size information

The `⎕FCSIZE` function returns information about the size of a component. The syntax is:

```
      R ← ⎕FCSIZE TIENO COMPONENT {PASS}
```

`TIENO` is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

The parameter `COMPONENT` is the component number. This must be an integer in the range *Lowest existing component number* to *Highest existing component number*.

The explicit result is a two-element integer vector:

1. The size the variable would use if read into the workspace.

2.The size of the slot allocated to the component in the file, excluding the component header.

Both are expressed in bytes.

⎕FDELETE Delete component from a file

The ⎕FDELETE function deletes a component from the file, renumbering the remaining components accordingly. The syntax is:

```
        ⎕FDELETE TIENO COMPONENT {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter COMPONENT is the component number you want to delete. This must be an integer in the range *Lowest current component number* to *Highest existing component number*. The component will be deleted, and any later components will be re-numbered so that they remain in integral sequence. For example, suppose the file currently has components numbered 1 to 5. If you delete component 3, then the old components 4 and 5 will be re-numbered 3 and 4 respectively. If you now delete component 1, the remaining components will be re-numbered 1 2 3 (corresponding to the original components 2 4 and 5).

See also the function ⎕FDROP which deletes components at the start or end of the file, but does not re-number the remaining components.

⎕FDROP Drop components from start or end of file

The ⎕FDROP function deletes one or more components from the start or end of the file, without renumbering the remaining components. The syntax is:

```
        ⎕FDROP TIENO N {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter N is a positive or negative integer. If it is positive, the first N components of the file are deleted. If it is negative, the last -N components are deleted. In both cases, existing components are not re-numbered.

For example, suppose you have a file with components numbered from 1 to 12. After executing the two statements:

```
        ⎕FDROP TIENO,2
        ⎕FDROP TIENO, ¯3
```

the first component in the file will be component 3, and the last will be component 9. The original components 1, 2, 10, 11, 12 will no longer exist. The original components 3 through 9 still exist, and retain the same component numbers.

See also the function `⎕FDELETE` which deletes a single component anywhere in the file, re-numbering the remaining components accordingly.

⎕FDUP Duplicate component file, reclaiming wasted space

The `⎕FDUP` function makes a copy of an existing tied component file. Component ownership and timestamp information is retained in the copy of the file. Because the file is copied component-by-component, any wasted space caused by fragmentation of the original file is not reflected in the copy. (You can tell how much space is wasted in a file by using `⎕FSIZE`.)

The syntax is:

```
      DESTNAME ⎕FDUP TIENO {PASS}
```

`DESTNAME` is a character vector specifying the name of the destination file. The name is specified in the same way as for `⎕FCREATE`. If the name contains a directory-separator character (: / or \), it is treated as a full host path name, and you need to specify the file extension, by convention `.aqf`. Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set the preferences dialog or `⎕MOUNT`), and the file extension `.aqf` is added automatically.

Note that `DESTNAME` must not be the same as the original file name. If you want to achieve the effect of reclaiming space whilst keeping the file name unchanged, you should first make a copy using a temporary file name, then erase the original and rename the temporary file back to the original name.

⎕FERASE Erase component file

The `⎕FERASE` function allows you to erase a component file from the host file system. You must have the correct host access permission to erase the file. If the file is currently tied it will be untied before attempting to erase it.

The usual syntax of `⎕FERASE` is :

```
      ⎕FERASE FILENAME
```

`FILENAME` is a character vector specifying the name of the file to erase. The name is specified in the same way as for `⎕FCREATE`. If the name contains a directory-separator character (: / or \), it is treated as a full host path name, and you need to specify the file extension, by convention `.aqf`. Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set the preferences dialog or `⎕MOUNT`), and the file extension `.aqf` is added automatically.

For compatibility with some other APL interpreters which require the file to be tied before it can be erased, an alternate syntax for `⎕FERASE` is supported :

```
      FILENAME ⎕FERASE TIENO
```

`⎕FERROR` Return operating-system error

When a component-file operation fails because the operating system reports an error, APLX usually generates a `FILE I/O ERROR` (error code `17` in `⎕LER`, or `6 9` in `⎕ET`). The niladic system function `⎕FERROR` returns a character vector with further information (if available) from the operating system about what caused the error. For example:

```
      ⎕FLIB 'C:\JIM\REGIONS'
The system cannot find the path specified.
FILE I/O ERROR
      ⎕FLIB 'C:\JIM\REGIONS'
      ^
      ⎕ET
6 9
     ⎕FERROR
The system cannot find the path specified.
```

`⎕FHOLD` Hold/Release component files for exclusive access

The function `⎕FHOLD` is used for synchronizing access when multiple users and/or tasks are reading or writing the same file or files. It takes a single argument, which is usually a vector of tie numbers. If you need to supply pass numbers, the argument is a 2-row matrix of tie number/pass number pairs. The tie numbers should correspond to files you have share-tied using `⎕FSTIE`, or should be an empty vector (meaning release all holds). The effect of `⎕FHOLD` is as follows:

Firstly any existing file-holds belonging to this APL task are released. Secondly the system attempts to secure an exclusive lock on all the file numbers you have specified in the argument. Only one task can hold a given file at any time. If any of the files you try to hold are already held by another task, the operation waits until all files are available.

All file holds are automatically released when the APLX task reaches desk-calculator mode (this means you cannot experiment with `⎕FHOLD` in desk-calculator mode), or when the APL task ends. For best performance in multi-user or multi-tasking applications, you should attempt to minimize the amount of time that your application holds files for exclusive use.

Technical Notes:

`⎕FHOLD` is protected against deadlock. Supposing Task 1 tries to lock files A and B, and at the same time Task 2 tries to lock files B and A. You could get a situation where each task successfully locks the first file it tries, and then both tasks wait for ever for the second to be unlocked. APLX detects this situation, and automatically corrects for it by backing off (i.e. releasing any locks), waiting a random period, and then trying again.

`⎕FHOLD` uses operating-system locks. Where you have files accessed over a network, some network systems do not honor locks correctly, particularly when mixing clients of different types (e.g. Windows and Linux tasks). We recommend testing the effect of `⎕FHOLD` when implementing multi-user or multi-tasking file-systems across networks.

`⎕FHOLD` may do nothing on some single-user versions of APLX, for example *APLX Personal Edition*.

⎕FI Convert formatted input

The monadic system function `⎕FI` is used in conjunction with `⎕VI` to validate a text string and convert it to numeric form. In both cases, the argument is a character vector (or scalar), containing one or more sub-strings of characters separated by blanks. For each non-blank sub-string, `⎕VI` returns a 1 if the sub-string represents a valid number, and 0 if it does not. `⎕FI` returns the numeric representation of the sub-string, or 0 if it is not a valid number. Numbers are validated and converted in the same way as normal APL input. Scientific notation is supported, and negative numbers are prefixed by the high minus (¯) character. For example:

```
      ⎕FI '100.32 $4 2,,3 0 12.2 ¯3 +2 -2'
100.32 0 0 0 12.2 ¯3 0 0
      ⎕VI '100.32 $4 2,,3 0 12.2 ¯3 +2 -2'
1 0 0 1 1 1 0 0
```

`⎕VI` and `⎕FI` are usually used to validate and convert user input, or to convert text files to numeric form. To allow users to enter negative numbers using the ordinary (non-APL) minus sign, you can use `⎕SS` to translate minus to high-minus first. To allow comma-delimited input, use `⎕SS` to translate comma to space:

```
      STRING←'3,-2,45.5,-0.08'
      ⎕VI STRING
0
      ⎕SS (STRING; ('-';','); ('¯';' '))
3 ¯2 45.5 ¯0.08
      ⎕VI ⎕SS (STRING; ('-';','); ('¯';' '))
1 1 1 1
      ⎕FI ⎕SS (STRING; ('-';','); ('¯';' '))
3 ¯2 45.5 ¯0.08
```

⎕FLIB Return names of component files in directory

⎕FLIB returns a character matrix of the names of the component files in a particular directory. Names are padded to the right with blanks as necessary. It takes a single argument, which can be either a library number (usually 0 to 9, corresponding to the rows of the ⎕MOUNT table), or a character string representing an operating-system path.

For example, under Windows you might have:

```
      ⎕FLIB 1
RUN4
JIM
COPY
      ⎕FLIB 'C:\TEMP'
RUN3
```

Only files which end in the .aqf extension will appear in the list. The extension is stripped from the names of the files returned.

See also ⎕LIB which returns the names of all files in a directory.

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine.

In such systems, you can specify whether the directory being searched is on the Client or the Server machine. You do this by preceding the path name with either an Up Arrow ↑ to indicate that the directory is on the Client, or a Down Arrow ↓ to indicate that it is on the Server. (If you do not specify, the default is that the access takes place on the Client.) This is true either if you specify the full path name in the ⎕FLIB call, or via the ⎕MOUNT table.

⎕FMT Formatting Function

(See also ∝ (Picture format) and ⍕ (Format by specification and Format by example) for other formatting functions.)

⎕FMT is a FORTRAN-like formatter. The ⎕FMT function takes the form:

```
                A ⎕FMT B
```

and converts the elements of an object B which may be a scalar, vector or matrix (2≥⍴⍴B) to a character representation of B based on the specifications of A, which consist of phrase types, qualifiers and decorators, as summarised below. If B is a vector it is treated as a 1 column matrix, if it is a scalar it is treated as if it were a 1 by 1 matrix. B must be a simple array.

The left argument, A, is a character vector consisting of one or more phrases, each phrase separated from the previous phrase by a comma. Spacing within the phrase does not matter unless it is within two text delimiters.

The phrases are processed from left to right, with each phrase determining the format of a field in the result. A phrase may be enclosed by a single pair of parentheses, but nested parentheses or parentheses which enclose more than one phrase, are not allowed. If 'xxxx' represents a phrase, then 'xxxx,(xxxx)' is valid but '(xxxx,xxxx)' is not. Each phrase, possibly preceded by a repetition factor, consists of 2 basic parts in the following order:

```
            i       qualifiers and decorators
            ii      phrase identifier with associated parameters
```

The repetition factor is optional and if omitted defaults to 1. If specified, the repetition factor must be first and may not be enclosed in parentheses. It is used to indicate how often a phrase is to be repeated before the system uses the next one. 3xxxx,yyy means 'repeat phrase 'xxxx' 3 times, then use phrase 'yyy'. A repetition of 0 is not allowed.

`⎕FMT` will also repeat phrases, wrapping around to the first phrase again if necessary, until all columns in the right argument have been processed.

Delimiters

Delimiters are used to insert text, or a pattern into the result of `⎕FMT`. Delimiters may be used by themselves, in conjunction with picture formatting (G) or with decorators (R,M,N,P,Q,S). The following pairs of delimiters are used to delimit literal strings, picture patterns and decorators:

```
             Left delimiter    Right delimiter

                    ⎕                 ⎕
                    ⍞                 ⍞
                    <                 >
                    ⊂                 ⊃
                    ¨                 ¨
```

For readability, you are recommended to reserve the `'⎕ ⎕'` delimiters for decorators and the `'< >'` delimiters for strings and pictures.

```
           '⎕ THIS IS A <LITERAL> ⎕,I5' ⎕FMT 10
      THIS IS A <LITERAL>   10
```

Phrase types

```
    A   -   character format
    E   -   exponential format
    F   -   fixed point format
    G   -   picture format
    I   -   integer format
    T   -   place the result of subsequent phrases following rightmost column
```

```
Tn  -   place the result of subsequent phrases starting in absolute column
        n
Xn  -   offset the result of subsequent phrases by n columns from current
        location
```

A - character format

Syntax: An n is the total field width in the result

Place a character from the current column of the right argument in a right- justified field of width n. The width specification is usually 1 and it almost always appears with a replication factor equal to the column dimension of the character matrix being formatted. If the right argument is not character, then an overflow condition exists (the * character).

Qualifiers: None
Decorators: R
Substitutable Characters: * in overflow

```
      '4A1,A3'⎕FMT 2 5⍴'ABCDEFGHIJ
ABCD  E
FGHI  J
      'A4'⎕FMT 4 6            (A vector is treated as a 1 column
****                           matrix, and overflow occurs with a
****                           numeric argument.)
```

E - exponential format

Syntax: En.m n is the total field width in the result, m is the number of significant digits to be displayed.

Format a number from the current column in the right argument using the exponential form. If the field width is 0, then the field is empty. Following the last digit of the mantissa, 5 positions are reserved for the letter E and the exponent. The exponent is left justified and consists of a minus sign (if required) and up to 3 digits which represent the magnitude of the exponent. If the m value is 1 then the . is omitted, and if m is 0 then the mantissa is omitted.

Qualifiers: B,C,K,L,Z Decorators: M,N,P,Q,R,S Substitutable Characters: E in the exponent, . leading zeros, the ¯ in the exponent and the * for overflow.

```
      'E11.4'⎕FMT 100  7.6978E23 ¯.004
 1.000E2
 7.698E23
¯4.000E¯3

      'E8.1,E8.0'⎕FMT 1 2⍴16 .02
2E1     E¯2
```

F - fixed point format

Syntax: FN.m n is the total field width in the result, m is the number of digits to be displayed after the decimal point.

Format a number from the current column of the right argument using the fixed point form. If the field width is 0, then the field is empty. If m is 0, then the decimal point is not suppressed.

Qualifiers: B,C,K,L,Z Decorators: M,N,P,Q,R,S Substitutable Characters: The . leading 0s, commas and the `*` for overflow.

```
      'CF10.2'⎕FMT 98.672 ¯12796.497
     98.67
¯12,796.50
      'S<,..,>CF10.2'⎕FMT 98.672 ¯12796.497
     98,67
¯12.796,50
      'F4.0'⎕FMT 6
6.
```

G - Picture format

Syntax: Gtext text is a COBOL-like 'picture' format statement within a pair of delimiters.

The text consists of Zs, 9s, and any other characters. A Z will become a digit only if that digit is significant, otherwise it is replaced in the result by a blank. A 9 is replaced with a digit only if that digit is significant, otherwise it appears as a 0. A text string with Zs and 9s on both sides which has a Z on either side is displayed only if the Z becomes a significant digit in the result. Periods are treated like any other character. Numbers are rounded to the nearest integer before formatting. Fractional digits can be formatted by using the K qualifier and putting the appropriate number of Z's and 9's after the decimal point in the picture.

Qualifiers: K,B (Z,C,L will cause a DOMAIN ERROR) Decorators: M,N,P,Q,R,S Substitutable characters: Leading 0's, the overflow `*`, and the Z's and 9's used in the picture

```
      'G<DATE IS 99/99/9999>'⎕FMT 08031985
DATE IS 08/03/1985
      'G<TEL: (999) 999-9999>' ⎕FMT 0719228866
TEL: (071) 922-8866
     'B K2 G< ZZZ POUNDS AND ZZ PENCE>' ⎕FMT 8.23 12.86 0 2.52
 8 POUNDS AND 23 PENCE
12 POUNDS AND 86 PENCE

 2 POUNDS AND 52 PENCE
```

I - integer format

Syntax: In n is the total field width in the result.

Format a number from the current column in the right argument as an integer after rounding. A field width of 0 will cause the field to be empty.

Qualifiers: B,C,K,L,Z
Decorators: M,N,P,Q,R,S
Substitutable characters: Leading 0s, commas

```
          'I4' ⎕FMT ⍳4
        1
        2
        3
        4
          'I6,I3,CI6' ⎕FMT 1 3⍴12 7896 48723.4
        12***48,723
          'I3'⎕FMT 2.6792
    3
```

T - move to absolute position

Syntax: Tn n is the absolute position, reckoned from the left margin. or T

Move to print position n. If T alone is used the cursor or print-element is moved to the 'first free' position to the right. The latter form is used only in very complicated formatting exercises involving backspacing.

NB: This operation does not require a column of data in the right argument.

Qualifiers: None
Decorators: None
Substitutable characters: None

```
          'I3,T3,I2' ⎕FMT 1 2⍴678 92
    6792
          'I10,T1,I3,T,I2'⎕FMT  1 3⍴789 672 12
    672    78912
```

X - move to relative position

Syntax: Xn n is the number of positions the cursor or print-head is to be moved relative to the current position. n may be positive (right move) or negative (left move).

If m is current print position: move to print position `m+n`.

NB: This operation does not require a column of data in the left margin.

Qualifiers: None
Decorators: None
Substitutable characters: None

```
          'I3,X2,I2' ⎕FMT 1 2 ⍴ 789 67
    789  67
       'I3,X¯1,I2' ⎕FMT 1 2 ⍴789 67
    7867
```

Qualifiers and Decorators

Qualifiers and decorators are used with the format codes to provide further tailoring to specific needs. All qualifiers and decorators come before the format codes and after the replication factor (if supplied).

Qualifiers:

B	-	blank all zero results. Allow background text to show through.
C	-	put commas after each third digit to the left of the decimal point
Kn	-	multiply the right argument by 10 to the power n before formatting.
L	-	left justify the result field using blanks to fill in positions to the right of the right-most formatted digit. This does not allow background text to show through unless the B qualifier is used and the value being formatted is 0.
Z	-	put in leading zeros. If the C qualifier is used, however, commas will not be inserted between any of the leading zeros.

Decorators:

M⎕TEXT⎕	-	put TEXT immediately in front of negative numbers. If the Z qualifier is being used, the text will begin at the left of the field and overlay as many leading zeros as there are characters in TEXT.
N⎕TEXT⎕	-	put TEXT immediately after negative numbers.
P⎕TEXT⎕	-	put TEXT before non-negative numbers (otherwise as M)
Q⎕TEXT⎕	-	put TEXT after non-negative numbers (otherwise as N)
R⎕TEXT⎕	-	put in background TEXT to fill any unused positions in the field (eg. In cheque protection).
S⎕TEXT⎕	-	substitute standard characters. TEXT must consist of pairs of characters, where the second of each pair is to replace all occurrences of the first.

```
      A
B K2 G< ZZ9 DOLLARS AND 99 CENTS>
      A ⎕FMT 8.23 12.86 0 2.52
 8 DOLLARS AND 23 CENTS
12 DOLLARS AND 86 CENTS

 2 DOLLARS AND 52 CENTS
      B
<TOTAL >,C M⎕(⎕ N⎕)⎕ Q⎕ ⎕ I8,X2,CF 12.2,X2,ZE12.3
      C
   100            1321321.2           ¯12332.7
¯19232.563           ¯232                212.219
      B ⎕FMT C
TOTAL      100    1,321,321.20   ¯001.23E4
TOTAL ( 19,233)        ¯232.00   0002.12E2
```

Literals

In the last example, the literal string 'TOTAL ' was placed between < > delimiters to insert characters into the result. Note that no corresponding column was required in the right argument. When ⎕FMT scans its left argument it moves from left to right. If a left delimiter is found, all text up to the first corresponding right delimiter, including commas and other delimiters, is treated as a literal:

```
             '⎕THIS IS AN <INTEGER>:⎕, I5' ⎕FMT 47
       THIS IS AN <INTEGER>:   47
```

⎕FNAMES Return names of currently-tied files

The niladic function ⎕FNAMES returns a character matrix of the names of the component files currently tied by this APL task, in the same order as the tie numbers returned by ⎕FNUMS. Names are padded to the right with blanks as necessary. The names are returned in the same form in which they were tied.

For example, under Linux you might have:

```
      'RUN3' ⎕FTIE 2
      '1 RUN4' ⎕FTIE 0
3
      '/home/jim/RUN5.aqf' ⎕FTIE 8
      ⎕FNUMS
2 3 8
      ⎕FNAMES
RUN3
1 RUN4
/home/jim/RUN5.aqf
```

⎕FNUMS Return tie numbers in use

The niladic function ⎕FNUMS returns an integer vector of the component-file tie numbers currently in use by this APL task.

See ⎕FNAMES for more information.

⎕FRDAC Read component-file access matrix

The ⎕FRDAC function returns the access matrix for a file. The syntax is:

```
      R ← ⎕FRDAC TIENO {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The explicit result is the file's current N by 3 access matrix (which may be empty). See the description of ⎕FSTAC for details.

⎕FRDCI Read component information

The ⎕FRDCI function returns information about a component. The syntax is:

```
      R ← ⎕FRDCI TIENO COMPONENT {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter COMPONENT is the component number. This must be an integer in the range *Lowest existing component number* to *Highest existing component number*.

The explicit result is a three-element integer vector:

1. The free workspace required to read the component

2. The account number (1↑⎕AI) of the user who last wrote the component.

3. The time at which the component was last written, expressed in seconds since the start of the millennium (00:00:00, 1 Jan 2000).

⎕FRDFI Read file information

The ⎕FRDFI function returns information about the creation and last update of a file. The syntax is:

```
      R ← ⎕FRDFI TIENO {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The explicit result is a 4 by 2 array of information about the file. The first column represents user numbers, and the second timestamps, expressed as seconds since the start of the millennium (00:00:00, 1 Jan 2000). The rows are:

1. User who created file, Timestamp file was created

2. Reserved (returns ¯1 ¯1)

3. Reserved (returns ¯1 ¯1)

4. User who last updated file, Timestamp file was last updated.

⎕FREAD Read component from a file

The ⎕FREAD function reads a component from the file. The syntax is:

```
      R ← ⎕FREAD TIENO COMPONENT {PASS}
```

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

The parameter COMPONENT is the component number you want to read. This must be an integer in the range *Lowest existing component number* to *Highest existing component number*.

The explicit result is the data (array or overlay) last written to that component by any user.

If the component number does not exist, APLX will generate an error COMPONENT NOT IN FILE (error code 20 in ⎕LER, or 6 3 in ⎕ET).

⎕FRENAME Rename component file

The ⎕FRENAME function allows you to rename a component file. You must have the correct host access permission to rename the file. If the file is currently tied, the name change will be reflected in the name list returned by ⎕FNAMES.

Two alternative syntax forms are supported for ⎕FRENAME:

```
 NEWNAME ⎕FRENAME OLDNAME
```

or

```
 NEWNAME ⎕FRENAME TIENO
```

OLDNAME is a character vector specifying the file to rename. The name is specified in the same way as for ⎕FCREATE. If the name contains a directory-separator character (: / or \), it is treated as a full host path name (including the file extension, by convention .aqf). Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is created in the directory 0 to 9 specified (the actual path is set the preferences dialog or ⎕MOUNT), and the file extension .aqf is added automatically.

NEWNAME is a character vector specifying the new name of the file, specified in the same way.

The second syntax form is an alternative way to rename a file which has already been tied. TIENO is the tie number on which the file is tied.

Note that the host operating system may not allow you to rename a file across different file systems or physical disks.

⎕FREPLACE Replace existing component

The ⎕FREPLACE function writes new data to an existing component in the file, or appends to the file. The syntax is:

```
      DATA ⎕FREPLACE TIENO COMPONENT {PASS}
```

DATA is any APL array or an overlay created using ⎕OV. TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS.

The parameter `COMPONENT` is the component number at which you want to write the new data. This must be an integer in the range *Lowest current component number* to *Highest existing component number plus 1*.

The effect of `⎕FREPLACE` is similar to that of `⎕FWRITE` with an integral non-zero component number.

`⎕FRESIZE` Set maximum file size

The `⎕FRESIZE` function allows you to specify a maximum size for a component file. The syntax is:

```
      NEWSIZE ⎕FRESIZE TIENO {PASS}
```

`TIENO`is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

The left argument `NEWSIZE` is the maximum size (in bytes) to which the file will be allowed to grow. If `NEWSIZE` is 0, the file size is not limited by APLX, although the maximum size is still subject to any operating-system imposed limit and available disk space.

If you try to write to a file in a way which would cause the file size to exceed the limit, APLX will generate the error `FILE ALLOCATION EXCEEDED`.

The default is 0, meaning there is no limit.

`⎕FSIZE` Read file-size and component-range information

The `⎕FSIZE` function returns information about the size of a file and the range of its components. The syntax is:

```
      R ← ⎕FSIZE TIENO {PASS}
```

`TIENO` is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the `PASS` parameter.

The explicit result is a five-element integer vector:

1. The number of the first component in the file

2. The next available component number (i.e. current highest + 1)

3. The file size currently used (including unused space), in bytes

4. The file size limit (set using ⎕FRESIZE) in bytes; 0 means unlimited

5. The approximate number of bytes of unused space that would be reclaimed by ⎕FDUP

As you write components to a file, and delete existing components, APLX attempts to make the freed-up space available for future use. However, in some cases the file may become fragmented, and the amount of unused space may increase over time. The fifth element of the result of ⎕FSIZE is useful in deciding whether to make a clean copy of the file (using ⎕FDUP), thus reclaiming the space.

⎕FSTAC Set component-file access matrix

In multi-user versions of APLX, each user can be allocated a unique user number (shown by 1↑⎕AI). Individual APLX component files are tagged with a User Number, and have an associated File Access Matrix which indicates which users can access the file, what operations they may perform, and whether they need to specify a pass number to tie the file. Users will be allocated their user number by the logon procedure adopted by their system. Each user can thus 'own' a number of files, and can grant or deny access to these files.

You can change the Access Matrix for a file using the syntax:

```
      MATRIX ⎕FSTAC TIENO {PASS}
```

The Access Matrix is three columns wide. The first column is a list of user numbers, with 0 being taken to mean ALL users. The second column is a list of integers which indicate the access privileges for the indicated user. The third column is the list of pass numbers which must be used by the given user to access the file with those rights (0 means no pass number is required). The access matrix may have a maximum of 19 rows. When a file is created, the default access matrix allows only the owner to access it, and grants the owner all rights, with no pass number.

TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

When any operation on a file is attempted, APLX looks through the access matrix to find the first match for the user number (in the first column) and the pass number supplied when the file was tied (in the third column). A user number of 0 in the access matrix matches any user ID. If a match is found, the user is granted the permissions specified by the second element of the row. If no match is found, and the user is not the owner of the file, no permissions are granted. If no match is found, and the user is the owner of the file, all permissions are granted.

The access privileges can be specified in two ways. A positive privilege states what the user can do, and a negative privilege states what the user cannot do.

The privilege code is a number generated by adding various powers of 2 (1, 2, 4, 8, 16,....), each power of 2 corresponding to a particular privilege. Positive privilege codes are merely the sum of the individual privileges granted, whilst negative privilege codes are generated by adding `¯1` and the result of negating the sum of all the privileges denied.

Power of 2	Value	Operation
0	1	⎕FREAD
1	2	⎕FTIE (exclusive)
2	4	⎕FERASE
3	8	⎕FAPPEND
4	16	⎕FREPLACE ⎕FWRITE
5	32	⎕FDROP ⎕FDELETE
7	128	⎕FRENAME
9	512	⎕FRDCI ⎕FCSIZE
10	1024	⎕FRESIZE
11	2048	⎕FHOLD
12	4096	⎕FRDAC
13	8192	⎕FSTAC
14	16384	⎕FDUP

Permission to use `⎕FSTIE` (shared tie) is implicitly granted to any user who has any permission to use the file. `⎕FCREATE ⎕FLIB ⎕FNAMES ⎕FNUMS` and `⎕FUNTIE` do not need explicit APLX permissions (although the operating-system may restrict your rights). `⎕FSIZE` does not require explicit permission, but you must supply a pass number if the file was tied with one.

Examples of privilege codes are:

Privilege	Meaning
0	No access
1	⎕FSTIE ⎕FREAD
3	⎕FSTIE ⎕FTIE ⎕FREAD
17	⎕FSTIE ⎕FREAD ⎕FREPLACE ⎕FWRITE
¯33	Full access except for ⎕FDROP ⎕FDELETE
¯1	Full access

`⎕FSTIE` Open (tie) an existing file for shared use

`⎕FSTIE` opens (ties) an existing file for shared use.

See the description of `⎕FTIE` for full details.

⎕FTIE Open (tie) an existing file for exclusive use

The ⎕FTIE and ⎕FSTIE functions open ('tie') an existing component file. The syntax is:

```
        FILENAME ⎕FTIE TIENO {PASS}
```

or:

```
        FILENAME ⎕FSTIE TIENO {PASS}
```

If you tie the file using ⎕FTIE, it is tied for exclusive use and no other users or tasks will be able to tie it until you untie it. If you tie it using ⎕FSTIE, other tasks and/or users can also tie it using ⎕FSTIE.

`FILENAME` is a character vector specifying the name of the file to tie, following the same rules as ⎕FCREATE. The name may be specified in either of two ways. If the name contains a directory-separator character (: / or \), it is treated as a full host path name, and you need to specify the file extension, by convention `.aqf`. Otherwise it is treated as basic file name only, optionally preceded by a volume number 0 to 9, separated by one or more spaces from the name. In the latter case, the file is searched for in the directory 0 to 9 specified (the actual path is set the preferences dialog or ⎕MOUNT), and the file extension `.aqf` is added automatically.

`TIENO` is an arbitrary non-zero integer to be used in subsequent read/write operations to identify the file (the tie is exclusive). The tie number must not currently be in use to tie another file. Alternatively, you can provide a tie number of 0, in which case APLX automatically allocates the next available unused tie number, and returns it as the explicit result of the function.

The optional `PASS` parameter is a pass number. If you use this parameter, it must match one of the valid pass numbers for your user ID set in the access matrix (see the discussion of ⎕FSTAC below), and you must use the same pass number in all subsequent operations until you untie the file.

For example:

Exclusive-tie the file `RUN3.aqf` in library 0, under tie number 2:

```
      'RUN3' ⎕FTIE 2
```

Share-tie the file `RUN4.aqf` in library 1, allowing APLX to allocate the tie number:

```
      'RUN3' ⎕FSTIE 0
3
```

Share-tie the file `c:\temp\RUN3.aqf`, allowing APLX to allocate the tie number and using pass number 33421:

```
      'c:\temp\RUN3.aqf' ⎕FSTIE 0 33421
4
```

If you try to tie a file which is already tied for exclusive use by another user or task, APLX will retry for a few seconds in the hope that the tie will be released. If this does not occur, the operation will fail with error code 24 in `⎕LER`, or `⎕ET`:

```
      'RUN3' ⎕FTIE 2
FILE OR COMPONENT HELD
      'RUN3' ⎕FTIE 2
      ^
      ⎕LER
24 0
      ⎕ET
6 5
```

Files are untied automatically when an APLX task ends. They are NOT untied when you `)CLEAR` the workspace or `)LOAD` another workspace.

Special considerations for Client-Server implementations of APLX

See `⎕FCREATE` for details on how component files can be located on either the Client or Server machine.

Mixing 32-bit and 64-bit Component Files

If you are running both 32-bit and 64-bit versions of APLX, then it is possible to share component files between the two architectures, but there are some special points you should be aware of. See the introduction to the `⎕Fxxx Component File System` for details.

`⎕FUNTIE` Untie component file(s)

The function `⎕FUNTIE` unties zero, one or more currently-tied files. The syntax is:

```
      ⎕FUNTIE TIENOS
```

`TIENOS` is an integer vector of any length, containing the tie numbers of the files to be untied. If the tie number is not in use, no error is generated. You never need to supply a pass number to untie a file.

To untie all the files you currently have tied, use the expression:

```
      ⎕FUNTIE ⎕FNUMS
```

Files are automatically untied when the APL task ends. They are not automatically untied on `)CLEAR` or `)LOAD`.

⎕FWRITE Append, replace or insert component

The ⎕FWRITE function allows you to write a component anywhere in the file, either inserting, replacing or appending. The syntax is:

```
DATA ⎕FWRITE TIENO {COMPONENT} {PASS}
```

DATA is any APL array, or an overlay created using ⎕OV. TIENO is the tie number you used to tie or create the file (or the tie number returned by APLX if you tied or created it using 0 instead of your own tie number). If you tied the file using a pass number, you must provide the same pass number, as the PASS parameter.

COMPONENT is the component number at which the data should be written. This can be one of the following:

- If COMPONENT is zero, or is omitted, or is equal to the highest current component number plus 1, the data is written to the next available component number in the file (i.e. the highest current component number plus 1). If the file currently contains no components, the data will be written as component 1. This behavior is the same as that of ⎕FAPPEND.

  For example, if the file currently contains components 1 to 6, the following are all equivalent; the data will be written as a new component 7:

  ```
  DATA ⎕FWRITE TIENO
  DATA ⎕FWRITE TIENO,0
  DATA ⎕FWRITE TIENO,7
  ```
- If COMPONENT is an integer which corresponds to an existing component, the existing component is replaced by the new data supplied. For example, if the file currently contains components 1 to 6, and you supply a COMPONENT parameter of 3, the third component will be replaced by the new data you write. The number of components is unchanged. If the new array is bigger than the previous data, the necessary space will be allocated automatically. If the new array is smaller, the released space will be kept in a pool of available space and re-used if possible in a future operation. This behavior is the same as that of ⎕FREPLACE.
- If COMPONENT is a non-integral number between the first existing component minus 1, and the last component number plus 1, a new component is inserted and the data is written to the new component. The new component will be placed at the component number ⌈COMPONENT. Any later components will be renumbered to allow for the inserted component.

  For example, if the file currently contains components 1 to 6, the following statement would insert a new component between the existing fifth and sixth components, with the old sixth component becoming the new component 7:

  ```
  DATA ⎕FWRITE TIENO,5.5
  ```

If the file currently contains components 3 to 8, then the following statement would insert a new component (as the new component 3) before the current first component. The existing components would be renumbered 4 to 9:

```
            DATA ⎕FWRITE TIENO,2.5
```

If the number is between the current highest component number and the current highest component number plus 1, the operation is equivalent to appending a new component.

- Any other component number gives rise to an error `COMPONENT NOT IN FILE` (error code 20 in `⎕LER`, or 6 3 in `⎕ET`)

The following complete sequence illustrates the various possibilities:

Create a new file, and append four components to it in different ways:

```
      'EXAMPLE' ⎕FCREATE
      'First component' ⎕FWRITE 2 1
      'Second component' ⎕FWRITE 2
      'Third component' ⎕FWRITE 2 0
      'Fourth component' ⎕FWRITE 2 3.5
```

Read back the four components:

```
      ⎕DISPLAY ⎕FREAD ¨2,¨⍳4
┌→─────────────────────────────────────────────────────────────────────────────┐
│ ┌→──────────────┐ ┌→───────────────┐ ┌→──────────────┐ ┌→───────────────┐ │
│ │First component│ │Second component│ │Third component│ │Fourth component│ │
│ └───────────────┘ └────────────────┘ └───────────────┘ └────────────────┘ │
└∊─────────────────────────────────────────────────────────────────────────────┘
```

Drop the first component, components are now number 2 to 4:

```
      ⎕FDROP 2 1
      ⎕FSIZE 2
2 5 3584 0 256
      ⎕FREAD 2 1
COMPONENT NOT IN FILE
      ⎕FREAD 2 1
      ^
```

Replace component 3 with a new value:

```
      'Third rewritten' ⎕FWRITE 2 3
      ⎕FREAD 2 3
Third rewritten
```

Insert a new component before the existing first component, renumbering existing components to allow for it:

```
      'Inserted first' ⎕FWRITE 2 1.5
      ⎕FREAD 2 2
Inserted first
```

Insert a new component between the existing third and fourth components (it will become the new component 4):

```
      'Interloper' ⎕FWRITE 2 3.5
```

See what we've ended up with:

```
      ⎕FSIZE 2
2 7 4096 0 0
```

(First component number is 2, next available is 7)

```
      ⎕FREAD 2 2
Inserted first
      ⎕FREAD 2 3
Second component
      ⎕FREAD 2 4
Interloper
      ⎕FREAD 2 5
Third rewritten
      ⎕FREAD 2 6
Fourth component
```

We're done. Untie the file:

```
      ⎕FUNTIE 2
```

⎕FX Fix function/operator/class

This system function is the inverse of ⎕CR.

It accepts as a right argument the canonical representation of a function, operator, or class (i.e. the text representation of the item). The right argument may be a character matrix, a simple character vector with embedded carriage returns, or a nested character vector with each line a separate item in the vector.

It returns a result which is either a character vector containing the name of the function, operator or class which has been fixed, or an integer showing the first line which was invalid. An item cannot be fixed if the name of that item is already in use for another object type.

For functions and operators, a left argument of 0 (or no left argument) causes the function to be left unlocked, while 1 causes it to be locked. Normal applications can include functions which are modified under program control, and it is in such applications that ⎕CR and ⎕FX are of use.

For details of the format for the canonical representation of a class, see ⎕CR. Note that the order in which the members appear in the text form does not matter.

⎕HC Hard Copy

The monadic system function ⎕HC is used to write hard copy to an output device, usually a printer. The exact implementation of ⎕HC depends on the system on which APLX is implemented.

Available Printers

For **Server editions** of APLX, up to 9 logical printers may be available. These are defined using the Configure Printer option of the File.. Print.. menu, or may be pre-defined in a resource file. Each logical printer defines the spool queue and type of printer to be used.

Once printers have been set up, they can be used through the Print menu, or through ⎕HC. In ⎕HC printing, all output to the screen is also sent to the printer, unless screen printing has been switched off using ⎕HC 1000.

In **APLX for Windows** and **APLX for MacOS**, the operation of ⎕HC is similar except that printing always takes place to the currently-selected printer, i.e. the printer last selected using the *Print Setup* dialog. (You can display this dialog under APL program control using the 'Printer' object class in ⎕WI, which also provides alternative and more flexible printing facilities). Thus, only printer 1 can be selected using ⎕HC, even though there may be more printers potentially available via the dialog.

Syntax of ⎕HC

The syntax of ⎕HC is:

```
     RESULT ← ⎕HC CODE
```

where CODE defines the operation you want to carry out, and RESULT is usually an error code (see below). 0 always means successful operation.

⎕HC Function Codes

```
1 to 9       Switch on printing to printer number 1 to 9 inclusive
             (only printer 1 valid under MacOS and Windows)

¯1 to ¯9     Keep the specified spool file open, but do not write to it.
             This option is useful if you want to ask for input, or display
             progress messages which you do not want to print.

0            Switch off printing, submit the spool file to the
             printer.

1000         Suppress screen output, but continue to write to the spool file.
             This option is useful if you want APL output to be printed, but not
             appear on the screen. This option is valid only if printing is
             on. Screen output is reset automatically by ⎕HC 0.
```

¯1000	End suppression of screen output.
99	Return the current value of ⎕HC. In this case, the result is not an error code, but a printer number 1 to 9 (or ¯1 to ¯9), or 0 if printing is not on.

The following codes are valid for Server editions of APLX only:

100,200...	Open spool file (append if it exists), but don't automatically submit on Quad-HC 0. 100 is Printer 1, 200 is Printer 2, etc.
101,201...	Submit spool file opened with 100, 200 etc, then delete.
102,202...	Submit but don't delete.
103,203...	Delete spool file opened with 100, 200...

⎕HC Return Codes

0	Operation successful
1	Invalid command or printer not defined
2	Printer not accessible
3	I/O error or disc full
4	⎕HC 1000 requested, but printing not on

⎕GETCLASS Get reference to named class

The system function ⎕GETCLASS returns a reference to a named Internal or External class. It is not implemented for System classes.

The left argument is a character vector which specifies the environment in which the class exists, in the same format as for ⎕NEW. It can be omitted, in which case it is assumed that the class is an internal (user-defined) class in the workspace. The right argument is a character vector containing the name of the class. If the class is found, a scalar reference to the class is returned:

```
      ⎕GETCLASS 'Point'
{Point}
      (⎕GETCLASS 'Point').⎕NL 2
X
Y
Z
```

In this Java example, we fetch a reference to the class `TimeZone`, which can then be used to call a static method.

```
      tzclass←'java' ⎕GETCLASS 'java.util.TimeZone'
      tzclass.⎕CLASSNAME
java:java.util.TimeZone
```

```
      tz←tzclass.getTimeZone 'America/Los_Angeles'
```

Note: If you are creating many thousands of instances of an external class, it may be much more efficient to use `⎕GETCLASS` to fetch a reference to the class once, and pass that reference to `⎕NEW`, rather than passing the class name to `⎕NEW`.

If you)SAVE a workspace containing references to external classes, the references will be set to `⎕NULL` when the workspace is reloaded.

`⎕HOST` Command to Host

The `⎕HOST` system function allows you to issue a command directly to the host operating-system environment and display or capture the result, without leaving the APL workspace. This system function is highly implementation-specific and some operating system commands may not be allowed.

`⎕HOST` takes as its right argument the command to be executed. It returns the output (if any) from the command as a character vector, possibly with embedded newline characters. You can also optionally specify a left argument which is the maximum time (in seconds) to wait before returning control to APL. This is useful to prevent your APL program being blocked if, for example, the program called encounters an error which causes it to wait for input from the command-line. If the optional left argument is omitted, a default timeout of 20 seconds is used.

If the right argument to `⎕HOST` is an empty vector, it returns a character vector with the operating system name: '`LINUX`' '`AIX`' '`WINDOWS`' or '`MACOS`'

For example, under Linux:

```
            ⎕HOST ''
      LINUX
            SERVER←⎕HOST 'hostname'
            SERVER
      penguin
           10 ⎕HOST 'telnet rs6000'
```

Note that, under Windows, many common commands are 'built-in' to the command-line shell, rather than being separate executable programs. Under Windows NT, 2000, XP and Vista, you can run these using the '`CMD`' program with the '`/C`' option. (Under Windows 95, 98 and ME, use '`COMMAND.COM /C`'). For example:

```
            ⎕HOST 'CMD /C dir'
     Volume in drive C has no label.
     Volume Serial Number is 07D0-0B11

         Directory of C:\aplx\ws
     20/06/2001  19:13       [DIR]          .
     20/06/2001  19:13       [DIR]          ..
     05/09/2001  16:43               17,792 JIM.aws
     05/09/2001  17:06                  574 EXPLORE.atf
     30/07/2001  19:49               17,828 QNA.aws
                3 File(s)        39,313 bytes
                2 Dir(s)  14,797,176,832 bytes free
```

Under MacOS 8 and 9, because there is no command-line interface at the operating system, `⎕HOST` is not implemented except to report the OS name. Under MacOS X, `⎕HOST` runs the command under the BSD Unix-style shell.

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine. The two parts of the application communicate via a TCP/IP network. Typically, the Client will be the APLX front-end built as a 32-bit Windows application running on a desktop PC, and the Server will be a 64-bit APLX64 interpreter running on a 64-bit Linux or Windows server.

In such systems, `⎕HOST` allows you to specify whether the command should be executed on the Client or the Server machine. You do this by preceding the `⎕HOST` right argument with either an Up Arrow `↑` to indicate that the command should be executed on the Client, or a Down Arrow `↓` to indicate that it should run on the Server. If you do not specify, the default is that the call should take place on the Client.

In this example, the Client is running under Windows, and the Server under Linux x86_64:

```
      ⎕HOST '↑cmd /c vol c:'          ⍝ Execute on Windows Client machine
 Volume in drive C has no label.
 Volume Serial Number is 07D0-0B11

      ⎕HOST '↓uname -nsp'             ⍝ Execute on Linux Server machine
Linux Server23 x86_64
```

⎕I Idle Character

The niladic system function `⎕I` returns the Idle or time-fill character

```
              ⎕AV⍳⎕I
       1
```

⎕IC Insert into Class

The dyadic system function `⎕IC` allows you to insert existing functions and/or variables into a class. It is particularly useful for converting older APL systems into object-oriented form.

Example:

```
      'Point' ⎕IC ⎕BOX 'X Y MOVE'
```

The left argument is a character vector containing the name of a new or existing class. If you specify a new class, it will be created. If you want the new class to inherit from an existing class, you can follow the class name by a colon and the name of the parent class.

The right argument is a matrix of the names of the global functions, operators and variables which you want to insert into the class, padded to the right with blanks if necessary. If there is only one name, the right argument can be a character vector. If the right argument is an empty vector, the class will be created but no members will be inserted into it. The named items will be transferred into the class as follows:

- If the name corresponds to an existing function or operator, it will become a public method of the class.
- If the name is the name of a function and is the same as the name of the class being created (i.e. the left argument), it will become the Constructor for the class.
- If the name corresponds to an existing variable, it will become a public, read-write instance property of the class, with the default value of the property being the current value of the variable. However, the variable must not contain a class or object reference.
- If the name is currently undefined, it will become an uninitialized, public, read-write instance property of the class.

Of course, you can alter the attributes of the members of the class later on by using the class editor, for example if you want to make one of the properties read-only.

Note that the existing global definition of the inserted functions, operators and variables will be deleted, i.e. the operation *moves* rather than *copies* the items into the class. If the name corresponds to an existing member of the class, the existing member will be overwritten.

The explicit result of `⎕IC` is a boolean vector with one element per name, with a value of 1 if the corresponding member was successfully inserted into the class, and 0 if it was not. (It will fail if the name is invalid, or is the name of something other than an operator, function or variable, or if a variable contains class or object references).

In this example, we start with a workspace with a few variables and functions:

```
      Subject←''
      Sender←'Charles Barker'
      Text←''
      ∇Message B
[1]   Text←B
[2]   ∇
      ∇R←Length
[1]   R←ρText
[2]   ∇
      )FNS
Length  Message
      )VARS
Sender  Subject Text
```

Now we insert all the existing functions and variables into a new class called `Message`. Note that the existing function `Message` becomes the Constructor of the new class:

```
      'Message' ⎕IC ⎕NL 2 3
1 1 1 1 1
      )FNS
      )VARS
      )CLASSES
Message
      ⎕CR 'Message'
Message {
Text←''
Subject←''
Sender←'Charles Barker'

∇Message B
Text←B
∇

∇R←Length
R←ρText
∇
}
      Message.⎕NL 2    ⍝ Public properties
Sender
Subject
Text
      Message.⎕NL 3    ⍝ Public methods
Length
      Message.⎕NL 10   ⍝ Constructor
Message
```

Finally, we create another class `EMail` which inherits from `Message`, and adds an extra uninitialized property `Recipient`:

```
      'EMail:Message' ⎕IC 'Recipient'
1
      ⎕CLASS EMail
{EMail} {Message}
      )CLASSES
EMail   Message
```

⎕ID ID Number

The niladic system function ⎕ID returns a single integer which is the System ID Number.

To provide security for applications and packages written to run under APLX, ⎕ID, can be used to query the system identification number. Each APLX interpreter has a unique identity code. This feature (used, of course, within a locked function) could help prevent unauthorised use of software by detecting the use of a function on an unauthorised system and taking appropriate action.

⎕IMPORT Import data from file in specified format

The monadic system function ⎕IMPORT imports data from an external file into an APL array. It supports a number of data formats which are commonly used for data exchange by spreadsheets and other (non-APL) applications. (See also ⎕EXPORT which allows you to export data to a file of specified format, which can then be read into another application.)

The right argument determines the name of the file to be read, and the format of the file. If the right argument is a character vector, it is interpreted as the name of the file you want to import (including full path if required) and the format of the file is inferred from the file extension. If the right argument is a two element nested vector, the first element is the filename (or full pathname), and the second is a text string specifying the file type. File types are case-insensitive.

The explicit result is the converted data.

For example, the two following statements are equivalent, and will import the contents of the file Budget2007.csv in 'comma-separated variables' ('CSV') format, into a variable called BUDGET in the workspace:

```
      BUDGET ← ⎕IMPORT 'Budget2007.csv'
      BUDGET ← ⎕IMPORT 'Budget2007.csv' 'csv'
```

The following file formats are supported by ⎕IMPORT, with the behavior shown:

File type/extension	***Description***	***Behavior***
`'txt'`	Text file, with data represented in 8-bit extended ASCII form.	The contents of the file are read as text, and converted to APL text form. The result is a character vector, with APL newline characters (⎕R) between each line.
`'utf16'` or `'utf-16'`	Same as 'txt', but with characters represented in 16-bit UTF-16 Unicode form (2 bytes per character).	Same as 'txt'. Any Unicode characters which cannot be represented in APLX will be converted to the character set by ⎕MC (by default, question mark).

`'utf8'` or `'utf-8'`	Same as 'txt', but with characters represented in the 8-bit UTF-8 Unicode form (variable number of bytes per character).	Same as 'txt'. Any Unicode characters which cannot be represented in APLX will be converted to `⎕MC` (by default, question mark).
`'csv'`	'Comma-separated variables' format, as used by many applications such as spreadsheets for data exchange. The file comprises one line of text per row of the data, with individual elements separated by commas. Numeric elements are expressed in text form. Text elements are usually surrounded by double-quotation marks.	The result is always a matrix. Elements which are either enclosed in quotes, or are not valid numbers, are converted to text vectors. Elements which are valid numbers and not enclosed in quotes are converted to APL numeric form. The result is either a nested or a numeric matrix, depending on whether any of the elements were text vectors.
`'tsv'`	'Tab-separated variables' format. Same as CSV, except the fields are separated by tab characters instead of commas.	Same as for CSV.
`'slk'`	Symbolic Link (SYLK) format, a Microsoft-specified file format typically used to exchange data between applications such as Excel and other spreadsheets.	The result is always a matrix. Elements which are either enclosed in quotes, or are not valid numbers, are converted to text vectors. Elements which are valid numbers and not enclosed in quotes are converted to APL numeric form. The result is either a nested or a numeric matrix, depending on whether any of the elements were text vectors. Some non-ASCII characters may not be representable in APL; these will be replaced by `⎕MC` (by default, question mark).
`'xml'`	Extensible Markup Language (XML) format, a format used for saving structured data with markup information.	The file is read and converted to an APL array with the same specification as `⎕XML`. This conversion is equivalent to the two-stage command: `      ⎕XML ⎕IMPORT 'filename' 'utf8'` The file may be encoded in UTF-8 or UTF-16 format; APLX determines the file encoding automatically. Some non-ASCII characters may not be representable in APL; these will be replaced by `⎕MC` (by default, question mark).

For example, suppose that the file 'Budget2007.csv' contains the following lines in CSV format:

```
"","Q1","Q2","Q3","Q4"
"Sales",11300,13220,16550,19230
"Expenses",12450,12950,13640,13980
"Profit",-1150,270,2910,5250
```

This can be read into a 4-row matrix of text vectors and numbers as follows:

```
      BUDGET ← ⎕IMPORT 'Budget2007.csv'
      BUDGET
             Q1    Q2    Q3    Q4
 Sales    11300 13220 16550 19230
 Expenses 12450 12950 13640 13980
 Profit   ¯1150   270  2910  5250
      ⎕DISPLAY BUDGET
┌→───────────────────────────────────┐
↓ ┌⊖┐        ┌→─┐  ┌→─┐  ┌→─┐  ┌→─┐  │
│ │ │        │Q1│  │Q2│  │Q3│  │Q4│  │
│ └─┘        └──┘  └──┘  └──┘  └──┘  │
│                                    │
│ ┌→────┐                            │
│ │Sales│    11300 13220 16550 19230 │
│ └─────┘                            │
│                                    │
│ ┌→───────┐                         │
│ │Expenses│ 12450 12950 13640 13980 │
│ └────────┘                         │
│                                    │
│ ┌→─────┐                           │
│ │Profit│   ¯1150 270   2910  5250  │
│ └──────┘                           │
└∊───────────────────────────────────┘
```

Errors

If the file import fails because the format does not match what APLX is expecting, the error `DATA DAMAGED` will reported. In addition, a longer text message which gives more information will be displayed to the Session window (this is suppressed if error trapping is enabled). This example fails because the data is not in SYLK format:

```
      BUDGET ← ⎕IMPORT 'Budget2007.csv' 'slk'
Could not determine bounds of array
DATA DAMAGED
      BUDGET←⎕IMPORT 'Budget2007.csv' 'slk'
             ^
```

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine. Typically, the Client will be the APLX front-end built as a 32-bit Windows application running on a desktop PC, and the Server will be a 64-bit APLX64 interpreter running on a 64-bit Linux or Windows server.

In such systems, you can specify whether the file should be accessed on the Client or the Server machine. You do this by preceding the file name with either an Up Arrow ↑ to indicate that the file should be accessed on the Client, or a Down Arrow ↓ to indicate that it should be accessed on the Server. If you do not specify, the default is that the access takes place on the Client.

⎕INSTANCES Instances of a Class or Descendants

The monadic or dyadic system function ⎕INSTANCES returns a vector of references to all the instances of a given internal class. The right argument is either a class reference, or a character vector containing the name of an internal class.

If the left argument is omitted, or is 0, the list will include instances of the specified class and of any descendants of the class. If the left argument is 1, the list will be restricted to instances of the specified class itself only.

In this example, classes `Rectangle` and `Triangle` both inherit from class `Polygon`:

```
      )CLASSES
Polygon Rectangle       Triangle
      T1←⎕NEW 'Triangle'  ⋄ T1.⎕DF 'Triangle1'

      T2←⎕NEW 'Triangle'  ⋄ T2.⎕DF 'Triangle2'

      P1←⎕NEW 'Polygon'  ⋄ P1.⎕DF 'Polygon1'

      R1←⎕NEW 'Rectangle'  ⋄ R1.⎕DF 'Rectangle1'

      ⎕INSTANCES 'Polygon'    ⍝ Form using class name as right arg
Triangle1 Triangle2 Polygon1 Rectangle1
      0 ⎕INSTANCES Polygon    ⍝ Form using class reference as right arg
Triangle1 Triangle2 Polygon1 Rectangle1
      1 ⎕INSTANCES Polygon
Polygon1
```

⎕INSTANCES cannot be used to find instances of external classes.

⎕IO Index Origin

The system variable ⎕IO has a numeric value of 0 or 1 which states whether the origin for counting is 0 or 1. This determines whether the first element in a count is element 0 or element 1. The default is 1. A new value can be assigned. ⎕IO may be localised in a function.

```
             ⎕IO←1
             ⍳4
       1 2 3 4
             ⎕IO←0
             ⍳4
       0 1 2 3
```

Operations which involve indexing of arrays (`[ ]`) are affected by ⎕IO.

A number of functions are affected by `⎕IO`:

```
              ⍳ ? ⍋ ⍒ ⍉ ⊃ ⌷
```

Functions and operators which use axis specifications are also affected:

```
              ⌽  ⊖  /  ⌿  \  ⍀  , ↑ ↓ ⊂ ⊃
```

and finally, the line number returned by `⎕FX` if an error occurs is affected by `⎕IO`.

⎕L Linefeed Character

The niladic system function `⎕L` returns the Linefeed character. On output, a linefeed in APLX moves the cursor to the next line without changing the column position:

```
      ⎕A,⎕L,⎕D
ABCDEFGHIJKLMNOPQRSTUVWXYZ
                          0123456789
      ⎕AF ⎕L
10
```

`⎕L` is the same as `⎕TCLF` or `⎕TC[⎕IO+2]`. See `⎕TC` and `⎕TCxx`.

⎕LC Line Counter

The niladic system function `⎕LC` returns a numeric vector of all current line numbers of functions in the State Indicator. The first number is that of the function at the top of the state indicator `(⎕SI)` stack.

⎕LE Last Exception

Not implemented for APL internal classes or system classes

Syntax:

```
    error_message ← 'env' ⎕LE 0
    objectref ← 'env' ⎕LE 1
```

The system function `⎕LE` can be used to get information about the most recent exception that was caught by APLX during a call to an external environment such as .Net or Java.

The left argument is a character vector which specifies the external environment for which you want exception information, in the same format as for `⎕NEW`. The right argument is an integer which specifies which information you require:

Code	*Information returned*
0	Return the last error message associated with an exception in the specified environment
1	Return a reference to the exception object

Once obtained, the object reference to the exception can be retained: it is unaffected by any subsequent exceptions. If no exception has ever occurred in the specified environment the error message is an empty character vector and a NULL object reference is returned.

Example using .NET

```
      ⍝ Do something which causes an exception
      date←'.net' ⎕NEW 'System.DateTime' 2008 02 16
      date.Year
2008
      date.Year←2009
Property Year is read-only
DOMAIN ERROR
      date.Year←2009
      ^

      ⍝ Use ⎕LE to investigate
      '.net' ⎕LE 0
Property Year is read-only
      exception←'.net' ⎕LE 1
      exception.⎕CLASSREF
{.net:Exception}
      exception.Message
Property Year is read-only
```

Example using Java

```
      ⍝ Do something which causes an exception
      a←'java' ⎕NEW 'java.math.BigInteger' '123'
      a.testBit ¯3
Cannot call method
JVM: Exception in thread "main"
DOMAIN ERROR
      a.testBit ¯3
      ^

      ⍝ Use ⎕LE to investigate
      x←'java' ⎕LE 1
      x.⎕ds
java.lang.ArithmeticException: Negative bit address
      x.getStackTrace
[java:StackTraceElement]
      x.getStackTrace.⎕DS
[Ljava.lang.StackTraceElement;@ca0b6
      (x.getStackTrace).toString
 java.math.BigInteger.testBit(Unknown Source)
```

For the R interface, the right argument of 1 (exception object) is not supported, but the latest error message is.

⎕LER Line Error Report

The niladic system function ⎕LER returns a two-element numeric vector comprising the error code and line number of the last error that occurred. See the earlier chapter on Error Handling for a fuller discussion of ⎕LER and the pre-assigned Error Codes which it returns.

See also ⎕ERS (Error signalling) and ⎕ET (APL2-style error codes).

⎕LIB Return names of files in directory

⎕LIB returns a character matrix of the names of all the files in a particular directory. Names are padded to the right with blanks as necessary. It takes a single argument, which can be either a library number (usually 0 to 9, corresponding to the rows of the ⎕MOUNT table), or a character string representing an operating-system path.

For example, under Windows you might have:

```
      ⎕LIB 1
run2.aws
run3.aws
README.txt

      ⎕LIB 'C:\'
temp
My Music
DELL
I386
AdobeWeb.log
PDOXUSRS.NET
PAGEFILE.SYS
NTDETECT.COM
DRIVERS
Documents and Settings
Firefox
temp.gif
temp.bmp
```

The file names are returned in full, including any file extension.

See also ⎕FLIB which returns the names of component files only in a directory.

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine.

In such systems, you can specify whether the directory being searched is on the Client or the Server machine. You do this by preceding the path name with either an Up Arrow ↑ to indicate that the directory is on the Client, or a Down Arrow ↓ to indicate that it is on the Server. (If you do not specify, the default is that the access takes place on the Client.) This is true either if you specify the full path name in the `⎕LIB` call, or via the `⎕MOUNT` table.

⎕LX Latent Expression

The system variable `⎕LX` contains a text expression to be executed when the workspace is loaded. When the workspace is loaded, the execute function is used to execute `⎕LX`, and so `⎕LX` can be used to invoke a function, print some text, or execute a multi-statement line.

```
            ⎕LX←'START'                        Runs START
            ⎕LX←''' TEXT STRING'''             Prints TEXT STRING
            ⎕LX←''' TEXT STRING'' ⋄ START'     Does both
```

⎕M Months

The niladic system function `⎕M` returns a 12-row 9-column character matrix of the months.

```
            ⎕M
      JANUARY
      FEBRUARY
      MARCH
      APRIL
      MAY
      JUNE
      JULY
      AUGUST
      SEPTEMBER
      OCTOBER
      NOVEMBER
      DECEMBER
```

⎕MC Missing Character

The System Variable ⎕MC contains the character used to replace a Unicode or other character which cannot be represented in APLX. By default, it is set to a question mark, but you can set it to any character in ⎕AV. It can be localized in the header of a function or method.

See ⎕UCS for an example, and for more information on translating to and from Unicode.

Note that, as well as translation using ⎕UCS, the character specified in ⎕MC is also used when translating Unicode text received anywhere within APLX. This can include Unicode text which has been:

- pasted from the clipboard
- returned by an external object using the .Net, Java or other interface
- read from a native file using ⎕NREAD
- imported from a file using ⎕IMPORT
- read from a database using ⎕SQL
- returned from an external shared-library call using ⎕NA

⎕MOUNT Allocate Libraries

Under APLX, you can access, at one time, up to ten user-specified libraries (0-9, regardless of the ⎕IO setting). *(Note: Library 10 is always mapped to the library containing utility and demonstration workspaces supplied with APLX.)* You can also specify full pathnames in system commands such as)LOAD, in order to access directories which are not allocated to library numbers.

⎕MOUNT allows you to read or alter the mapping between APLX Library Numbers and directories on the system. Library numbers are used by System Commands such as)LOAD and)SAVE, as well as by the APLX file systems.

Interrogation of ⎕MOUNT:

Invoking ⎕MOUNT with an empty vector character right argument will return the current mapping. This will be a matrix with 10 rows and up to 260 columns, corresponding to library numbers 0 to 9. For example, under MacOS you might have:

```
      ⎕MOUNT ''
MacHD:apl:ws:                       Logical Unit 0
MacHD:users:jim:                    Logical Unit 1
                                    Logical Unit 2 (Blank)
```

... etc

Alteration of the table:

You can change the mapping of library numbers to directories by using `⎕MOUNT` with a vector or matrix text right argument.

```
           ⎕MOUNT VARIABLE
```

will place `VARIABLE` in `⎕MOUNT`. If `VARIABLE` is a 10 row matrix, all of `⎕MOUNT` changes. Otherwise as much of `⎕MOUNT` as is affected, changes. For example, if `VARIABLE` has 4 rows, it will alter the top four rows (libraries 0 to 3) of `⎕MOUNT`. A vector right argument alters the top row, as does a scalar right argument.

The result returned is the new setting of `⎕MOUNT`.

Note that you should include the trailing directory separator in directory specifications. This is `/` for AIX and Linux systems, \ for Windows systems, and : for MacOS.

A blank row means the current default directory (ie the directory you were in when you started APLX).

Note that you can specify the `⎕MOUNT` table to be used when APL starts up in the future. To do this, choose the Preferences item in the Tools menu.

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine.

In such systems, you can specify whether the directory to be associated with a line of `⎕MOUNT` is on the Client or the Server machine. You do this by preceding the path name with either an Up Arrow `↑` to indicate that the directory is on the Client, or a Down Arrow `↓` to indicate that it is on the Server. (If you do not specify, the default is that the access takes place on the Client.)

⎕N Null Character

The niladic system function ⎕N returns a non-printing fill character.

```
      ⎕AV⍳⎕N
2
```

⎕NA Define External Function

The ⎕NA ("name association") system function allows you to associate a name in the APL workspace with an external (non-APL) function call. It thus defines an 'external function' which can be used like an ordinary APL function, but which calls out to the operating system or to a shared library (DLL).

Describing the external function

⎕NA takes as its right argument a function descriptor which specifies the library in which the external function resides, its name, and the calling conventions of the function. You can also supply an optional left argument which is the name to be used for the function in the APL workspace. If you omit the left argument, the APL name is the same as the external routine name.

A typical routine descriptor looks like this:

```
'I4 msupport.dll|AddNode I4 U I4 ={I[2] U I[4]}'
```

where the first field (`I4`) is the return type and the next (`msupport.dll`) is the library. The function name follows the vertical bar '`|`' character, and is `AddNode`. This is followed by the function parameters. Thus, the above descriptor contains the following information:

```
I4              Return type = 4-byte integer
msupport.dll    DLL name/path
AddNode         Function name
I4              1st parameter: Integer
U               2nd parameter: Unsigned integer
I4              3rd parameter Integer
={I[2] U I[4]}  4th parameter: Pointer to structure of
                2 Signed Ints, 1 Unsigned Int, 4 Signed Ints
```

The supported basic types are:

```
I or I4        4-byte signed integer
I8             8-byte signed integer (supported in APLX64 only)
I2             Signed short (2-byte integer)
I1             Signed byte (1-byte integer)
U or U4        4-byte unsigned integer
```

U8	8-byte unsigned integer (supported in APLX64 only)
U2	Unsigned short (2-byte integer)
U1	Unsigned byte (1-byte integer)
D or F8 or D8	8-byte double-precision floating point
F or F4 or D4	4-byte single-precision floating point
C or CT	Character (with translation)
CU	Character (without translation)
W	Wide (Unicode) character, translated to APLX character
P or PT	Pascal string character (with translation)
PU	Pascal string character (without translation)

The supported qualifiers are:

<	Pointer, in only. No result comes out, but incoming data goes in.
>	Pointer, out only. On return, the result is returned in the nested result
=	Pointer, in and out
{}	Encloses structure elements
[]	Encloses array elements

Arrays are typically fixed-size, and are represented using the `[]` notation. This corresponds to an APL vector of the same length. You can specify an array of indeterminate length using * as a wildcard character (eg `C[*]`), but only when passing a pointer to an array into the external function, not for returned values or inside structures.

Structures are represented using the `{}` notation, and the corresponding APL data is a nested array. Arrays and structures are typically passed by pointer, i.e. the external routine expects a pointer to the array or structure. This is indicated by one of the < = or > qualifiers, depending on whether the data is passed into the function (case '<'), returned by the function (case '>'), or both (case '='). If data is returned into a structure or array, it will be returned by APLX in a nested vector returned when the function is run.

The result is either a scalar (being the result in the first field), or a nested vector of the result followed by one element per > or = field in the descriptor. If there is no explicit result to the function (`void` in the C language), you can simply omit the result type.

Strings and Character Arrays

There are various ways in which you can specify strings and character arrays. Usually, you should use an array of `C` or `CT` type, to indicate a character string which will be translated between APL's internal representation (`⎕AV` position) and the normal non-APL equivalent (extended ASCII). It will also be null-terminated as is usual in the C language, by adding a null byte at the end of strings passed from APL to the external function, and setting the length of any string returned to APL to one byte less than the position of the first null character. In both cases, the string will be also be terminated if the maximum length you specify is reached. For example, the type `C[64]` indicates a C string of up to 64 bytes including any final null character. The type `CU` is similar, but no translation takes place.

Wide or Unicode characters (type `W`) are similar. These are 16-bit characters; if you use this type, text is automatically translated from APLX's internal representation to Unicode before making the call. Any returned Unicode text is translated back to APLX's internal representation after making the call. Any Unicode characters which are not in the APLX character set are converted to question marks. (If you want to retrieve the raw Unicode values, use type `U2` for unsigned short, which will retrieve them as integers).

The Pascal string types are valid only in array definitions, not as simple data elements. They indicate a Pascal-type string where the first byte is the length of the string, followed by the string. APLX will automatically add the length byte before calling the external function, and will remove it before returning any result to APL. For example, `P[63]` indicates a Pascal string of up to 63 characters (i.e. 64 bytes including the length byte). `PU[63]` is the same, except that no character translation takes place

Structure alignment

When the function descriptor defines a structure using the `{}` notation, APLX will automatically try to ensure that the elements in the structure are aligned correctly relative to the start of the structure. This generally means that 2-byte elements (`I2` and `U2`) are aligned on 2-byte boundaries, and 4-byte elements (`I`, `I4`, `U`, `U4` and pointers) are aligned on 4-byte boundaries. APLX will automatically allow for any padding bytes required to achieve this.

Occasionally you may need to modify this default behavior. You can do this by adding an alignment modifier enclosed in curly brackets, just after the library name. This takes the form `{a=N}` where `N` is one of the values 1, 2, or 4. The effect of this is to specify the maximum alignment of any element; if the size of an element is less than this maximum, it will be aligned on its natural boundary. This gives the following results:

- `{a=1}` No padding is inserted; elements will be packed together.
- `{a=2}` Two-, four- and eight-byte elements (and pointers) will all be aligned on 2-byte boundaries only.
- `{a=4}` Two-byte elements will be aligned on 2-byte boundaries. Four- and eight-byte elements (and pointers) will be aligned on 4-byte boundaries.

For example, under MacOS the Quicktime routines are in a library called 'QuickTimeLib'. This expects structures to be aligned according to Motorola 68000-style alignment rules, with four-byte elements aligned on 2-byte boundaries. The function `OpenMovieFile` takes as its first argument a pointer to an `FSSpec` structure comprising a 2-byte integer, a 4-byte integer and a 63-long Pascal string (see below for more details). The second argument is a pointer to a location where a 2-byte integer will be written, and the third is a 1-byte integer. It returns a two-byte integer error code. You can define this function and specify the correct structure alignment as follows:

```
      ⎕NA 'I2 QuickTimeLib{a=2}|OpenMovieFile <{I2 I4 P[63]} >I2 I1'
```

Under MacOS, a special case applies for calls to the operating system shared library 'CarbonLib', which also uses 68000-style alignment rules. This special case is detected automatically by APLX, so you do not need to specify the alignment explicitly.

Using external functions

Once an external function has been defined using ⎕NA, you can use it rather as you would an ordinary user-defined monadic (or possibly niladic) APL function. However, the right argument to the function must correspond to the data type which the external function expects. This means that where there is a parameter of type I or U, an APL integer value must be provided. (APLX will automatically convert your data to and from the 1-byte and 2-byte integer forms if appropriate). Where character data or a Pascal string is specified, you must provide an APL character scalar or vector. Arrays must be represented by APL vectors, and structures by APL nested arrays.

External functions are saved with the workspace, and they can be copied using)COPY or placed in overlays using ⎕OV. You do not need to run ⎕NA again to re-enable the name association, although there is no harm in doing so.

Shared libraries under Windows

Under Windows, shared libraries usually have the file extension .dll (Dynamic Link Library). Windows will search for the DLL in the following places:

If 'SafeDllSearchMode' is enabled (the default for Vista and XP SP2 or later):

1. The directory from which APLX was loaded.
2. The system directory.
3. The 16-bit system directory.
4. The Windows directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable.

For older versions of Windows:

1. The directory from which APLX was loaded.
2. The current directory.
3. The system directory.
4. The 16-bit system directory.
5. The Windows directory.
6. The directories that are listed in the PATH environment variable.

32-bit calls can be made to functions in a DLL which follow either the 'cdecl' or 'stdcall' calling convention. APLX automatically fixes up the stack as necessary.

Shared libraries under MacOS

The original shared library format under MacOS versions prior to MacOS X was accessed through the 'Code Fragment Manager'. In this format, shared libraries were files of type 'shlb' with an appropriate 'cfrg' resource. An example is 'CarbonLib', the library which implements most of the MacOS user-interface routines. APLX can call routines inside shared libraries directly using the `⎕NA` mechanism. For this case, you should omit the file path and just give the name of the shared library.

In MacOS X, Apple introduced an extended shared-library file type known as a 'bundle'. In addition, a special directory known as a 'framework' (containing one or more bundles) is also available. (The core operating system libraries are contained in frameworks.) You can call both of these using `⎕NA`. To access a 'bundle', the library name should be the full path to the bundle, for example `'WorkDisk:RocketScience:newton.bundle'`. To access a framework, you should omit the full path, and give just the framework name, for example `'System.framework'` or `'Carbon.framework'`.

Shared libraries under Linux

Under Linux, shared libraries usually have the file extension `.so`. Linux will search for the shared library in the following places:

1. The directories specified in the LD_LIBRARY_PATH environment variable, if any.
2. `/lib`
3. `/usr/lib`

Note that it is very common in Linux to have file links to shared libraries, sometimes with multiple versions of a library. For example, the current version of the library might be might be `/usr/lib/libisc.so.9.1.5`, specifying the exact version and build number, with a link `/usr/lib/libisc.so` pointing at it.

Example 1 (Windows)

The Windows `GetTickCount` function retrieves the number of milliseconds that have elapsed since Windows was started. The function prototype given in the Windows documentation is:

```
DWORD GetTickCount(VOID);
```

This means that the function takes no argument, and returns a DWORD (4-byte unsigned integer) result. The Windows documentation also states that this function is implemented in the library 'kernel32'. To declare this function in *APLX for Windows*, you would therefore write:

```
      ⎕NA 'U kernel32|GetTickCount'
```

This will create a niladic function in the APL workspace called `GetTickCount`. Each time you run it, it will return an integer giving the number of milliseconds that have passed since Windows started:

```
      GetTickCount
22340594
```

```
            GetTickCount
      22342907
```

If you wanted to give a different APL name to the same function, you could provide a left argument to `⎕NA`:

```
            'GETMS' ⎕NA 'U kernel32|GetTickCount'
            GETMS
      22343405
```

Example 2 (Windows)

The Windows `GetSystemDirectory` function retrieves the path of the Windows system directory. It exists in two versions, one using single-byte characters (`GetSystemDirectoryA`) and one for Unicode characters (`GetSystemDirectoryU`). The function prototype given in the Windows documentation is:

```
UINT GetSystemDirectoryA(LPTSTR lpBuffer, UINT uSize);
```

This means that the function takes two arguments, and returns a 4-byte unsigned integer result. The arguments are a pointer to a buffer where the result will be returned, and an integer giving the length of the buffer. The Windows documentation also states that this function is implemented in the library 'kernel32'. To declare this function in *APLX for Windows*, you would therefore write something like:

```
            'GetSystemDirectory' ⎕NA 'U kernel32|GetSystemDirectoryA >C[256] U'
```

This will create a monadic function in the APL workspace called `GetSystemDirectory`. It takes two arguments, one for the buffer (which will not be used, but an element should be supplied), and an integer giving the length of the buffer (up to 256). It will return a nested vector, the first element of which is the result of the function itself. The Windows documentation states that this result is the length required for the returned string, or 0 if an error occurs. The second element in the nested array returned will be the string itself, corresponding to the > pointer argument. To use the function, you need to supply a dummy argument for the first parameter (pointer to buffer where the returned string will be placed), and the length of the buffer:

```
            GetSystemDirectory '' 255
       17 C:\WINNT\System32
```

Example 3 (Linux)

Under Linux, the `getuid` function returns the user ID. The function prototype given in the Linux documentation is:

```
uid_t getuid(void);
```

where the return type `uid_t` is a 4-byte integer result. The function exists in the 'libc' library, which on a typical system is held in '/lib/libc.so.6' To declare this function in *APLX for Linux*, you would therefore write something like:

```
            ⎕NA 'U /lib/libc.so.6|getuid'
            getuid
      203
```

Example 4 (AIX)

Under AIX, the `getpid` function returns the process ID. The function prototype given in the AIX documentation is:

```
pid_t getpid(void);
```

where the return type `pid_t` is a 4-byte integer result. The function exists in the AIX kernel, which in AIX is referenced using the library name '/unix'. To declare this function in *APLX for AIX*, you would therefore write something like this (here we name the function `PROCID` in the APL workspace):

```
      'PROCID' ⎕NA 'U /unix|getpid'
      PROCID
15482
```

Example 5 (MacOS Shared Library)

Under MacOS, the `SetWTitle` function allows you to change a window's title. The function prototype given in the MacOS documentation is:

```
void SetWTitle (WindowRef window, ConstStr255Param title);
```

where the first parameter is a 4-byte window reference (the same as the `Handle` property of a window in `⎕WI`), and the second is a pointer to a Pascal string of up to 255 characters. The function exists in the 'CarbonLib' library under MacOS 9 and MacOS X. To declare and use this function in *APLX for MacOS*, you could write:

```
      'SETTITLE' ⎕NA 'Carbon.framework|SetWTitle U <P[255]'
      'W' ⎕WI 'New' 'Window'
      H←'W' ⎕WI 'Handle'
      SETTITLE H 'My new window'
```

You can also use the `GetWTitle` call to return the window's title as a Pascal string. The function prototype is:

```
void GetWTitle (WindowRef window, Str255 title);
```

To use this from APL, you could type:

```
      'GETTITLE' ⎕NA 'Carbon.framework|GetWTitle U >P[255]'
      GETTITLE H ''
My new window
```

Example 6 (MacOS Shared Library)

This example shows a more complex case involving a structure. Under MacOS, the `FSMakeFSSpec` function is used to take a filename, a volume reference, and a directory reference, and fill in a structure of type `FSSpec` which can be used to access the file. The function prototype given in the MacOS documentation is:

```
OSErr FSMakeFSSpec (short vRefNum, long dirID, ConstStr255Param fileName, FSSpec
*spec);
```

where `OSErr` is a two-byte signed integer, and the `FSSpec` structure is:

```
struct FSSpec {
   short          vRefNum;
   long           parID;
   StrFileName    name; /* a Str63 on MacOS*/
};
```

(This structure is aligned according to 68000-style alignment rules, so there is no padding). The third item in the structure is a Pascal string of up to 63 characters.

You could define and use this in APLX as:

```
            ⎕NA 'I2 Carbon.framework|FSMakeFSSpec I2 I4 <P[255] >{I2 I4 P[63]}'
            FSMakeFSSpec 0 0 ':APLX' (0 0 '')
      0  ¯2  ¯2078211007  APLX
```

In this example, we have passed in 0 for both the `vRefNum` and `dirID` parameters, and `':APLX'` for the Pascal string `fileName`. We have used `(0 0 '')` as the dummy argument for the `spec` parameter pointer, which receives the result. The result is a two element nested vector. The first element is the function return code, in this case `0`. The second element is a nested array representing the structure. MacOS has filled in this structure with `¯2` for `vRefNum`, `¯2078211007` for `parID`, and `'APLX'` for the Pascal string `name`.

Example 7 (MacOS X Framework)

Under MacOS X, a standard Unix-style programming environment exists in addition to the familiar Macintosh user-interface. This can be acccessed through the `System.framework` interface. An example of a simple routine in `System.framework` is the `gethostname` function, which returns the network name of the Macintosh system. The function prototype given in the MacOS (BSD) documentation is:

```
int gethostname(char *hostname, int namelen);
```

where `hostname` is the address of a buffer where the returned string should be placed, and `namelen` is the length of the buffer. The function returns 0 if it is successful, else an error code. To declare this function in *APLX for MacOS*, you could write:

```
            ⎕NA 'I System.framework|gethostname >C[256] I'
            gethostname '' 256
      0 PowerMacG4
```

APLX has returned a two-element vector, with the first element being the explicit result 0, and the second the string which has been placed in the 256-long buffer. This example will not work under MacOS 9, which does not implement frameworks or the BSD interface.

Errors

Note that, when you define the function using ⎕NA, only minimal error checking is done; if the routine descriptor is invalid you will get a DEFN ERROR when you try to use the function. If the library cannot be found, LOGICAL UNIT NOT FOUND will be generated. If the library is found, but the function name is not exported by the library, APLX will generate COMPONENT NOT FOUND. If the argument given to the function does not match the routine descriptor, APLX will report DOMAIN ERROR or possibly LENGTH ERROR.

Finally, you should be aware that a descriptor which does not correctly match the parameters expected by the external function may cause a fatal error and possibly crash APL.

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the APLX interpreter itself (the "Server") runs on one system, and the front-end which implements the user-interface (the "Client") runs either on a different machine, or under a different operating environment on the same machine. Typically, the Client might be a 32-bit Windows application running on a desktop PC, and the Server might be a 64-bit interpreter (APLX64) running either on the same machine, or on a different machine (such as a Linux server) over a network.

In such systems, ⎕NA allows you to specify whether the call should take place on the Client or Server side. You do this by preceding the ⎕NA function descriptor with either an Up Arrow ↑ to indicate that the call should take place on the Client, or a Down Arrow ↓ to indicate that the call should take place on the Server. If you do not specify, the default is that the call should take place on the Client.

In practice, there are two main cases to consider:

Client and Server run on the same physical machine

This case occurs if you are running APLX64 on a desktop 64-bit PC, because the front-end (Client) is still a 32-bit application, and the interpreter (Server) is a 64-bit application. Thus, you make 32-bit operating-system or library calls from the Client side, and 64-bit operating-system or library calls from the Server side.

For example, consider the Windows operating-system call GetSystemInfo. This takes as an argument a pointer to a SYSTEM_INFO structure, which it fills in with details of the current system:

```
typedef struct _SYSTEM_INFO {
    DWORD dwOemId;
    DWORD  dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD  dwActiveProcessorMask;
    DWORD  dwNumberOfProcessors;
    DWORD  dwProcessorType;
    DWORD  dwAllocationGranularity;
    WORD   wProcessorLevel;
    WORD   wProcessorRevision;
} SYSTEM_INFO;
```

The fields of type `WORD` and `DWORD` are 16-bit and 32-bit unsigned integers respectively. The fields of type `LPVOID` are pointers, i.e. 32-bit values in 32-bit versions of Windows, and 64-bit values in 64-bit versions of Windows. Under Windows XP64, 32-bit applications (such as the Client program) run within a virtual 32-bit environment, and access different versions of the operating-system libraries (confusingly, the 32-bit versions reside in `C:\WINDOWS\SysWOW64`, and the 64-bit versions in `C:\WINDOWS\system32`). We can therefore make either the 32-bit or 64-bit version of this system call from within *APLX64 for Windows*:

Making the 32-bit call:

```
      ⎕NA 'kernel32|GetSystemInfo >{U4 U4 U4 U4 U4 U4 U4 U4 U2 U2}'
      GetSystemInfo ''
 0  4096  65536  2147418111  1  1  586  65536  15  9218
```

Making the 64-bit call:

```
      'GetSystemInfo64' ⎕NA '↓kernel32|GetSystemInfo >{U4 U4 U8 U8 U4 U4 U4 U4 U2
U2}'
      GetSystemInfo64 ''
 9  4096  65536  8796092956671  1  0  1  8664  0  1
```

In the second (64-bit) example, the down arrow has been used to force the call to take place on the Server side. Note that the third and fourth elements in the structure (the minimum and maximum application addresses) are 32-bit integers (U4) in the 32-bit version, and 64-bit integers (U8) in the 64-bit version. Note also that the fourth value returned, which is the maximum application address, is limited (by Windows) to 2GB in the 32-bit case.

Client and Server run on different physical machines

In this case, any `⎕NA` calls made on the Client side will occur on the desktop machine, and any calls made on the Server side will occur on the remote server machine. The two systems may be running different operating systems and possibly even run under completely different processor architectures.

As an example, suppose you are running the APLX64 Server on a twin-processor Xeon Linux 64-bit system, and running the 32-bit APLX client on a 32-bit Windows PC. In these circumstances, you can define two external function, one which makes a call similar to Example 3 above (a 32-bit Windows call) on the Client machine, and another which makes a 64-bit Linux call on the Server:

```
      'GetSystemDirectory' ⎕NA '↑U kernel32|GetSystemDirectoryA >C[256] U'
      GetSystemDirectory '' 255
 17 C:\WINNT\system32
       ⎕NA '↓/lib64/libc.so.6|getcwd >C[512] U8'
      getcwd '' 512
/home/david/aplx64
```

⎕NAPPEND Append data to a native file

The ⎕NAPPEND function is used to append data to the end of a tied file. The file must have been opened in a mode which permits writing. The data may be of any simple data type; nested and mixed arrays are not permitted. The data may be of any shape or rank but will be ravelled before writing to file.

The syntax of ⎕NAPPEND is :

```
          DATA ⎕NAPPEND TIENO {CONV}
```

where CONV is an optional parameter specifying that the data should be converted to a different form before being written (see ⎕NWRITE for the conversion codes).

⎕NAPPEND is provided for compatibility with some other APL interpreters. The same facility is provided in a more general form by the ⎕NWRITE function.

⎕NC Name Classification

The monadic system function ⎕NC takes a right argument which is a character string specifying a name, or character matrix of names. It returns a numeric result which is the classification of each name:

```
           1       =        line label
           2       =        variable
           3       =        function
           4       =        operator
           5       =        group
           9       =        class
           0       =        name not in use
          ¯1       =        invalid name
```

Note: If ⎕CS is non-zero, an invalid name may be represented by 4 for compatibility with APL.68000 Level I. See ⎕CS for details.

⎕NCREATE Create a new native file and tie it

This function is used to create a new native file and associate it with the supplied tie number, or one allocated automatically by APLX. You must have the correct host access permission to create the file. The file is automatically opened for reading and writing. If the file already exists, its length is truncated to zero and its owner, group and access permissions are unchanged. Otherwise the file's owner and group are set to the effective owner ID and group ID of the user and the access permissions are set from the PERMISSIONS argument (see below).

The syntax of ⎕NCREATE is ({} means optional) :

```
          'FILENAME' ⎕NCREATE TIENO {,PERMISSIONS}
```

'FILENAME' is a character vector specifying the name of the file to create. The name may be either a full host path name, or the path may be omitted in which case the new file is created in the current working directory.

TIENO is an arbitrary non-zero integer to be used as the tie number in subsequent read/write operations to identify the file. It must not currently be in use to tie another file.

Alternatively, TIENO can be zero. In this case, APLX allocates the next available free tie number and returns it as the explicit result of the function. (The value returned is equivalent to the expression `1+⌈/0,⎕NNUMS` before the function is created).

PERMISSIONS is used to specify the host access permissions for a newly created file (see below). The value of PERMISSIONS is ANDed with the complement of the user's file mode creation mask to determine the resulting access permissions for the new file. The default value is 438 - read and write but no execute access for all three categories of user.

Access to a host file can be set for three categories of user, the owner of the file, other members of the owner's group, and other users of the system. For each of the three categories of user permission to read, write and execute the file can be granted or denied. The access permissions for a file are shown by the Unix command 'ls `-l'`, which displays them using a 9-character string format to denote the three sets of three permissions. The first set refers to the owner's permissions; the next refers to the permissions off others in the user- group of the file, and the last to all others. For example :

rwxr-x--x User has read, write and execute permissions Group members have read and execute permissions Other users have only execute permission

The PERMISSIONS parameter is formed by adding any combination of the following values :

```
     256   - read by owner
     128   - write by owner
      64   - execute by owner
      32   - read by group
      16   - write by group
```

```
        8    - execute by group
        4    - read by others
        2    - write by others
        1    - execute by others
```

In order to obtain the access permissions for the new file, the value of PERMISSIONS is ANDed with the complement of the user's file mode creation mask. This mask, set by the shell command 'umask', determines the default access mode for created files.

For example, if the current user's file mode creation mask is 2 and the value of PERMISSIONS is 438 (the default), the created file will have access permissions rw-rw-r--.

Note that if the file already exists its access permissions are not changed by `⎕NCREATE`

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine.

In such systems, you can specify whether the file should be created on the Client or the Server machine. You do this by preceding the path name with either an Up Arrow ↑ to indicate that the file should be created on the Client, or a Down Arrow ↓ to indicate that it is should be created the Server. (If you do not specify, the default is the Client.)

`⎕NERASE` Erase a native file

The `⎕NERASE` function allows you to erase a file from the host file system. You must have the correct host access permission to erase the file. If the file is currently tied it will be untied before attempting to erase it.

The usual syntax of `⎕NERASE` is :

```
          ⎕NERASE 'FILENAME'
```

'FILENAME' is a character vector specifying the name of the file to erase. The name may be either a full host path name, or the path may be omitted in which case the file to be erased must be located in the current working directory.

For compatibility with some other APL interpreters which require the file to be tied before it can be erased, an alternate syntax for `⎕NERASE` is supported :

```
     'FILENAME' ⎕NERASE TIENO
```

⎕NERROR Return an error message describing the last file error

The niladic ⎕NERROR function returns a character vector with a message giving details of the last error from the native file system. The error message is set whenever a native file system function returns a FILE I/O ERROR, and only cleared by)CLEAR. Note that it is not affected by native file functions which complete without error.

⎕NEW Create new instance of class

Syntax:

```
    objectref ← ⎕NEW classref
    objectref ← ⎕NEW classref param1 param2..
    objectref ← ⎕NEW 'classname'
    objectref ← ⎕NEW 'classname' param1 param2..
    objectref ← 'env' ⎕NEW 'classname'
    objectref ← 'env' ⎕NEW 'classname' param1 param2..
```

The system function ⎕NEW is the principal means by which you create an object, i.e. an instance of a class. The class can be either written in APL (an *internal* or *user-defined* class), or a built-in System class, or a class written in an external environment such as .Net, Java or Ruby (an *external* class). ⎕NEW creates a new instance of the class, runs any constructor defined for the class, and returns a reference to the new object as its explicit result.

The class is specified as the right argument (or first element of the right argument). It can be specified either as a class reference, or as a class name (i.e. a character vector). Any parameters to be passed to the *constructor* of the class (the method which is run automatically when a class is created) follow the class name or reference.

If you specify the class by name, you also need to identify the environment where the class exists, unless it is internal. The left argument is a character vector as follows (the keywords are not case-sensitive):

For 32-bit implementations of APLX:

Left arg	***Environment***	***Windows DLL***	***Macintosh bundle***	***Linux shared library***
`'apl'` or omitted	User-defined APL class	None	None	None
`'⎕'`	System-defined class (⎕WI)	None	None	None
`'.net'`	Microsoft .Net	`aplxobj_net.dll`	Not supported	Not supported

`'java'`	Java	`aplxobj_java.dll`	`aplxobj_java.bundle`	`aplxobj_java.so`
`'r'`	R	`aplxobj_r.dll`	`aplxobj_r.bundle`	`aplxobj_r.so`
`'ruby'`	Ruby	`aplxobj_ruby.dll`	`aplxobj_ruby.bundle`	`aplxobj_ruby.so`
Other	Customized environment	`aplxobj_XXX.dll`	`aplxobj_XXX.bundle`	`aplxobj_XXX.so`

For 64-bit implementations of APLX:

Left arg	***Environment***	***Windows DLL***	***Linux shared library***
`'apl'` or omitted	User-defined APL class	None	None
`'⎕'`	System-defined class (`⎕WI`)	None	None
`'.net'`	Microsoft .Net	`aplx64obj_net.dll`	Not supported
`'java'`	Java	`aplx64obj_java.dll`	`aplx64obj_java.so`
`'r'`	R	`aplx64obj_r.dll`	`aplx64obj_r.so`
`'ruby'`	Ruby	`aplx64obj_ruby.dll`	`aplx64obj_ruby.so`
Other	Customized environment	`aplx64obj_XXX.dll`	`aplx64obj_XXX.so`

Creating instances of internal (user-defined) classes

Normally, you create an instance of a user-defined class by passing the class reference directly as the right argument (or first element of the right argument). For example, if you have a class called `Invoice`, you can create an instance of it by entering:

```
     I←⎕NEW Invoice
```

What is really happening here is that the symbol `Invoice` refers to the class definition, and when it is used in this way, it returns a *reference to the class*. You can see this explictly:

```
      Invoice     ⍝ Entering the name of a class returns a reference to that class
{Invoice}         ⍝ Class references are displayed as class name in curly braces
      C←Invoice   ⍝ C contains a second reference to the class Invoice
      C
{Invoice}
      I2←⎕NEW C   ⍝ ... so we can use C as the argument to ⎕NEW
      I2.⎕CLASSNAME
Invoice
```

Note that you can also pass the class name rather than a class reference. The following are alternative ways of creating an instance of a user-defined class:

```
      I←⎕NEW 'Invoice'
      I←'apl' ⎕NEW 'Invoice'
```

Passing arguments to the constructor

A constructor is a special method of a class, which is run automatically when the class is created using `⎕NEW`, and is used to initialize the class. For APL classes, the constructor is a method whose name is the same as the name of the class. It should be a function which takes a right argument, and does not return a result.

Any arguments to the constructor can be provided as extra elements on the right argument of ⎕NEW. When the constructor is run, these extra elements are passed as the right argument to the constructor. If there are no extra elements, an empty vector is passed as the right argument to the constructor.

For example, suppose the class Invoice looks like this:

```
Invoice {
  TimeStamp
  Account
  InvNumber
  {Serial←0}

 ∇Invoice B
  ⍝ Constructor for class Invoice.  B is the account number
  Account←B
  TimeStamp←⎕TS
  Serial←Serial+1
  InvNumber←Serial
 ∇
}
```

When a new instance of this class is created, the constructor will be run. It will store the account number (passed as an argument to ⎕NEW) in the property Account, and store the current time stamp in the property TimeStamp. It will then increment the class-wide property Serial (common to all instances of this class), and store the result in the property InvNumber.

```
      S←⎕NEW Invoice 23533
      S.⎕DS
Account=23533, TimeStamp=2007 10 11 15 47 34 848, InvNumber=1
      T←⎕NEW Invoice 67544
      T.⎕DS
Account=67544, TimeStamp=2007 10 11 15 48 11 773, InvNumber=2
```

(To see the properties, we use the system method ⎕DS which summarizes the property values):

Creating instances of System classes

A System class is a pre-built class which is part of APLX. Examples are the Form, Timer, ChooseColor and Chart classes. (In previous versions of APLX, these classes were accessed through ⎕WI).

To create an instance of a top-level System class, you can provide the name of the class as the right argument, and use '⎕' as the left argument:

```
      DLG←'⎕' ⎕NEW 'ChooseColor'
      DLG.⎕NL 3
Close
Create
Delete
New
Open
Send
Set
Show
Trigger
```

You can then use dot notation to access the properties and methods of the object:

```
      DLG.color←234 23 56
      DLG.Show
1
```

Creating instances of External classes

To create an instance of an external class, you call `⎕NEW` with the class name as the right argument (possibly followed by constructor arguments), and the environment keyword (e.g. `'.net'`, `'java'`, or `'ruby'`) as the right argument. (In some cases, you may need to call `⎕SETUP` first, to set up environment parameters such as the search path and namespace prefix).

Create an instance of the .Net `DateTime` class, defined in the .Net `System` namespace:

```
      NETDATE←'.net' ⎕NEW 'System.DateTime'  2007 6 20 9 32 3
```

Create an instance of the Ruby `DateTime` class, defined in the Ruby class library `Date`:

```
      'ruby' ⎕SETUP 'require' 'Date'
      RUBYDATE←'ruby' ⎕NEW 'DateTime'  2007 6 20 9 32 3
```

Create an instance of the Java `Date` class:

```
      JAVADATE←'java' ⎕NEW 'java.util.Date' 2007 6 20 9 32 3
```

Use the `⎕DS` System method to convert each of the different objects to string form:

```
       (3 1⍴NETDATE,RUBYDATE,JAVADATE).⎕DS
 20/06/2007 09:32:03
 2007-06-20T09:32:03+00:00
 Sat Jul 20 09:32:03 BST 3907
```

Using references to external classes

You can also obtain a reference to an external class, and use that instead of the class name (and left argument) to specify the class. For example:

```
      NetDateClass←'.net' ⎕GETCLASS 'DateTime'
      NetDateClass
{.net:DateTime}
      START←⎕NEW NetDateClass 2004 12 4 12 34 2
      START.⎕DS
04/12/2004 12:34:02
```

Because there may be a significant overhead in searching in the external environment for the class name, this method is likely to be more efficient if your application needs to create many different instances of a given class.

Object references and object lifetimes

When you use ⎕NEW to create a new object, that object persists until there are no more references to it in the workspace. It is then deleted immediately, if it is an internal or system object. If it is an external object, such as an instance of a .Net class, the fact that there are no more references to it in the APL workspace means that it available for deletion by the external environment (unless the external environment itself has further references to the same object). However, in typical external environments such as .Net, Java and Ruby, the actual deletion of the object may not occur until later.

Consider this sequence, where we create an instance of a class called `Philosopher` which has a property `Name`:

```
      A←⎕NEW Philosopher
      A.Name←'Aristotle'
```

At this point, we have created a new instance of the class, and we have a single reference to it, in the variable `A`. We now copy the *reference* (not the object itself) to a variable `B`:

```
      B←A
      B.Name
Aristotle
```

We now have two references to the same object. So if we change a property of the object, the change is visible through either reference - they refer to the same thing:

```
      B.Name←'Socrates'
      A.Name
Socrates
```

Now we erase one of the references:

```
      )ERASE A
```

We still have a second reference to the object. The object will persist until we delete the last reference to it:

```
      B.Name
Socrates
      )ERASE B
```

At this point, there are no more references to the object left in the workspace, and the object itself is deleted.

It follows from this that, if you use ⎕NEW to create an object, and do not assign the result to a variable, it will immediately be deleted again. In this example, we create an instance of the class Philosopher. The explicit result of ⎕NEW is a temporary workspace entry (of type object reference), which is displayed using the default display format for objects, and then deleted. AT that point the object itself is also deleted, as there are no references left:

```
      ⎕NEW Philosopher
[Philosopher]
```

Customized interfaces

The ⎕NEW mechanism also allows the same APL syntax to be used for accessing other object-oriented environments or custom class libraries written in languages such as C++. For example, you wanted to make use of a timeseries analysis class library written in C++, you could write a simple interface DLL `aplxobj_ts.dll` which would allow classes contained in that library to be used from APL:

```
      TS←'ts' ⎕NEW 'TimeSeries'
```

The APLX interpreter, seeing this line, would pick up the environment identifier 'ts' which would cause it to search for `aplxobj_ts.dll` to handle the creation and use of the classes within the custom library. Contact MicroAPL for more information on how to write customized interfaces.

⎕NL Name List

The nomadic system function ⎕NL returns a character matrix of sorted names of the type specified as one or more integer codes in the right argument:

Code	***Type of name***
2	Variables
3	Functions
4	Operators
5	Groups
9	Classes

For example:

```
             ⎕NL 3                       (List of functions)
             ⎕NL 2 3                     (List of functions and variables)
```

If a letter, or letters, are included as the left argument, only objects starting with that letter are listed.

⎕NL is also a system method, which can be used to list the properties, methods and constructors of an object or class. See the documentation for `⎕NL Names of members`.

⎕NLOCK Lock/Unlock a file or a segment of a file

The ⎕NLOCK function allows the user to lock a file or a segment of a file. Setting a file lock prevents another user from locking the same file or segment in an incompatible mode.

The syntax of ⎕NLOCK is ({} means optional) :

```
          ⎕NLOCK TIE, MODE {,COUNT {,STARTBYTE {,TIMEOUT}}}
```

TIE specifies the file to lock or unlock

MODE specifies the lock operation to perform, as follows:

```
          0 - unlock file or segment
          1 - establish a read lock on a file or segment
          2 - establish a write lock on a file or segment
```

COUNT specifies the length in bytes of the segment to lock or unlock. The default value of ¯1 will set a lock from the specified start position to the current and any future end of file.

STARTBYTE specifies the offset from the start of the file at which the lock segment begins. A value of ¯1 (default) specifies the start of the file.

TIMEOUT is used to specify a request timeout value in milliseconds (see below). A value of ¯1 specifies no timeout. The default value of 10000 is chosen to specify a sensible 10 second default.

The implementation of ⎕NLOCK differs slightly on AIX and Linux systems from other APLX systems. Under AIX and Linux, file locking facilities provide controlled multi-user file access for cooperating processes which agree on a locking scheme; locks do not prevent another user who ignores the scheme from reading or writing the file. That is, a write lock set on a segment of a file prevents another user from setting a write lock on the same file segment, but does not prevent writing data to the segment.

Two lock modes are supported. A read lock prevents other users from write- locking the file or segment. A write lock prevents other users from read-locking or write-locking the file or segment. An attempt to use ⎕NLOCK to set a lock on a file or segment which is already locked in an incompatible mode by another user will fail with a FILE I/O ERROR, and the ⎕NERROR message gives details of the reason for the failure. To set a read lock the file must be tied in a mode which permits reading; to set a write lock the file must be tied in a mode which permits reading and writing.

Using ⎕NLOCK it is possible to lock a whole file or a specified segment. The extent of the segment to lock is specified by a combination of the COUNT and STARTBYTE parameters. By using a COUNT value of ¯1 it is possible to set a lock which extends from the specified start position to the current and any future end of file. It is also valid to specify a lock segment which starts beyond the current end of file. It is possible to have locks on several segments of a file simultaneously.

Locking a segment which is already locked by the user causes the old lock type to be removed and the new lock type to take effect. Changing or unlocking a portion from the middle of a larger locked segment leaves a smaller segment at either end.

The TIMEOUT parameter is used to specify a timeout value for the lock request. If the requested lock cannot be immediately granted because another user has locked the segment in an incompatible mode, the process will sleep for TIMEOUT milliseconds waiting for the segment to become free. If a TIMEOUT value of 0 is specified, the lock request will give an error immediately if the lock cannot be granted. A value of `¯1` causes the process to sleep forever. The default value of 10000 causes the process to sleep for up to ten seconds. A process that is sleeping while waiting for a lock can be woken with an interrupt.

The host locking mechanism detects deadlock situations. If a lock request cannot be immediately granted because another process has the segment locked, and the other process is sleeping on a request for a segment which the user has locked, the potential for a deadlock exists. In this case the requested lock will be refused. Note however that deadlocks are not always detected in a distributed file system. When such deadlocks are possible `⎕NLOCK` should always be used with a timeout value.

All locks set on a file are automatically cleared by `⎕NUNTIE`, `)CLEAR` and `)OFF`. If the same file is tied on more than one tie number, untying it on any tie will clear all the locks set on the file.

Note that there is a limit on the total number of locks a process may have at one time; this limit is imposed by the host operating system.

Examples:

Set exclusive lock on the whole of the file tied on tie number 100. All other users are prevented from locking any current or future segment of the file:

```
      ⎕NLOCK 100 2
```

Try to get a read lock on the first 1000 bytes of the file, with a timeout value of 2 seconds:

```
      ⎕NLOCK 100 1 1000 0 2000
```

Release all locks on the file:

```
      ⎕NLOCK 100 0
```

⎕NNAMES Return file names of all tied files

The niladic ⎕NNAMES function returns a character matrix identifying all the files currently tied. The rows of the result have the same order as the tie numbers returned by ⎕NNUMS. Note that each row gives the full path name of a currently tied file, irrespective of whether a full or relative path was specified when the file was tied.

⎕NNUMS Return tie numbers of all tied files

The niladic function ⎕NNUMS returns an integer vector of all the tie numbers currently in use. The ordering of the result is the same as the ordering of the rows of the result of ⎕NNAMES.

⎕NREAD Read data from a native file

The ⎕NREAD function allows you to read data from anywhere in the file, specifying an optional start byte, count and conversion mode. The file must have been opened in a mode which permits reading. The full syntax is ({} means optional) :

```
          R ← ⎕NREAD TIENO {,CONV {,COUNT {,STARTBYTE}}}
```

TIENO is the tie number associated with the file to read

CONV specifies any conversion to apply to the data - e.g. read as raw data, translated characters, 4-byte integers, booleans, etc. The default is to read the file as raw character data. See below for details.

COUNT specifies the number of elements to read (except when CONV = 11, see below). The number of bytes read from the file will depend on the conversion mode used (see below). A value of ¯1 (default) specifies read to end-of-file.

STARTBYTE may be used to specify the offset in bytes from the beginning of the file at which to start reading data. A value of ¯1 (default) specifies the current file position, the position at which the last successful read or write operation completed.

The conversion mode parameter CONV can be used to read data very easily from a file with a known structure. Data can be read as raw bytes, translated characters, booleans, integers or floating point numbers. In addition byte-swapping facilities allow data to be read from a file created on a host with different local byte-ordering conventions. The full list of supported values for the conversion mode is as follows :

```
   Normal modes:

   0      read data as a stream of raw bytes
   1      read data as booleans, 1 bit per element
   2      read data as 32-bit integers
   3      read data as 64-bit IEEE double-precision floating point numbers
   4      read character data and translate from external representation
          to APLX's own internal format
   5      read Unicode UTF-16 characters (two bytes per element), and convert to
          APLX internal representation as characters.  Any Unicode values which
          cannot be mapped to APLX characters are converted to the value set
          by ⎕MC (by default, question mark).
   6      read data as 32-bit IEEE single-precision floating point numbers
   8      read Unicode UTF-8 characters (variable bytes per element), convert to
          APLX internal representation as characters.  Any Unicode values which
          cannot be mapped to APLX characters are converted to the value
          set by ⎕MC (by default, question mark).

   Byte-swapped modes:

  ¯2      read data as 32-bit byte-swapped integers
  ¯3      read data as 64-bit byte-swapped floats
  ¯5      read data as byte-swapped Unicode characters
  ¯6      read data as 32-bit byte-swapped floats
```

For compatibility with some other APL interpreters the following conversion specifiers are also supported :

```
  11      read data as booleans (same as mode=1)
  82      read data as raw characters (same as mode=0)
 163      read data as 16-bit integers.   Values are converted to 32-bit
          integers before being returned. They are treated as unsigned.
 323      read data as 32-bit integers (same as mode=2)
 325      read data as 32-bit floating point numbers (same as mode=6)
 645      read data as 64-bit floating point numbers (same as mode=3)
¯163      read data as unsigned 16-bit integers with byte-swapping
¯323      read data as 32-bit integers with byte-swapping (same as mode=¯2)
¯325      read data as 32-bit floats with byte-swapping (same as mode=¯6)
¯645      read data as 64-bit floats with byte-swapping (same as mode=¯3)
```

Under APLX64, the following additional conversion types are available:

```
   7      read data as 64-bit integers
  ¯7      read data as 64-bit integers with byte-swapping
 643      read data as 64-bit integers (same as mode=7)
 643      read data as 64-bit integers with byte-swapping (same as mode=¯7)
```

The optional COUNT specified in the ⎕NREAD argument relates to the number of elements to read, not necessarily the number of bytes. For example when reading 32-bit integers, the number of bytes read will be four times the value of COUNT. Note that when reading boolean data with CONV = 11, the value of COUNT specifies the number of bytes to read rather than the number of bits. This is done for compatibility with some other APL interpreters.

Note that all file i/o operations start on a byte boundary. In particular, following a boolean read operation that returns a non-integral number of bytes, the current file position will be aligned on the next byte boundary.

Two possible errors may occur when specifying an inappropriate value of COUNT or CONV. If the number of bytes remaining in the file is insufficient to satisfy the request, a FILE I/O ERROR occurs and the current file position is unchanged. For example it is an error to try to read 10000 bytes from a file with only 9000 bytes remaining :-

```
     ⎕NREAD 100 0 10000
 Insufficient data available
 FILE I/O ERROR
      ⎕NREAD 100 0 10000
      ^
```

To read all data up to the end of file, omit the count parameter or specify it as `¯1`.

A second type of error can arise when trying to read integers or floats. If the count is not explicitly specified and the number of bytes remaining in the file is not an exact multiple of the element size, a FILE I/O ERROR occurs and the current file position is again unchanged. For example, if there are 23 bytes remaining in the file an attempt to read them as 4-byte integers will fail since there is too much data for five integers and not enough for six:

```
     ⎕NREAD 100 2
 Wrong number of bytes remain for data type requested
 FILE I/O ERROR
      ⎕NREAD 100 2
      ^
```

Examples:

Read all bytes from current file position to end of file as raw data:

```
     ⎕NREAD 100
```

Read from current file position to end of file as integers:

```
     ⎕NREAD 100 2
```

Read next ten integers from file:

```
     ⎕NREAD 100 2 10
```

Read ten floats starting at offset 20 bytes from start of file:

```
     ⎕NREAD 100 3 10 20
```

Reading Unicode UTF-16 text files

By convention, Unicode UTF-16 plain-text files start with a 'byte-order' mark. This is the special hex value FEFF, represented as a two-byte value in the byte-ordering used to create the file. Thus, on 'big-endian' systems such as the Macintosh, the first two bytes of the file will normally be hex FE and FF (decimal 254 and 255). On a 'little-endian' system such as Windows or x86 Linux, numbers are represented backwards so the first two bytes will normally be FF and FE.

You can use this information to determine whether to use the conversion type `5` or `¯5` when reading the contents of a UTF-16 text file, by reading the first element of the file as a 16-bit integer (conversion code 163 for `⎕NREAD`. If you get the value 65279 (hex FEFF), the Unicode file was written using the same byte-ordering as the machine you are running on, so no byte reversal is required and you can use conversion code `5` to read the Unicode characters from the remainder of the file. If you get the value 65534 (hex FFFE), the Unicode file was written using the opposite byte-ordering convention to that of the machine you are using, so you need to use conversion code `¯5`. For example:

```
      'c:\temp\uni.txt' ⎕NTIE 1    ⍝ Open a UTF-16 text file
      ⎕NREAD 1 163 1 0             ⍝ Read first two bytes as 16-bit integer
65279                              ⍝ This is the correct value for hex FEFF
      ⎕AF 4 ⎕DR 65279
0 0 254 255
      TEXT←⎕NREAD 1 5 ¯1           ⍝ Read the remainder of the file as Unicode
      ⎕NUNTIE 1
```

If you want to read UTF-16 files without converting them to APLX characters, use conversion type `163` or `¯163`, to read them as 2-byte (unsigned) integers, with byte-swapping if necessary. This allows you to process Unicode values which cannot be represented in the APLX character set. If you later need to convert the returned integer values to APLX text, use `⎕UCS`.

See also `⎕MC`, which contains the character used to replace Unicode characters which cannot be represented in APLX.

⎕NRENAME Change the name of a native file

The `⎕NRENAME` function allows you to rename a file in the host file system. You must have the correct host access permission to rename the file. If the file is currently tied, the name change will be reflected in the name list returned by `⎕NNAMES`.

Two alternative syntax forms are supported for `⎕NRENAME` :

```
          'NEWNAME' ⎕NRENAME 'OLDNAME'
or
          'NEWNAME' ⎕NRENAME TIENO
```

'OLDNAME' is a character vector specifying the file to rename. The name may be either a full host path name, or the path may be omitted in which case the file to be renamed must be located in the current working directory.

'NEWNAME' is a character vector specifying the new name of the file.

TIENO: The second syntax form is an alternative way to rename a file which has already been tied. TIENO is the tie number on which the file is tied.

Note that the host operating system may not allow you to rename a file across different file systems or physical disks.

Special considerations for Client-Server implementations of APLX

See `⎕NCREATE` for the conventions applying to file locations in Client-Server versions of APLX.

`⎕NREPLACE` Replace data in a native file

The `⎕NREPLACE` function is used to replace data in a tied file. The file must have been opened in a mode which permits writing. The data may be of any simple data type; nested and mixed arrays are not permitted. The data may be of any shape or rank but will be ravelled before writing to file. Data is written starting at the specified file position and the file is extended if necessary.

The syntax of `⎕NREPLACE` is `({}` means optional) :

```
          DATA ⎕NREPLACE TIENO {,STARTBYTE} {,CONV}
```

TIENO is the tie number associated with the file to write to

STARTBYTE may be used to specify the offset in bytes from the beginning of the file at which to start writing data. A value of `¯1` (default) specifies the current file position.

CONV is an optional parameter specifying that the data should be converted to a different form before being written (see `⎕NWRITE` for the conversion codes).

`⎕NREPLACE` is provided for compatibility with some other APL interpreters. The same facility is provided in a more general form by the `⎕NWRITE` function.

`⎕NRESIZE` Alter the size of a native file

The `⎕NRESIZE` function can be used to reserve space on disk for a file. The file must have been opened in a mode which permits writing. The syntax of `⎕NRESIZE` is :

```
          SIZE ⎕NRESIZE TIENO
```

TIENO is the tie number of file to resize

SIZE specifies the new size for the file in bytes

If the new size is larger than the current size of the specified file, the file is extended to the new size by writing bytes of zeros.

⎕NSIZE Return file size information

The ⎕NSIZE function returns the current size in bytes of a tied file as an integer scalar :

```
          R ← ⎕NSIZE TIENO
```

⎕NTIE Open an existing file and associate it with a tie number

This function is used to open an existing file and associate it with the supplied tie number, or with a tie number allocated by APLX. You must have the correct host access permission to open the file in the specified mode.

The syntax of ⎕NTIE is ({} means optional) :

```
          'FILENAME' ⎕NTIE TIENO {,MODE}
```

'FILENAME' is a character vector specifying the name of the file to tie. The name may be either a full host path name, or the path may be omitted in which case the file to tie must be located in the current working directory.

TIENO is an arbitrary non-zero integer to be used in subsequent read/write operations to identify the file. It can be positive or negative. It must not currently be in use to tie another file.

Alternatively, TIENO can be zero. In this case, APLX allocates the next available free tie number and returns it as the explicit result of the function. (The value returned is equivalent to the expression `1+⌈/0,⎕NNUMS` before the function is tied).

The optional MODE parameter indicates the access mode as follows : 0=read only, 1=write only, 2=read/write (default).

MacOS Resource Forks

Under *APLX for MacOS* prior to APLX Version 3.5, ⎕NTIE can optionally take a third argument indicating the file 'fork' to open. If this parameter is omitted or is 0, the data fork is opened. If this parameter is 1, the resource fork is opened.

Because Apple are phasing out the use of resource forks in MacOS X, this facility has had to be withdrawn from APLX Version 3.5 onwards.

Special considerations for Client-Server implementations of APLX

In Client-Server implementations of APLX, the front-end which implements the user-interface (the "Client") runs on one machine, and the APLX interpreter itself (the "Server") can run on a different machine.

In such systems, you can specify whether the file being accessed is located on the Client or the Server machine. You do this by preceding the path name with either an Up Arrow ↑ to indicate that the file is on the Client, or a Down Arrow ↓ to indicate that it is on the Server. (If you do not specify, the default is that the access takes place on the Client.)

⎕NTYPE Get/Set the file type/creator for a MacOS file

This function is used in MacOS to set the type and optionally the creator of a native file. The file type is a 4-character code which defines, under MacOS, the class of file. For example, 'TEXT' is the type for ordinary text files, and 'APPL' is the type for a program file. The creator of a file is also a 4-character code, and is used by the operating system to associate a file with an application. For example, when you double-click on a document file, the creator determines which application is started up in order to open the document.

The syntax of `⎕NTYPE` is:

```
          ⎕NTYPE FILENAME
          TYPE ⎕NTYPE FILENAME
or        TYPE ⎕NTYPE TIENO
```

The right argument is either a character vector specifying the name of the file (including the path), or an integer tie number if the file has already been tied using `⎕NTIE`. Omitting the left argument simply returns the current type and creator.

If you want to set just the file type, and not the creator, you supply a character vector of 4 characters as the left argument. If you want to set both the type and the creator, you can supply a 2-element nested vector; the first element is the type, and the second the creator. Both are 4-element character vectors.

Under operating systems other than MacOS, this system function exists in APLX, but it has no effect.

⎕NULL Return reference to null object

The niladic system function ⎕NULL returns a reference to the Null object, i.e. an object which has no properties or methods. It can be used as a placeholder for an uninitialized object.

⎕NUNTIE Untie native file(s)

The ⎕NUNTIE function is used to untie one or more native files. Any file locks set by ⎕NLOCK are released and the file is closed.

The syntax of ⎕NUNTIE is ({} means optional) :

```
          ⎕NUNTIE TIENO {,TIENO} {,TIENO} ...
```

where the right argument is an integer scalar or vector of tie numbers to untie.

Note that all files are automatically untied by a)CLEAR or an)OFF. File ties are not affected by)LOAD operation.

⎕NWRITE Write data to a native file

⎕NWRITE is used to write data to a tied file. The file must have been opened in a mode that permits writing. The syntax of ⎕NWRITE is similar to ⎕NREAD ({} means optional) :

```
          DATA ⎕NWRITE TIENO {,CONV {,STARTBYTE}}
```

TIENO specifies the file to write to

CONV specifies any conversion to apply to the data before writing it; default is no conversion (see below)

STARTBYTE specifies the offset from the start of the file at which to begin writing. A value of ¯1 (default) specifies the current file position. A value of ¯2 specifies that the data should be appended to end of file.

The DATA may be of any simple data type; nested and mixed arrays are not permitted. The data may be of any shape or rank but will be ravelled before writing to file.

The conversion mode parameter allows the user to specify a conversion to apply to the data before writing it to the file. It may be used to coerce data to a desired data type before writing. For example an array of booleans can be converted to integers before writing to file. A full list of possible values for CONV is as follows :

```
   Normal modes:

   0      write data unconverted
   1      write data as booleans, 1 bit per element
   2      write data as 32-bit integers
   3      write data as 64-bit IEEE double-precision floating point numbers
   4      write character data, translating from APLX's own internal
          representation to external format.
   5      write character data, translating from APLX's own internal
          representation to Unicode UTF-16 (2 bytes per element).
   6      write data as 32-bit IEEE single-precision floating point numbers
   8      write character data, translating from APLX's own internal
          representation to Unicode UTF-8 (variable bytes per element).

   Byte-swapped modes:

  ¯2      write data as 32-bit byte-swapped integers
  ¯3      write data as 64-bit byte-swapped floats
  ¯5      write data as byte-swapped Unicode characters
  ¯6      write data as 32-bit byte-swapped floats
```

For compatibility with some other APL interpreters the following conversion specifiers are also supported:

```
   11      write data as booleans (same as mode=1)
   82      write data as raw characters (same as mode=0)
  163      write data as 16-bit integers.
  323      write data as 32-bit integers (same as mode=2)
  325      write data as 32-bit floating point numbers (same as mode=6)
  645      write data as 64-bit floating point numbers (same as mode=3)
 ¯163      write data as 16-bit integers with byte-swapping
 ¯323      write data as 32-bit integers with byte-swapping (same as mode=¯2)
 ¯325      write data as 32-bit floats with byte-swapping (same as mode=¯6)
 ¯645      write data as 64-bit floats with byte-swapping (same as mode=¯3)
```

Under APLX64, the following additional conversion types are available:

```
    7      write data as 64-bit integers
   ¯7      write data as 64-bit integers with byte-swapping
  643      write data as 64-bit integers (same as mode=7)
 ¯643      write data as 64-bit integers with byte-swapping (same as mode=¯7)
```

An inappropriate combination of source data type and conversion mode will lead to a DOMAIN ERROR. For example an attempt to convert the following floating point vector to integers will fail :

```
      (1.0 1.5 2.0) ⎕NWRITE 100 2
 DOMAIN ERROR
       (1.0 1.5 2.0) ⎕NWRITE 100 2
       ^
```

In the event of an illegal conversion, no data is written to disc and the current file position is unchanged.

When writing booleans, note that the number of elements to write is padded with zeros to give a whole number of bytes before writing to file.

⎕OV Overlay

The dyadic system function ⎕OV allows several variables, functions and operators to be grouped together to form a single item (known as an *overlay*) within the workspace. The overlay can then be stored, and later unpacked and dispersed. An overlay in a workspace behaves like an empty vector, but occupies the space taken up by its members. Overlays are mostly used in conjunction with files, where it is useful to be able to store a number of APLX variables, functions, and operators (including locked objects) as one file component.

An overlay is created by a statement of this form:

```
            OV←0 ⎕OV  NAMETABLE
```

where NAMETABLE is a character matrix of the names of the variables, functions, and operators to be placed in the overlay, or a character vector if only one item is to to be placed in the overlay,.

The contents of the overlay are dispersed by statements of this form:

```
            1 ⎕OV OV
            2 ⎕OV OV
```

OV is an overlay. The left argument 1 specifies that the overlay should be dispersed with functions and operators locked. The left argument 2 specifies that unlocked functions and operators should be left unlocked. Both return a matrix of the names of the objects dispersed.

A statement of this form:

```
            3 ⎕OV OV
```

returns a character matrix of the names of the items in the overlay.

```
            (0 ⎕OV ⎕NL 2 3)⍈1 1          (Write all functions and variables to
                                          file 1 component 1)
            VAR←0 ⎕OV ⎕BOX 'BILL JOE'    (Make up an overlay)
            ⍴VAR                         (Looks like an empty vector)
      0
            3 ⎕OV VAR                    (What is in the overlay)
      BILL
      JOE
```

Notes: The maximum total size of the objects in an overlay is 16MB. This is to ensure compatibility with previous APL.68000 releases. Classes and class members cannot currently be placed in an overlay.

Overlays in APLX64

Overlays are implemented in 64-bit versions of APLX; the format of the overlay is unchanged from the 32-bit version. They can therefore be used for exchanging data in both directions with 32-bit implementations of APLX.

In order to retain this compatibility, functions and variables placed in overlays using `⎕OV` are converted to 32-bit form. This means that arrays containing 64-bit integers will be converted to floating-point if they cannot be represented as 32-bit integers. Loss of precision will occur for any integers of magnitude bigger than `2*53`.

⎕PFKEY Set up Function keys

The system function `⎕PFKEY` function allows you to associate your own strings with function keys. It takes a left argument which is the character string you want to associate with the key (with a closing `⎕R` or `⎕TCNL` character if you want to simulate pressing Return at the end). The right argument is a numeric scalar giving the key number. The range 1 to 15 corresponds to function keys F1 to F15. Add 15 to the value to represent the shifted key, and 30 to the value to represent the key with Control held down. For example, the following two commands would cause F5 to execute a `)WSID` command and Shift-F5 to execute a `)TIME` command, in the session window:

```
      (')WSID',⎕R) ⎕PFKEY 5
      (')TIME',⎕R) ⎕PFKEY 5+15
```

The strings you attach can be used for any purpose you like, either as short-cuts in APL development, or within your application. However, be aware that some function-key combinations may be reserved for use by the operating system or window-manager, or as menu short-cuts. In this case programming the function key yourself may not work (because the operating-system may intercept it before APLX sees the keystroke), or may be incompatible with the user-interface guidelines for the system you are using.

The exact implementation of this system function varies according to the host system. Under MacOS, the Control-key strings are not effective. In Linux Server Edition (dumb-terminal), neither the Shift nor Control-key strings are effective.

Associating a sequence of strings with a function key

An extension to standard function-key programming allows you to associate a sequence of strings with a given function key. (This works with the Session window only). Each time you press the function key, the next string in the sequence is output to the session window, at the end of the current session. If you press the function key whilst Shift is held down, the previous string in the sequence is output. The sequence wraps round at the beginning and end.

To use this mode, supply a text matrix as the left argument of the `⎕PFKEY`. The right argument should be the function key number in the range 1 to 15. Each line of the matrix corresponds to a string in the sequence (trailing blanks are suppressed).

For example, you could program function key 2 as follows:

```
      strings←'/' ⎕BOX 'x←1/y←2/x run y'
      strings
x←1
y←2
x run y
      strings ⎕PFKEY 2
```

If you now press function key 2 four times in succession, the three strings will be output in turn, and then the sequence will wrap round to the first again on the fourth key press.

This facility is very useful for:

- Presentations, where you want to pre-store a set of lines (or fragments of lines) rather than typing them in during the talk
- Storing a set of useful commands which you can cycle through in order to select the one you want.

⎕PP Print Precision

The system variable ⎕PP contains the number of digits after the point in the default display of numeric values. The default is 10. ⎕PP can be changed by assignment to a value from 1 to 15. Assigning a larger value will cause the maximum allowed value (15) to be set.

Note: By default, numbers which are represented internally as integers, or which are floating-point numbers which could be repesented internally as exact integers, are displayed in full precision irrespective of ⎕PP. In 32-bit versions of APLX, this applies to all whole numbers whose magnitude is less than `2*31`.

In 64-bit versions of APLX, it applies to any number represented internally as an integer, and to any number which is held in floating-point representation and which is exactly equal to an integer of magnitude less than `2*53`.

⎕PR Prompt Replacement

The System Variable ⎕PR contains the character used to indicate the prompt portion of a ⍞ input line. The default value is blank. Under normal circumstances, if ⍞ output is used to place a prompt on a line and ⍞ input is then used to capture user input, the length of the output prompt is indicated by blanks. ⎕PR can substitute another fill character. For more details see ⍞.

```
      ⍞←'PLEASE TYPE YOUR NAME : ' ⋄ NAME←⍞
PLEASE TYPE YOUR NAME : MYNAME
      NAME
                        MYNAME
      ⎕PR←'*'
      ⍞←'PLEASE TYPE YOUR NAME : ' ⋄ NAME←⍞
PLEASE TYPE YOUR NAME : MYNAME
      NAME
************************MYNAME
```

If ⎕PR is set to an empty vector, then the actual contents of the prompt line are returned.

```
      ⎕PR←''
      ⍞←'PLEASE TYPE YOUR NAME : ' ⋄ NAME←⍞
PLEASE TYPE YOUR NAME : MYNAME
      NAME
PLEASE TYPE YOUR NAME : MYNAME
```

⎕PROFILE Performance Profiling

Performance profiling can be used to find out which parts of your APL code take the most time to execute, or are executed most often, and so helps you to determine which functions to concentrate on when optimising performance.

For simple profiling it is not necessary to use ⎕PROFILE. Instead you can enable profiling through the APLX Tools menu and then run the code to be profiled. When the code completes and APLX returns to desktop calculator mode, the profile is automatically shown in a Profile window.

For more control over the profiling process you can use the ⎕PROFILE system function described here.

Syntax

⎕PROFILE is monadic. The right argument is usually a nested vector, the first element of which is a keyword (such as 'on' or 'data'), and the remaining elements of which are parameters specific to the operation being carried out. (For certain operations, which take no parameters, a simple character vector argument of just the keyword can be supplied.) Keywords are case-insensitive.

Turning profiling on

Syntax:
`⎕PROFILE 'on' [method]`

To turn on profiling you use the 'on' keyword. This causes any previous profiling data to be discarded, and a new profiling session is started. The optional 'method' parameter specifies which of the following profiling types to use:

Code	***Profiling method***
1	Measure time in CPU cycles used by APLX application
2	Measure time in CPU cycles used by APL task
3	Measure time used by APLX application
4	Measure time used by APL task
5	Measure elapsed time
0	Use the first method supported by the OS (default)

Depending on which platform you are using, one or more of the timing methods may not be available. For example, earlier versions of Windows cannot measure the number of CPU cycles used by an application. If the method specified is not available a DOMAIN ERROR occurs.

Measuring CPU cycles (method 1 or 2) usually gives the most accurate results, because the CPU count is updated continuously. If this method is not available you can fall back on methods 3 and 4, which make use of a low-level timer provided by the operating system. This may be less accurate: under Windows the timer value is only updated each time a thread reaches the end of its time slice, so that a number of APL lines may execute for each tick of the timer.

In most implementations, APLX uses multiple process threads. There is typically one thread for each APL session in progress, one for each additional APL child task started under program control, and one shared thread to handle user interaction via the GUI. Depending on how your application is structured you might choose the following:

- For most applications it is best to measure the time taken by the whole APLX application (method 1 or 3). This will provide a more accurate reflection of the cost of executing each line of APL code because it includes any time used by the GUI thread - for example to handle any drawing operations that the line performs.
- For applications where you start additional tasks under APL control (or if you have multiple APL sessions executing simultaneously), choose method 2 or 4. This avoids wrongly charging the time taken in the other APL tasks to the current profile.
- Measuring the elapsed time can also return useful information; for example it can help you to find where time is spent by APL waiting for network operations to complete or executing `⎕DL`.

Controlling Profiling

Syntax:
```
⎕PROFILE 'pause'
⎕PROFILE 'resume'
⎕PROFILE 'reset'
r←⎕PROFILE 'state'
```

There is a small performance penalty when running APL code with profiling turned on, so you may wish to suspend profiling temporarily. You can do this using the 'pause' and 'resume' keywords.

To end profiling completely and discard all profiling data, use the 'reset' keyword. Profiling is also ended by)CLEAR or by loading a new workspace.

To determine the current profiling state use the 'state' keyword. This returns a five-element numeric vector as follows:

- [1] State: 0 if profiling off, 1 if on, 2 if paused, 3 if aborted because of e.g. insufficient memory
- [2] Method: Profiling method currently being used (See 'new' keyword)
- [3] Tick Period: For time-based profiling methods, this contains the period of the timer tick in nanoseconds (0 if unknown)
- [4] Resolution: The approximate resolution of the timer in ticks, or 0 if not known.
- [5] Total: The total time covered by the profiling data, in timer ticks

The Tick Period and Resolution values may only be approximate, depending on the capabilities of the underlying operating system. For example calls to measure thread times under Windows use the QueryThreadCycleTime method. This returns results in multiples of 100 nanoseconds (the tick period), but Windows only increments the thread time at the end of each time slice so the resolution is poor. You should use measurements in CPU cycles for greater accuracy if your version of Windows supports this

Viewing the profile data

Syntax:
```
⎕PROFILE 'show'
⎕PROFILE 'save' filename [detail]
r←⎕PROFILE 'data' [functions]
```

Profiling results can be viewed at any time while profiling is in progress or is paused. If you wish to perform cumulative profiling over several runs you can do so, because time spent in desk calculator mode is not recorded. Previous results are only discarded if you start a new profiling session, clear the workspace or load a new one, or if you explicitly discard them using the 'reset' keyword.

The easiest way to view the results is to use the 'show' keyword, which will cause a new Performance Profile window to be displayed. You can use this to explore the data in a number of ways, for example to find out where most time was spent or which functions were called most often.

To save the results as a file in HTML format, use the 'save' keyword. This takes a character vector containing the name of the file to create, which can be a full pathname or just a file name in the current directory. If you supply an empty vector, a dialog is displayed allowing the user to select a file.

Because the profiling information can be quite large, a second parameter to 'save' allows you to control the level of detail written to the HTML file. The values are:

- 0 - Write summary information only (default). This includes only the functions and lines which contibute most to the time taken
- 1 - Write detailed information which includes every function which executed during profiling

To obtain the profiling data as an APL array you can use the 'data' keyword. This returns a multi-row, 8 column nested array of profiling data, ordered by function and line number. The columns are as follows:

- [;1] Function name
- [;2] Line number within function
- [;3] Number of times line was executed
- [;4] Total time spent in the line itself
- [;5] Total time spent in the line and any functions it calls (its children).
- [;6] Average time taken to execute the line, excluding children
- [;7] Maximum time taken to execute the line, excluding children
- [;8] Minimum time taken to execute the line, excluding children

In the case of recursive functions, the time spent back in a function line is included in the 'self' figure, not in the figure for the line and its children.

You can restrict the data to one or more specified functions by supplying the function name(s), for example: `⎕PROFILE 'data' 'DRAW' 'UPDATE'`

⎕PW Print Width

The system variable `⎕PW` contains the numeric value of the maximum width of line output. Default is usually 80 (depending on the implementation). `⎕PW` can be altered by assigning a value between 40 and 390.

⎕R Carriage Return

The niladic system function ⎕R returns a Carriage Return character (with Line Feed), also known as a New Line character. On output, it has the effect of moving the cursor to the start of the next line, so it is commonly used as a line separator in text vectors.

```
      ⎕A,⎕R,⎕D
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789

      ⎕AF ⎕R
13
```

⎕R is the same as ⎕TCNL or ⎕TC[⎕IO+1]. See ⎕TC and ⎕TCxx.

⎕RECLASS Change class of objects

The dyadic system function ⎕RECLASS allows you to change the class of one or more object instances, provided these are user-defined APL objects.

The left argument should be either a single class reference, or a class name as a character vector. The right argument should be an array of object instances to be converted to the class specified in the left argument.

If the instances being reclassified contain non-default property values, these will be deleted unless they are also valid in the new class.

The main occasion when ⎕RECLASS is useful is if you have modified the class hierarchy. For example, you might have a class `Car` which inherits from `Vehicle`. You now decide you would like to create new classes `SportsCar` and `FamilyCar`, and change the class of your existing `Car` objects. To do this, you can use ⎕INSTANCES to find all instances of `Car` in the workspace, and change their class according to some criterion which you set:

```
      AllCars ← 1 ⎕INSTANCES Car
      AllCars.TopSpeed
130 135 143 120 91 141 117 123 98 84 135 155 111 122
      SportsCar ⎕RECLASS ((AllCars.TopSpeed) > 140)/AllCars
      FamilyCar ⎕RECLASS ((AllCars.TopSpeed) ≤ 140)/AllCars
      AllCars
[FamilyCar] [FamilyCar] [SportsCar] [FamilyCar] [FamilyCar] [SportsCar]
      [FamilyCar] [FamilyCar] [FamilyCar] [FamilyCar] [FamilyCar] [SportsCar]
      [FamilyCar] [FamilyCar]
```

⎕REPARENT Change parent of user-defined class

The dyadic system function ⎕REPARENT allows you to change the parent of an internal (user-defined) class.

The left argument is a reference to the class which you want to re-parent (or a character vector containing the name of the class). The right argument is a reference to the class which will be the new parent (or a character vector containing the new parent's name).

Other than the restriction that the new parent class (the right argument) must not be descended from the class you are modifying (the left argument), there is nothing to prevent you from re-parenting a class arbitrarily. However, the main use for ⎕REPARENT is for inserting an extra level into the class hierarchy. For example, if you have a class `Car` which inherits from `Vehicle`, you might want to create a new class `MotorVehicle` which inherits from `Vehicle`, and re-parent `Car` so that it now inherits from `MotorVehicle`:

```
      )CLASSES
Car     MotorVehicle    Vehicle
      ⎕CLASS Car
{Car} {Vehicle}
      ⎕CLASS MotorVehicle
{MotorVehicle} {Vehicle}
      Car ⎕REPARENT MotorVehicle
      ⎕CLASS Car
{Car} {MotorVehicle} {Vehicle}
```

The ⎕REPARENT line above could alternatively be written as:

```
      'Car' ⎕REPARENT 'MotorVehicle'
```

Any existing instances of the class will be unaffected except that any properties which are not valid in the new version of the class (because they were inherited from the old parent but are not inherited from the new parent) will be lost.

You can also re-parent a class using the `)REPARENT` system command, or by using the Class Editor (select Reparent Class.. from the File menu).

⎕RL Random Link

The system variable ⎕RL contains a random integer used by the random number generator. It changes after each use of the random-generator primitive ?. It can be set by assignment to ensure generation of identical numbers on different occasions.

See also the entries for ?, Roll and Deal.

⎕SETUP Set up external environment

The dyadic system function ⎕SETUP is used to set or query various parameters for the interface with external architectures such as .Net, Ruby and Java. Normally, you will want to set these parameters before using ⎕NEW or ⎕CALL to access the external system.

The left argument is a character vector which specifies the external environment, in the same format as for ⎕NEW. The right argument is either a character vector (containing a keyword specifying the parameter), or a nested vector where the first element is the keyword, and the remaining elements are parameters for that keyword. Keywords are case-insensitive. The options available depend on the environment, as follows:

.Net

Changing settings

Keyword: **using**

Sets the current search path for .Net namespaces and DLLs. These should be supplied as character vectors after the keyword:

```
      '.net' ⎕SETUP 'using' 'System' 'System.Text'
'QMath.Geom,c:\dev\qmath.dll'
```

Each element comprises either just a namespace (such 'System.Text'), or a namespace followed by the name of the DLL in which it is located. This can be a full path name, or just the name of the DLL (in which case the DLL should be in the Global Assembly Cache).

Keyword: **byref**

Sets or clears 'by reference' mode. The parameter should be a boolean scalar. The previous value is returned as the explicit result.

In 'by reference' mode, the list of arguments to each .Net method call is saved before the call. After the call has been made, it can be retrieved by querying the 'Args' parameter (see below). This is useful for the (relatively uncomon) cases where a .Net method takes an argument by reference, and modifies one or more of the arguments in-situ. (In C#, such parameters are identified by the keywords `out` or `ref`. In Visual Basic, they are identified by the keyword `ByRef`).

In this example, we call the HexUnescape method of the System.Uri class. This takes two parameters, a string (which might contain escaped character sequences such as '%20' corresponding to a space character), and an index position into the string. The index position is passed by reference. When the method completes, the index is incremented to point at the next character. By setting 'by reference' mode and reading back the arguments after the call, the new value of the by-reference argument is available to the APL application:

```
      uri←'.net' ⎕GETCLASS 'System.Uri'
      '.net' ⎕SETUP 'byref' 1
0
      ⎕AF  uri.HexUnescape 'The%20best%20way' 3
32
      '.net' ⎕SETUP 'args'
 The%20best%20way 6
      '.net' ⎕SETUP 'byref' 0
1
```

Notice how the index position passed in, which was 3 to point at the first '%20', has been incremented to 6 after the call, pointing now at the 'b' character.

Because there is an overhead in saving the argument list in this way, you should switch off 'by reference' mode once you have made the call and retrieved the argument list.

Querying values

Keyword: **version**

Returns the version of the Common Language Runtime as a character vector:

```
      '.net' ⎕SETUP 'version'
2.0.50727.312
```

Keyword: **using**

Returns the current search path for class names and assemblies, as a nested vector of character vectors:

```
      '.net' ⎕SETUP 'using'
System System.Text System.Drawing,system.drawing.dll
```

Each element comprises either just a namespace (such 'System.Text'), or a namespace followed by the name of the DLL in which it is located. This can be a full path name, or just the name of the DLL (in which case the DLL should be in the Global Assembly Cache).

Keyword: **args**

Returns the argument list after a call-by-reference (see description of the ByRef parameter above). If the is no argument list available, or the ByRef parameter is 0, it returns a Null object.

Keyword: **byref**

Returns the current value of the 'by reference' setting as boolean scalar.

Java

Java code runs inside a Java Virtual Machine (JVM). You can specify a number of settings to be used when the JVM is created. Creation will occur the first time you make a Java call other than setting one of the JVM creation parameters.

Changing settings

Keyword: **vm**

Specifies the full path and file name for the Java Virtual Machine (normally this will be set automatically from the registry or Java installation). The name should be supplied as a character vector after the keyword:

```
      'java' ⎕SETUP 'vm' 'c:\Program
Files\Java\jre1.6.0\bin\client\jvm.dll'
```

The main purpose for this is to specify a particular version of Java when running Java code. You must set this parameter before making any Java call.

Keyword: **classpath**

Sets the path for the Java classes (normally this will be set automatically from the registry or Java installation). The path should be supplied as a character vector after the keyword. You must set this parameter before making any Java call.

```
      'java' ⎕SETUP 'classpath' 'c:\myjava\'
```

Keyword: **vmoptions**

Sets one or more command-line options for the Java Virtual machine. Each option is specified after the keyword as a character vector, in the form `'Option=Value'`. You must set the options before making any Java call, since they are used as parameters when creating the Java Virtual Machine. Valid options vary depending on the JVM you are using.

Querying values

Keyword: **vm**

Queries the full path and file name for the Java Virtual Machine:

```
      'java' ⎕SETUP 'vm'
c:\Program Files\Java\jre1.6.0\bin\client\jvm.dll
```

Keyword: **classpath**

Queries the Java class path:

```
      'java' ⎕SETUP 'classpath'
.
```

Keyword: **vmoptions**

Queries the options passed to the Java Virtual Machine:

```
      'java' ⎕SETUP 'vmoptions'
```

Keyword: **version**

Returns the Java Virtual Machine version as a character vector:

```
      'java' ⎕SETUP 'version'
1.6
```

Note: This will cause the JVM to be created if this has not already happened.

Ruby

Changing settings

Keyword: **require**

Adds a Ruby script to the list of scripts in which Ruby will search for class definitions. The parameter is a character vector containing the script name. This can be specified either as a full path name, or as just a file name in which case Ruby will search in its current search path for the script:

```
      'ruby' ⎕SETUP 'require' 'Date'
      'ruby' ⎕SETUP 'require' 'c:\ruby\myapp.rb'
```

Keyword: **addpath**

Adds one or more directories to Ruby's current search path for scripts:

```
      'ruby' ⎕SETUP 'addpath' 'c:\rubyapps' 'c:\rubylibs\version2'
```

Keyword: **safelevel**

Sets the 'safe level' (security level) for Ruby execution. The parameter should be an integer in the range 0 to 4.

```
      'ruby' ⎕SETUP 'safelevel' 3
      'ruby' ⎕EVAL 's=String.new "Unsafe String"
Unsafe String
      'ruby' ⎕EVAL 's.taint'
      'ruby' ⎕EVAL 's.untaint'
#<SecurityError: (eval):0:in `untaint': Insecure operation `untaint'
at level 3>
DOMAIN ERROR
      'ruby' ⎕EVAL 's.untaint'
      ^
```

Keyword: **script**

Sets the nominal script name for Ruby execution (the default is 'embedded')

```
      'ruby' ⎕SETUP 'script' 'APLXBridge'
      'ruby' ⎕EVAL '$0'
APLXBridge
```

Querying values

Keyword: **version**

Returns the version of Ruby as a character vector:

```
      'ruby' ⎕SETUP 'version'
1.8.6
```

Keyword: **path**

Returns the current Ruby search path as a nested vector of character vectors:

```
      'ruby' ⎕SETUP 'path'
 c:/ruby/lib/ruby/site_ruby/1.8 c:/ruby/lib/ruby/site_ruby/1.8/i386-
msvcrt c:/ru
      by/lib/ruby/site_ruby c:/ruby/lib/ruby/1.8
c:/ruby/lib/ruby/1.8/i386-mswin
      32 .
```

Keyword: **script**

Returns the current nominal script name as a character vector:

```
      'ruby' ⎕SETUP 'script'
embedded
```

Keyword: **safelevel**

Returns the current Ruby 'safe level' as an integer scalar

```
      'ruby' ⎕SETUP 'safelevel'
0
```

⎕SI State Indicator

The niladic system function ⎕SI returns a character vector of functions that are currently halted. (See also)SI) This vector contains the same information as is produced by)SI, with embedded carriage returns to make it look like a matrix.

⎕SQL Interface to External Database

The system function ⎕SQL provides an extensible interface to a standard relational database such as Oracle, SQL Server, IBM DB2, MySQL, or PostgreSQL, using Structured Query Language (SQL). It can also be used (via ODBC) to access data from spreadsheets, Microsoft Access, DB2 files, and many other data sources. You can use it to retrieve data from relational databases as nested arrays in the workspace, or to update a database directly from within APL.

To use ⎕SQL, you need to be familiar with database concepts and with the SQL language. Depending on the database you are using, you may also need to know something about setting up ODBC. There are many standard textbooks and websites where you can learn more about these topics. In addition, most database systems are supplied with a sample set of tables which you can use to work through examples of SQL syntax.

In most cases, ⎕SQL interfaces to the database by calling ODBC (Open DataBase Connectivity). Under Windows, this should normally be installed as part of the system software, but you will probably need to configure the data sources which it interfaces to (see the Windows Help system, or the Microsoft web site http://www.microsoft.com for further information). Under Linux and AIX, APLX uses unixODBC; this is included in many Linux distributions, or can be downloaded from http://www.unixodbc.org. Under MacOS X, APLX interfaces to the open-source iODBC software, which is distributed as part of the MacOS X system software (see the iODBC web site http://www.iodbc.org). ⎕SQL is not supported under Mac OS 9.

To make full use of ⎕SQL, you may also need a fully-featured relational database. Nearly all database systems can be accessed via ODBC, so ⎕SQL can be used to interface to all the major commercial database systems such as Oracle, Sybase and SQL Server. There are also several open-source databases available free of charge; these include MySQL and PostgreSQL (either or both of these are included in many Linux distributions). IBM's DB2 Personal Edition is also available free of charge (subject to IBM's license terms), and provides a very high-specification database system for single-machine use.

Note: Because APLX ships on multiple platforms, and ⎕SQL can be used to interface to hundreds of different databases and other data sources, the examples shown below will probably behave differently or need to be adapted for your system and database manager.

Syntax

In most cases, ⎕SQL will be used monadically. The right argument is usually a nested vector, the first element of which is a keyword (such as `'Connect'` or `'Execute'`), and the remaining elements of which are parameters specific to the operation being carried out. (For certain operations, which take no parameters, a simple character vector argument of just the keyword can be supplied.) Keywords are case-insensitive.

If you are connecting simultaneously to more than one database, then you also need to supply a left argument to ⎕SQL. This is a numeric integer which is used as a tie number to indicate which database you wish to access. If you omit the left argument, a default tie number of 0 is assumed.

Result

In general, the result of ⎕SQL is a length-three nested vector, comprising:

- An integer vector giving information about whether the operation was successful. This consists of four integers.

  The first element is a code indicating the source of any error. A 0 indicates that no error occurred. A non-zero first element indicates that an error was reported, with 1 meaning that APLX itself reported an error, 2 indicating that the interface layer between APLX and the Database Driver reported an error, and 3 indicating that the driver or underlying database reported an error.

  The second element is the numeric error code, if any, which was returned by the software which reported the error. For example, for an ODBC interface it is the ODBC error number.

  The third element is a 'warning' code. 0 indicates no warning.

  The fourth element indicates whether more data might be available, for cases where you are retrieving data in chunks. 0 means that no more data is available. 1 means that more data might be available.

- The second item of the nested vector result is a character string, containing any error message as text. It is an empty vector if there is no error message.

- The third item of the nested vector result is the data (if any) returned by the SQL operation. If no data is returned, it is an empty vector.

It is recommended that you always use multiple assignment to split the result from ⎕SQL into these three constituent parts, as in this example:

```
      (RC ERRMSG DATA)←⎕SQL 'Connect' 'aplxdb2'
      RC
3 ¯1024 0 0
      ERRMSG
[IBM][CLI Driver] SQL1024N  A database connection does not exist.  SQLSTATE=08003
```

In this example, an error has been reported by IBM's DB2 driver software.

This method makes it very easy to check for errors by seeing if the first element of `RC` is non-zero.

Connecting to a database

Syntax:
```
[Tie] ⎕SQL 'Connect' Interface ConnectionString
[Tie] ⎕SQL 'Connect' Interface DataSourceName [User] [Password]
```

Before carrying out any database operation, you need to cause APLX to load the appropriate interface library, and connect to the underlying database. This is done using the ⎕SQL 'Connect' operation. As shown above, there are two variants of the syntax, depending on how you want to specify the database connection.

The first parameter is the name of the interface driver. This is a shared library which interfaces between APLX and the database manager. It can be either a simple shared-library name (such as `'aplxodbc'`), or a full path name (such as `'c:\apl\drivers\aplxodbc.dll'`). In the former case, which is normally preferred because it is operating-system independent, APLX will load the driver from the `bin` directory of the APLX installation; the shared-library extension (`.dll` for Windows, `.so` for Linux and AIX, or `.bundle` for Mac OS X) will be added automatically. In the latter case, the file name will be used as supplied, and you need to provide the file extension. If the library cannot be loaded, either because the APLX interface driver is not found, or because the underlying database manager library is not properly installed, APLX will report the error `LOGICAL UNIT NOT FOUND`.

Normally, you will use one of the drivers supplied as standard with APLX. These are currently:

For 32-bit versions of APLX:

- `aplxodbc` - Interface to Microsoft ODBC (under Windows), or UnixODBC (under Linux x86 or AIX), or iODBC (under MacOS X).

- `aplxdb2` - Direct interface to IBM's DB2 database.

For APLX64:

- `aplx64odbc` - Interface to Microsoft ODBC (under Windows XP 64), or UnixODBC (under Linux x86_64).

The second and subsequent parameters depend on the exact database interface you are using, and on how your system is set up. If you are using ODBC, you can specify either a 'Connection String' or a 'Data Source Name' (DSN). Other database interfaces use a Database Alias, which is similar to a DSN.

Specifying the database with a Connection String:

A Connection String is a text vector containing a series of semicolon-delimited phrases, each of which is of the form `'keyword=value;'` . These phrases fully or partially define the connection to the database. The parameters typically include the driver name, user name, connection parameters and so on, plus any driver-specific information such as the name of an Excel spreadsheet, the name of the database alias, or the network address of the database server. (*Technical note:* This method of connection uses the underlying `SQLDriverConnect` ODBC call.)

Under Windows, if the Connection String is incomplete (i.e. if it does not fully define all the parameters needed to make the connection), the ODBC sub-system will display a dialog which allows you to set up the remaining parameters of the connection interactively. You can even specify an empty vector, in which case you can choose the ODBC driver, database, and all other parameters which are required. For MacOS, Linux, and AIX you need to specify a full connection string.

On successful connection, `⎕SQL` returns as the third element of the result the complete connection string; you can store this, and use it for subsequent connections (without any dialog displaying).

For example, suppose you want to use `⎕SQL` under Windows to access a table saved in an Excel spreadsheet. You can specify an empty connection string:

```
    (RC ERRMSG CNS) ← ⎕SQL 'Connect' 'aplxodbc' ''
```

A dialog should appear. Select the 'File DSN' tab, and press the New... button. Select the driver 'Microsoft Excel Driver', and save the DSN in some suitable scratch directory. When you click Next.. , the ODBC Excel driver should ask you for the name of the spreadsheet containing the data. (You can use the sample spreadsheet 'alcohol.xls' which is supplied with APLX in the 'ws' directory of the APLX installation, usually `C:\Program Files\MicroAPL\APLX\ws`). This should establish a connection to the Excel spreadsheet as though it were a database, and the `CNS` variable defined above should contain the full connection string which you can use to repeat the connection without the dialog:

```
      CNS
DBQ=C:\Program Files\MicroAPL\APLX\ws\alcohol.xls;DefaultDir=C:\Program
Files\MicroAPL\
      APLX\;Driver={Microsoft Excel Driver (*.xls)};DriverId=790;FIL=excel 8.0;FI
      LEDSN=C:\Program Files\Common Files\ODBC\Data Sources\sample.dsn;MaxBufferSi
      ze=2048;MaxScanRows=8;PageTimeout=5;ReadOnly=1;SafeTransactions=0;Threads=
      3;UID=admin;UserCommitSync=Yes;
```

You should now be able to access the data using SQL commands, as described below:

```
      (RC ERRMSG DATA) ← ⎕SQL 'Do' 'select * from alcohol'
      RC
0 0 0 0
```

```
      ⍴DATA
30 18
      3 8↑DATA
 Australia  9.4 11.6 12.9 11.7 10.5   10  9.7
 Austria   10.9 13.9 13.8 12.1 12.6 12.8 12.1
 Belgium    8.9 12.3   14 12.9 12.1 11.5 11.7
```

Specifying the database with a Data Source Name:

If the third parameter to 'Connect' is a text vector which does not end in a semi-colon and is not empty, APLX will assume that it represents a Data Source Name (DSN). A DSN is effectively a pre-defined Connection String, which encapsulates all the details needed to connect to the database. It can be stored in a text file as a "File DSN", or in the registry (or UnixODBC configuration file) as a "User DSN" or "System DSN". It is identified by a simple name. In some cases, the User Name and Password have to be entered as extra parameters. (*Technical note:* This method of connection uses the underlying `SQLConnect` ODBC call.)

Under Windows, you set up DSNs using Microsoft's 'ODBC Data Source Administrator' dialog. Under Linux or MacOS, you set them up using UnixODBC or iODBC configuration files. See the appropriate ODBC documentation for details, which will depend on the database you are using and where the data is held.

Example using ODBC to access data from an Oracle Database. The Data Source Name has been configured in advance to be 'Oracle Sample', but you need to specify the user name and password:

```
      ⎕SQL 'Connect' 'aplxodbc' 'Oracle Sample' 'scott' 'tiger'
 0 0 0 0
```

Specifying the database with a Database Alias:

For DB2 and other database-specific (non-ODBC) drivers, the second parameter to Connect is the database alias, which is similar to a DSN. The third and fourth parameters are the User ID and Password, if these are required by the database.

Example using IBM's DB2, assuming you have a database alias called 'DB2' defined:

```
      ⎕SQL 'Connect' 'aplxdb2' 'DB2'
 0 0 0 0
```

Connecting to multiple databases

If you want to connect to more than one database simultaneously, you will need to specify an integer left argument to `⎕SQL` when you connect. You then use this tie number in all subsequent operations, to identify which database you want to access.

Checking which connections are open

Syntax: `⎕SQL 'Connections'`

This returns an integer vector of the `⎕SQL` tie numbers in use, or an empty vector if there are none. (Note that this is an exception to the general rule that `⎕SQL` returns a nested vector of Return Codes, Error Message, Data).

Disconnecting from a database

Syntax: `[Tie] ⎕SQL 'Disconnect'`

You must always disconnect from the database when you have finished using it, otherwise you may continue to tie up resources and in some cases may prevent other users from accessing the database.

Disconnecting automatically closes any database statements you have open. Depending on the underlying driver, it will usually also roll back any uncommitted transactions.

You can disconnect from all active databases using the statement:

```
      0 0ρ(⎕SQL 'Connections') ⎕SQL ¨⊂'Disconnect'
```

Executing simple database operations

Syntax: `[Tie] ⎕SQL 'Do' Statement`

The `'Do'` keyword allows you to execute almost any arbitrary SQL operation immediately. It takes a single parameter, which is a text vector containing the SQL statement you want to perform. For example, this can be a SELECT or UPDATE, or a more specialized operation such as GRANT. (However, it should not be a COMMIT or ROLLBACK, since these are done using the specific syntax described below). Any data returned by the database will be returned in the third element of the result from `⎕SQL`. To avoid getting a `WS FULL` error, make sure you do not retrieve large data sets using this method.

For example:

```
      ⎕DISPLAY ⎕SQL 'DO' 'SELECT * FROM CL_SCHED'
┌→────────────────────────────────────────────────────┐
│ ┌→──────┐ ┌⊖┐ ┌→────────────────────────────────────┐ │
│ │0 0 0 0│ │ │ ↓ ┌→─────┐   ┌→───────┐ ┌→───────┐ │ │
│ └~──────┘ └─┘ │ │CL3RPN│ 3 │09:00:00│ │10:30:00│ │ │
│               │ └──────┘   └────────┘ └────────┘ │ │
│               │ ┌→─────┐   ┌→───────┐ ┌→───────┐ │ │
│               │ │CL4SDM│ 3 │11:00:00│ │12:30:00│ │ │
│               │ └──────┘   └────────┘ └────────┘ │ │
│               │ ┌→─────┐   ┌→───────┐ ┌→───────┐ │ │
│               │ │CL4DSE│ 2 │09:00:00│ │10:30:00│ │ │
│               │ └──────┘   └────────┘ └────────┘ │ │
│               └∊──────────────────────────────────┘ │
└∊────────────────────────────────────────────────────┘
```

In this example, `⎕SQL` has returned a three-element nested vector as normal. The first element is the vector of error information, with all zeroes indicating that no error occurred and that no more data is available. The second element is the error message, which is empty because no error occurred.The third element is the result from the SQL query. In this case, a nested matrix of three rows of data has been returned, with one column per table column.

Using the `'Do'` keyword has the advantage of being very simple. However, it has two disadvantages which mean that it will often be better to use the multi-stage `'Prepare'` and `'Execute'` operations described below. These are, firstly, that you have no control over the amount of data which is returned to you, and secondly that you cannot 'bind' APL data to your SQL operation at run-time - any parameters need to be hard-coded as text in the SQL statement which you execute. For these reasons, we recommend that you do not use the `'Do'` keyword in applications for executing SQL `'Select'` statements, unless you are certain that the data set will always be small and that there is therefore no danger of a WS FULL error being generated at some time in the future when the database may have grown.

Dynamic SQL statements

As an alternative to executing a statement directly using the `'Do'` keyword, you can prepare a statement in advance, optionally bind parameters to it, and then execute it, retrieving any results in chunks rather than all in one go. To do this, you carry out the following steps:

- Prepare a named SQL statement using the `'Prepare'` keyword. This statement will usually be a SELECT, UPDATE or INSERT. You can have several such statements open simultaneously. The statement can optionally include place-holders for parameters which are supplied later from APL variables.
- Execute the named SQL statement using the `'Execute'` keyword. If the SQL statement which you have prepared requires parameters, these are supplied at this stage.
- If the statement which you have executed returns a result set, you can now loop round using the `'Fetch'` keyword, retrieving the results in stages (you specify the maximum number of rows to retrieve each time).
- Finally, you use the `'Close'` keyword to close the statement and release resources associated with it. Alternatively, you can re-execute it with a different set of parameters.

Preparing the dynamic SQL statement

Syntax: `[Tie] ⎕SQL 'Prepare' Name Statement`

This operation creates a new SQL statement. The parameter `Name` is a character vector which is the internal name by which you will refer to the statement subsequently (names are case-sensitive).

The parameter `Statement` is the SQL statement you want to execute. This can optionally include ? characters; these act as placeholders for APL data which will be substituted when the statement is executed. For example:

```
      ⎕SQL 'Prepare' 'cs1' "UPDATE CL_SCHED SET DAY=? WHERE CLASS_CODE=?"
 0 0 0 0
```

In this example, 'cs1' is the name of the statement, which will be used to identify it subsequently. The SQL statement is an UPDATE, with two parameters which will be filled in when it is executed.

Executing the prepared statement

Syntax: `[Tie] ⎕SQL 'Execute' Name [Param1] [Param2]...`

This causes the prepared statement to be submitted to the database for execution. `Name` is the name of the statement (i.e. the name you supplied when it was prepared). If there were any ? placeholders in the statement, you need to provide one element for each such placeholder; these will be substituted at this stage. For example:

```
      ⎕SQL 'Execute' 'cs1' 2 'CL4DSE'
 0 0 0 0   SQL_CURSH200C4
```

In this example, the numeric data 2 is substituted for the first ? in the SQL statement, the character vector `'CL4DSE'` is substituted for the second, and the statement is executed.

The result of the operation follows the standard pattern. If a data set is available (i.e. the statement is something like a SELECT), a 1 will be placed in the fourth element of the return-code part of the result - you can retrieve the data using the `'Fetch'` keyword.

When you use the `'Execute'` keyword, the third element of the result of `⎕SQL` is the name of the SQL cursor (if any) allocated by the underlying database. This can be used for advanced update operations using the SQL phrase 'WHERE CURRENT OF'.

Fetching the result set

Syntax: `[Tie] ⎕SQL 'Fetch' Name [Count]`

If the statement you have executed returns a result set, you can fetch the data in chunks, using the `'Fetch'` keyword. `Name` is the name of the statement you have just executed. The optional `Count` parameter is the maximum number of rows to return (if you omit this, APLX will try to return all the rows). The result follows the standard pattern, with a 1 in the fourth element of the return code vector indicating that more data might be available, so you should call `Fetch` again. The data itself is returned as a matrix (usually nested), as the third element of the result.

For example, suppose we have a table with just three rows. We can retrieve these immediately using the 'Do' keyword:

```
      ⎕SQL 'DO' 'SELECT * FROM CL_SCHED'
 0 0 0 0    CL3RPN  3 09:00:00 10:30:00
            CL4SDM  3 11:00:00 12:30:00
            CL4DSE  2 09:00:00 10:30:00
```

Alternatively, we can use dynamic SQL with parameter substitution as follows, in this case retrieving one row at a time:

```
      ⎕SQL 'Prepare' 'cs2' "SELECT * FROM CL_SCHED where Starting=?"
 0 0 0 0
      ⎕SQL 'Execute' 'cs2' '9:00'
 0 0 0 1   SQL_CURSH200C4
```

```
      ⎕SQL 'Fetch' 'cs2' 1
 0 0 0 1    CL3RPN  3 09:00:00 10:30:00
      ⎕SQL 'Fetch' 'cs2' 1
 0 0 0 1    CL4DSE  2 09:00:00 10:30:00
      ⎕SQL 'Fetch' 'cs2' 1
 0 0 0 0
      ⎕SQL 'Close' 'cs2'
 0 0 0 0
```

In this example, we have selected only those rows where the column 'Starting' (a time) is 9:00. The first time we call `'Fetch'`, we get the first row only because we have set the maximum number of rows to be returned to be 1. The return code indicates that more data might be available, so we call `'Fetch'` again to retrieve the next row. Finally, the call to `'Fetch'` returns an empty matrix as the third element (because we have now successfully read all the data), and the 'more' indicator is 0, so there is no more data to be read. We `'Close'` the statement.

In a real example, using a large database, there might be many thousands of rows returned. For efficiency, you would probably retrieve these in blocks of several hundred rows (depending on the workspace available and your processing requirements).

Note that the 'more' indicator of 1 means only that more data *might* be available; when you come to fetch it, the database may report that you are in fact at the end of the data set, and return no more rows. Once this happens, the 'more' indicator will revert to 0.

Describing the result set

Syntax: `[Tie] ⎕SQL 'Describe' Name`

After you have prepared or executed a statement, you can use the `'Describe'` keyword to determine the types of the columns (if any) which will be returned. (Doing this before the staement is executed is valid, but may be inefficient, depending on the underlying database). The data description is returned as a four-row matrix as the third element of the result of `⎕SQL`, with one column per column of the result set. The four rows are Column Name, Column Label, SQL Data type string (eg `'DECIMAL 7.2'`), and a binary flag indicating whether NULL is valid for the column. For example:

```
      ⎕SQL 'Prepare' 'cs1' 'SELECT * FROM CL_SCHED'
 0 0 0 0
      ⎕SQL 'Execute' 'cs1'
 0 0 0 1   SQL_CURSH200C4
      ⎕SQL 'Describe' 'cs1'
 0 0 0 0    CLASS_CODE      DAY STARTING ENDING
            CLASS_CODE      DAY STARTING ENDING
                CHAR 7 SMALLINT     TIME   TIME
                     0        1        1      1
```

In this example, the first column of the data set is called CLASS_CODE, and its label is also CLASS_CODE. It is of type CHAR, with a length of 7 characters. It cannot be NULL. The second column is called DAY, and is a small integer, which can be NULL. The third and fourth columns are called STARTING and ENDING, are of type TIME, and can be NULL.

Closing the prepared statement

Syntax: `[Tie] ⎕SQL 'Close' Name`

This closes the prepared statement and releases resources associated with it.

Transactions and Isolation Levels

One of the key features of most modern relational database systems is the ability to provide a consistent view of the data even when multiple users are changing it simultaneously, and to roll-back any changes if an error occurs within a sequence of updates. `⎕SQL` provides access to these facilities (provide they are supported by the underlying database) as follows:

Syntax: `[Tie] ⎕SQL 'Autocommit' [Value]`

This feature allows you to specify whether each SQL operation should be regarded as a separate transaction. If you set `Value` to 1 (which is the default), a COMMIT will implicitly be executed after every SQL operation you perform. This means that you will not be able to roll-back changes if an error occurs. If Value is set to 0, you will need to use the `'Commit'` or `'Rollback'` keywords explicitly when you make any changes to the database. If you omit `Value`, the current state of the parameter is returned as the third element of the result of `⎕SQL`. Note that only full database systems such as Oracle or DB2 support transactions; if you are using ODBC to access simple filing systems such as Excel tables or text files, `Autocommit` is always 1.

Caution: If you set `'Autocommit'` to 0, you must remember to Commit any changes before ending the session, otherwise they will be rolled back when you disconnect from the database.

Syntax: `[Tie] ⎕SQL 'Commit'`

Causes any pending changes to the database to be permanently committed. This has no effect if `'Autocommit'` is set to 1.

Syntax: `[Tie] ⎕SQL 'Rollback'`

Causes any pending changes to the database to be thrown away, and the database returned to the state it was in at the time of the most recent COMMIT. This has no effect if `'Autocommit'` is set to 1, since in this case all changes will automatically have already been committed.

Syntax: `[Tie] ⎕SQL 'Isolation' [Code]`

Allows you to set or query the method used to handle simultaneous changes to the database by multiple users (if this is supported by the underlying database). The parameter Code is a two-character string, which sets the isolation level as follows:

Code	*ODBC Name*	*DB2-style name*
UR	SQL_TXN_READ_UNCOMMITTED	Uncommitted Read
CS	SQL_TXN_READ_COMMITTED	Cursor Stability
RS	SQL_TXN_REPEATABLE_READ	Read Stability
RR	SQL_TXN_SERIALIZABLE	Repeatable Read

If you omit the Code parameter, the current setting is returned as the third element of the result.

Note that not all database systems and drivers support all these isolation levels, or any at all; if you try to use an unsupported case, an SQL error will be returned. For example, the Excel driver under ODBC does not support isolation levels and rollbacks:

```
      ⎕SQL 'Isolation'
 3 106 0 0  [Microsoft][ODBC Excel Driver]Optional feature not implemented
```

Determining available Data Source Names

Syntax: `[Tie] ⎕SQL 'Datasources' Interface`

This call allows you to determine which Data Sources are available. It takes a single parameter, which is the name of the APLX database interface (usually 'aplxodbc' for 32-bit versions of APLX, or 'aplx64odbc' for APLX64). The data returned as the third element of the result is a nested matrix of two columns. The first column is the Data Source Name (i.e. the name you should supply to `'Connect'`), the second the driver description. The results will depend on what User and System Data Sources you have set up on your system. In this example, we have five DSNs defined:

```
      ⎕DISPLAY ⎕SQL 'Datasources' 'aplxodbc'
┌→───────────────────────────────────────────────────────────────────────────┐
│ ┌→──────┐  ┌⊖┐  ┌→────────────────────────────────────────────────────────┐ │
│ │0 0 0 0│  │ │  ↓ ┌→──────────────────┐    ┌→──────────────────────────────┐ │ │
│ └~──────┘  └─┘  │ │MS Access Database│    │Microsoft Access Driver (*.mdb)│ │ │
│                 │ └──────────────────┘    └───────────────────────────────┘ │ │
│                 │ ┌→──────────┐           ┌→─────────────────────────────┐  │ │
│                 │ │dBASE Files│           │Microsoft dBase Driver (*.dbf)│  │ │
│                 │ └───────────┘           └──────────────────────────────┘  │ │
│                 │ ┌→───────────┐          ┌→─────────────────────────────┐  │ │
│                 │ │Excel Sample│          │Microsoft Excel Driver (*.xls)│  │ │
│                 │ └────────────┘          └──────────────────────────────┘  │ │
│                 │ ┌→───┐                  ┌→─────────┐                      │ │
│                 │ │MQIS│                  │SQL Server│                      │ │
│                 │ └────┘                  └──────────┘                      │ │
│                 │ ┌→──┐                   ┌→───────────────────┐            │ │
│                 │ │DB2│                   │IBM DB2 ODBC DRIVER│            │ │
│                 │ └───┘                   └────────────────────┘            │ │
│                 └∊──────────────────────────────────────────────────────────┘ │
└∊───────────────────────────────────────────────────────────────────────────┘
```

Note: Microsoft's ODBC driver will not allow you to use this call once you have connected to a data source.

Examining the catalog of tables

Syntax: `[Tie] ⎕SQL 'Tables' [Catalog] [Schema] [Table] [Types]`

Returns (as the third element of the result) a nested array giving details of the tables and pseudo-tables available in the database (including system tables and views). The details returned are contained in a five-column array, where each row represents a table in the database. The columns are:

- The name of the catalog
- The name of the schema
- The name of the table or pseudo-table
- The type of table or pseudo-table, one of: 'TABLE', 'VIEW', 'INOPERATIVE VIEW', 'SYSTEM TABLE', 'ALIAS', or 'SYNONYM'
- Descriptive information about the table or pseudo-table

The result from this call can be very large, so in most cases you should provide parameters to limit the scope of the enquiry. This is done by providing up to four character vectors as parameters. The first three are pattern-matching strings for the Catalog, Schema and Table name respectively. An underscore (_) character stands for any single character, a percent (%) character stands for any sequence of zero or more characters, and other characters stand for themselves. The case of a letter is significant. An empty vector matches against any string. The fourth parameter contains a list of table types you are interested in, separated by commas if there are more than one. Valid table type identifiers are: TABLE, VIEW, SYSTEM TABLE, ALIAS, SYNONYM. If this parameter is an empty vector, this is equivalent to specifying all of the possibilities for the table type identifier.

For example, this statement returns all user tables, but not Views or System tables:

```
      ⎕DISPLAY ⎕SQL 'TABLES' '' '' '' 'TABLE'
┌→─────────────────────────────────────────────────────────┐
│ ┌→──────┐ ┌⊖┐ ┌→──────────────────────────────────────┐ │
│ │0 0 0 0│ │ │ ↓ ┌⊖┐ ┌→──┐ ┌→───────┐    ┌→────┐ ┌⊖┐  │ │
│ └~──────┘ └─┘ │ │0│ │RPN│ │CL_SCHED│    │TABLE│ │0│  │ │
│               │ └~┘ └───┘ └────────┘    └─────┘ └~┘  │ │
│               │ ┌⊖┐ ┌→──┐ ┌→─────────┐  ┌→────┐ ┌⊖┐  │ │
│               │ │0│ │RPN│ │DEPARTMENT│  │TABLE│ │0│  │ │
│               │ └~┘ └───┘ └──────────┘  └─────┘ └~┘  │ │
│               │ ┌⊖┐ ┌→──┐ ┌→──────┐     ┌→────┐ ┌⊖┐  │ │
│               │ │0│ │RPN│ │EMP_ACT│     │TABLE│ │0│  │ │
│               │ └~┘ └───┘ └───────┘     └─────┘ └~┘  │ │
│               │ ┌⊖┐ ┌→──┐ ┌→────────┐   ┌→────┐ ┌⊖┐  │ │
│               │ │0│ │RPN│ │EMP_PHOTO│   │TABLE│ │0│  │ │
│               │ └~┘ └───┘ └─────────┘   └─────┘ └~┘  │ │
│               │ ┌⊖┐ ┌→──┐ ┌→─────────┐  ┌→────┐ ┌⊖┐  │ │
│               │ │0│ │RPN│ │EMP_RESUME│  │TABLE│ │0│  │ │
│               │ └~┘ └───┘ └──────────┘  └─────┘ └~┘  │ │
│               │ ┌⊖┐ ┌→──┐ ┌→───────┐    ┌→────┐ ┌⊖┐  │ │
│               │ │0│ │RPN│ │EMPLOYEE│    │TABLE│ │0│  │ │
│               │ └~┘ └───┘ └────────┘    └─────┘ └~┘  │ │
│               │ ┌⊖┐ ┌→──┐ ┌→──────┐     ┌→────┐ ┌⊖┐  │ │
│               │ │0│ │RPN│ │IN_TRAY│     │TABLE│ │0│  │ │
│               │ └~┘ └───┘ └───────┘    └─────┘ └~┘ │ │
│               │ ┌⊖┐ ┌→──┐ ┌→──┐        ┌→────┐ ┌⊖┐ │ │
│               │ │0│ │RPN│ │ORG│        │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └───┘        └─────┘ └~┘ │ │
│               │ ┌⊖┐ ┌→──┐ ┌→──────┐    ┌→────┐ ┌⊖┐ │ │
│               │ │0│ │RPN│ │PROJECT│    │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └───────┘    └─────┘ └~┘ │ │
│               │ ┌⊖┐ ┌→──┐ ┌→────┐      ┌→────┐ ┌⊖┐ │ │
│               │ │0│ │RPN│ │SALES│      │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └─────┘      └─────┘ └~┘ │ │
│               │ ┌⊖┐ ┌→──┐ ┌→────┐      ┌→────┐ ┌⊖┐ │ │
│               │ │0│ │RPN│ │STAFF│      │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └─────┘      └─────┘ └~┘ │ │
│               └∊────────────────────────────────────┘ │
└∊──────────────────────────────────────────────────────┘
```

The following variant limits the selection further, to tables whose name begins with 'EMP':

```
      ⎕DISPLAY ⎕SQL 'TABLES' '' '' 'EMP%' 'TABLE'
┌→──────────────────────────────────────────────────────┐
│ ┌→──────┐ ┌⊖┐ ┌→────────────────────────────────────┐ │
│ │0 0 0 0│ │ │ ↓ ┌⊖┐ ┌→──┐ ┌→──────┐    ┌→────┐ ┌⊖┐ │ │
│ └~──────┘ └─┘ │ │0│ │RPN│ │EMP_ACT│    │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └───────┘    └─────┘ └~┘ │ │
│               │ ┌⊖┐ ┌→──┐ ┌→────────┐  ┌→────┐ ┌⊖┐ │ │
│               │ │0│ │RPN│ │EMP_PHOTO│  │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └─────────┘  └─────┘ └~┘ │ │
│               │ ┌⊖┐ ┌→──┐ ┌→─────────┐ ┌→────┐ ┌⊖┐ │ │
│               │ │0│ │RPN│ │EMP_RESUME│ │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └──────────┘ └─────┘ └~┘ │ │
│               │ ┌⊖┐ ┌→──┐ ┌→───────┐   ┌→────┐ ┌⊖┐ │ │
│               │ │0│ │RPN│ │EMPLOYEE│   │TABLE│ │0│ │ │
│               │ └~┘ └───┘ └────────┘   └─────┘ └~┘ │ │
│               └∊────────────────────────────────────┘ │
└∊──────────────────────────────────────────────────────┘
```

Examining the catalog of columns

Syntax: `[Tie] ⎕SQL 'Columns' [Catalog] [Schema] [Table] [Column]`

This works in a similar way to the `'Tables'` keyword. Again, you should normally use parameters to limit the size of the returned result. All the parameters are pattern-matching strings, following the same rules as for `'Tables'`.

The result has 18 columns, as follows:

1. The name of the Catalog
2. The name of the Schema
3. Name of the table or pseudo-table
4. Name of the column
5. SQL data type code of the column, as an integer
6. Character string representing the name of the data type

7. If the column type is a character or binary string, the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is the total number of digits

8. The maximum number of bytes needed to store data from this column as text.

9. The scale of the column. An empty numeric vector is returned for data types where scale is not applicable.

10. The radix expressing the precision (either 10, 2 or NULL). If the data type is an approximate numeric data type, this column contains the value 2, and the column size is expressed in bits. If the data type is an exact numeric data type, this column contains the value 10, and the column size is expressed in decimal digits. An empty vector is returned for data types where radix is not applicable.

11. 0 if the column does not accept NULL values, 1 if the column accepts NULL values.

12. Descriptive information about the column, if available.

13. The column's default value. If the default value is a numeric literal, then this column contains the character representation of the numeric literal with no enclosing single quotes. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value a pseudo-literal, such as for DATE, TIME, and TIMESTAMP columns, then this column contains the keyword of the pseudo-literal (e.g. CURRENT DATE) with no enclosing quotes. If NULL was specified as the default value, then this column returns the word NULL, not enclosed in quotes. If the default value cannot be represented without truncation, then this column contains TRUNCATED with no enclosing single quotes. If no default value was specified, then this field is an empty vector.

14. SQL data type

15. The subtype code for datetime data types

16. The maximum length in bytes for a character data type column. For all other data types it is an empty vector.

17. The ordinal position of the column in the table. The first column in the table is number 1.

18. Contains the string 'NO' if the column is known to be not nullable; and 'YES' otherwise.

For example:

```
      ⎕DISPLAY 3⊃⎕SQL 'COLUMNS' '' '' 'CL%'
┌→──────────────────────────────────────────────────────────────────────────────────┐
↓ ┌⊖┐ ┌→──┐ ┌→───────┐ ┌→───────┐    ┌→───┐     ┌⊖┐ ┌⊖┐   ┌⊖┐ ┌⊖┐   ┌⊖┐       ┌→──┐ │
│ │0│ │RPN│ │CL_SCHED│ │CLASS_CD│ 1  │CHAR│ 7 7 │0│ │0│ 1 │0│ │0│ 1 │0│ 7   1 │NO │ │
│ └~┘ └───┘ └────────┘ └────────┘    └────┘     └~┘ └~┘   └~┘ └~┘   └~┘       └───┘ │
│ ┌⊖┐ ┌→──┐ ┌→───────┐ ┌→──┐         ┌→──┐                ┌⊖┐ ┌⊖┐   ┌⊖┐ ┌⊖┐   ┌→──┐ │
│ │0│ │RPN│ │CL_SCHED│ │DAY│      5  │INT│  5 2 0   10  1 │0│ │0│ 5 │0│ │0│ 2 │YES│ │
│ └~┘ └───┘ └────────┘ └───┘         └───┘                └~┘ └~┘   └~┘ └~┘   └───┘ │
│ ┌⊖┐ ┌→──┐ ┌→───────┐ ┌→───────┐    ┌→───┐         ┌⊖┐   ┌⊖┐ ┌⊖┐         ┌⊖┐   ┌→──┐ │
│ │0│ │RPN│ │CL_SCHED│ │STARTING│ 92 │TIME│ 8 6 0   │0│ 1 │0│ │0│ 9 2   │0│ 3 │YES│ │
│ └~┘ └───┘ └────────┘ └────────┘    └────┘         └~┘   └~┘ └~┘         └~┘   └───┘ │
│ ┌⊖┐ ┌→──┐ ┌→───────┐ ┌→─────┐      ┌→───┐         ┌⊖┐   ┌⊖┐ ┌⊖┐         ┌⊖┐   ┌→──┐ │
│ │0│ │RPN│ │CL_SCHED│ │ENDING│   92 │TIME│ 8 6 0   │0│ 1 │0│ │0│ 9 2   │0│ 4 │YES│ │
│ └~┘ └───┘ └────────┘ └──────┘      └────┘         └~┘   └~┘ └~┘         └~┘   └───┘ │
└∊──────────────────────────────────────────────────────────────────────────────────┘
```

Examining details of the connection

Syntax: `[Tie] ⎕SQL 'Describe'`

This uses the same keyword `'Describe'` which is used to provide details of a result set (see above). However, if no cursor name is supplied as a parameter, APLX returns details of the connection as a nested vector of 9 elements, as follows:

1. Data source name
2. Driver ODBC version
3. DBMS name
4. DBMS version
5. Driver name
6. Driver version
7. Database name
8. User name
9. SQL conformance, expressed as a string "Minimum SQL Grammar", "Core SQL Grammar", or "Extended SQL Grammar".

The following are the minimum sets of features available at each conformance level (each level includes the features of the lower levels, and a specific driver may optionally implement further features):

Minimum SQL Grammar:

- Data Definition Language (DDL): CREATE TABLE, DROP TABLE.
- Data Manipulation Language (DML): simple SELECT, INSERT, UPDATE and DELETE
- Simple Expressions: (such as A < B - C).
- Simple character-only data types: CHAR, VARCHAR, LONG VARCHAR.

Core SQL Grammar:

- DDL: ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT, and REVOKE.
- DML: full SELECT.
- Expressions: Subqueries, set functions such as SUM and MIN.
- Numeric Data types: DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE

Extended SQL Grammar:

- DML: Outer Joins, positioned UPDATE, positioned DELETE, SELECT FOR UPDATE, and Unions.
- Expressions: Scalar functions such as SUBSTRING and ABS, date, time, and timestamp literals.
- Data types: BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINARY, DATE, TIME, TIMESTAMP
- Batch SQL statements.
- Procedure calls.

For example, DB2 provides full SQL facilities:

```
      ⊃ 3⊃⎕SQL 'DESCRIBE'
SAMPLE
03.00
DB2/NT
08.01.0003
DB2CLI.DLL
08.01.0003
SAMPLE
ALICE
Extended SQL Grammar
```

Retrieving the Version number of the APLX database driver

Syntax: `[Tie] ⎕SQL 'Version' Interface`

This call returns (as the third element of the explicit result) an integer scalar of the APLX database driver version; 100 corresponds to version 1.00. The Interface parameter is the same as that of the `Connect` keyword, for example `'aplxodbc'`.

Retrieving the database connection and statement handles

Sometimes, you may need to access the underlying ODBC, CLI or other database driver directly, for example using `⎕NA` to access a facility not supported by `⎕SQL`. For this to work, you need the appropriate low-level handles, which you can retrieve using the `'Handle'` keyword.

Syntax: `[Tie] ⎕SQL 'Handle' [Name]`

The information returned (as the third element of the explicit result) is system-specific; for ODBC, if you do not specify a statement name, you get a two-element result of the Environment and Database Connection handles. If you specify a statement, you get a scalar which is the statement handle:

```
      ⎕DISPLAY ⎕SQL 'Handle'
┌→─────────────────────────────────────┐
│ ┌→──────┐ ┌⊖┐ ┌→────────────────┐ │
│ │0 0 0 0│ │ │ │52237280 52237480│ │
│ └~──────┘ └─┘ └~────────────────┘ │
└∊─────────────────────────────────────┘
```

```
      ⎕DISPLAY ⎕SQL 'Handle' 'cs1'
┌→───────────────────────┐
│ ┌→──────┐ ┌⊖┐          │
│ │0 0 0 0│ │ │ 52238968 │
│ └~──────┘ └─┘          │
└∊───────────────────────┘
```

Data Conversions

Because the underlying database will usually support a set of data type for columns which are more extensive than those available in APL, APLX automatically converts data types as follows:

SQL Data Type	*APLX Data Type*
Integer, Small Integer, Tiny Integer, Bit	Integer
64-bit Integer	Float (under 32-bit versions of APLX), or Integer (under APLX64)
Numeric, Decimal, Float, Real, Double	Float
Date, Time, DateTime, Timestamp, Interval	Character vector
Char, VarChar, LongVarChar, CLOB, GUID	Character vector
GUID, DataLink	Character vector
Graphic (i.e. non-ASCII character), VarGraphic, LongVarGraphic	Character vector
UnicodeChar, UnicodeVarChar, UnicodeLongVarChar	Character vector
Binary, VarBinary, LongVarBinary, Blob, OLE object, User-defined Type	Raw (untranslated) Character vector

See also `⎕MC`, which contains the character used to replace Unicode characters which cannot be represented in APLX.

NULL handling

An important concept in SQL is that of NULL data items, i.e. items which have no value. A NULL is handled in a very special (and often counter-intuitive) way in SQL statements; for example, in a SELECT statement, a NULL does not match anything, including another NULL. For a character field, a NULL is not the same as a zero-length string.

`⎕SQL` follows a very simple convention to represent NULL data items, both in parameter substitution (using `'Execute'`) and in returning result sets. A NULL is represented by an empty numeric vector. Note that this can easily be distinguished from a zero-length string, which will be represented by an empty character vector.

Summary of ⎕SQL Syntax

Connect to database:
```
 (RC ERRMSG CNS) ← [Tie] ⎕SQL 'Connect' Interface ConnectionString
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Connect' Interface DSN [User] [Password]
```

Execute SQL statement immediately:
```
 (RC ERRMSG RESULT) ← [Tie] ⎕SQL 'Do' Statement
```

Prepare SQL statement for later execution:
```
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Prepare' Name Statement
```

Execute prepared statement:
```
 (RC ERRMSG CURSORNAME) ← [Tie] ⎕SQL 'Execute' Name [Param1] [Param2]...
```

Fetch rows from statement result set:
```
 (RC ERRMSG ROWS) ← [Tie] ⎕SQL 'Fetch' Name [Count]
```

Describe result set:
```
 (RC ERRMSG DESCRIPTION) ← [Tie] ⎕SQL 'Describe' Name
```

Close statement
```
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Close'
```

Set/Query AutoCommit:
```
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Autocommit' Flag
 (RC ERRMSG FLAG) ← [Tie] ⎕SQL 'Autocommit'
```

Set/Query Isolation Level:
```
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Isolation' Code
 (RC ERRMSG CODE) ← [Tie] ⎕SQL 'Isolation'
```

Commit/Rollback transaction:
```
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Commit'
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Rollback'
```

Retrieving the catalog of tables and columns:
```
 (RC ERRMSG TABLES) ← [Tie] ⎕SQL 'Tables' [Catalog] [Schema] [Table] [Types]
 (RC ERRMSG COLUMNS) ← [Tie] ⎕SQL 'Columns' [Catalog] [Schema] [Table] [Column]
```

Examining details of the connection:
```
 (RC ERRMSG DETAILS) ← [Tie] ⎕SQL 'Describe'
```

Retrieving the Database Connection and Statement handles
```
 (RC ERRMSG CONNHANDLE) ← [Tie] ⎕SQL 'Handle'
 (RC ERRMSG STMTHANDLE) ← [Tie] ⎕SQL 'Handle' Name
```

Return list of tie numbers in use:
```
 TIES ← ⎕SQL 'Connections'
```

Disconnect from database:
```
 (RC ERRMSG EMPTY) ← [Tie] ⎕SQL 'Disconnect'
```

Query available Data Source Names
```
 (RC ERRMSG DSNS) ← [Tie] ⎕SQL 'Datasources' Interface
```

Query Version Number of the APLX database interface:
```
 (RC ERRMSG VERSION) ← ⎕SQL 'Version' Interface
```

⎕SS String Search/Replace

The system function `⎕SS` finds one or more text patterns in a character vector, and optionally replaces them with different sub-strings. It can either do a simple search, matching characters one-by-one, or search for regular expressions, which allow you to specify sophisticated pattern-matching rules in a compact form.

The right argument is a nested vector of either two or three elements. The first element is the character vector in which you want to search (we refer to this as the *string*). The second element contains the pattern or patterns you want to search for. If you are searching for just one pattern, it is a character vector. If you are searching for more than one pattern, it is a vector of character vectors. (Character scalars are treated as length-one character vectors). The optional third element contains replacement sub-strings for search-and-replace operations. The optional left argument determines the type of search.

Monadic form: Simple search

If the right argument is of length two, then `⎕SS` returns information about the occurrences of the pattern (or patterns) in the string. The exact information returned depends on the type of search you are doing. In its simplest form, looking for a single pattern in the string, it returns a vector of the positions where the pattern was found in the string:

```
      TEXT←'Potatoes are bad for you, very bad.'
      ⎕SS TEXT 'bad'
14 32
```

The pattern 'bad' has been found at character positions 14 and 32. *Note:* The result returned takes account of the index origin, `⎕IO`. All the examples shown assume an index origin of 1.

If you supply more than one pattern, then the simple search returns an N by 2 integer matrix, where each row corresponds to one match. The first column is the position where the match was found. The second column is the pattern number which was found. (Both values take account of the index origin, `⎕IO`) For example:

```
      ⎕SS TEXT ('bad' 'you')
14 1
22 2
32 1
```

This means that three matches were found. The first match was at position 14, where pattern 1 ('bad') was found. The second match was at position 22, where the second pattern ('you') was found. The third match was at position 32, where pattern 1 was found again.

When used in this monadic form, `⎕SS` searches for a match, at a given position, by looking at each of the patterns in turn (in the order in which they appear in the list). When it finds a match, it does not consider any more patterns at that position. Instead, it increments the position by one character, and again looks at each pattern. You can modify the way in which this works by supplying a left argument with modifier flags - see below.

Monadic form: Simple search-and-replace

If the right argument is of length three, then `⎕SS` carries out a string-search-and-replace operation, returning the original string but with each occurrence of one of the supplied patterns replaced by a new sub-string specified in the third element. Like the second element, the third element can either be a character vector (in which case all occurrences of any of the patterns are replaced by the same sub-string), or it can be a vector of character vectors. In the latter case, it should be of the same length as the second element, and it specifies a separate replacement sub-string for each of the patterns. For example:

```
      ⎕SS TEXT 'bad' 'good'
Potatoes are good for you, very good.
      ⎕SS TEXT ('bad' 'you') ('good' 'me')
Potatoes are good for me, very good.
      ⎕SS TEXT ('bad' 'you') '***'
Potatoes are *** for ***, very ***.
```

Subject to available workspace, the replacement sub-strings can be of any length. A zero-length replacement has the effect of deleting all occurrences of the matched pattern from the returned string. If the replacement sub-strings are longer than the patterns they replace, the returned string will be longer than the original string.

It is important to order the patterns and sub-strings correctly. When doing a search and replace, `⎕SS` does a replacement as soon as it finds a match for any of the patterns, considered in the order in which they appear in the list of patterns. After the replacement, the string pointer is incremented past the replacement, and the search starts again.

Dyadic form

The optional left argument is either a single integer, or a pair of integers. The first element determines the type of search, as follows:

Value	*Search type*	*Result (single search)*	*Result (multiple search)*	*Result (replace)*
0	**Simple search**	Integer vector of match positions	Nx2 Integer matrix of match position, pattern number	Character vector

1	**Regular-expression search**	Nx2 Integer matrix of match position, match length	Nx3 Integer matrix of match position, match length, pattern number	Character vector
2	**Regular-expression extract**	Nested vector of vectors of sub-strings	Nested vector of vectors of sub-strings	Not applicable

The second element is an integer which is the sum of a set of flags which modify how the search is carried out, as follows:

Flag	*Effect*	*Applies to*
1	Case-insensitive search	All searches
2	Stop after the first match position has been found	All searches
4	After finding a match, advance by 1, rather than the length of the match	Any search, but no effect on search-and-replace
8	When doing a multiple search, return all matches at all positions, not just the first pattern that matches	Any search, but no effect on search-and-replace
16	Treat string as multi-line, so ^ matches start of each line. rather than start of whole string only	Regular-expressions only
32	Cause a '.' in the pattern to match new-line delimiters	Regular-expressions only
64	Treat only CR as newline	Regular-expressions only
128	Treat only LF as newlines	Regular-expressions only
256	Treat only CR-LF pairs as newlines	Regular-expressions only

For example, if the second element is 5, then the search is case-insensitive and returns all matches at each position, not just the first. These options are described in detail below.

If you omit the left argument, `⎕SS` carries out a simple, case-sensitive search which advances the search position by 1 after each match. This is equivalent to a left argument of `0 4`.

Specifying the search type using the dyadic form of ⎕SS

Search type 0: Simple search

If the first element of the left argument is zero, ⎕SS carries out a simple search, similar to that of the monadic case, except that you have control over the exact way the search works, by setting options in the second element of the left argument as described below.

Search type 1: Regular-expression search

If the first element of the left argument is 1, ⎕SS carries out the search by treating the pattern (or patterns) as regular expressions. Because regular expressions can match against arbitrary-length sequences in the string, ⎕SS returns the length of each match as well as its position. If you supply only one pattern, the result of a search is an N by 2 matrix, where the first column in the position and the second is the length of the match. For example:

```
      1  ⎕SS 'A pixel color (or colour)' 'colou?r'
 9 5
19 6
```

In this example, the '?' meta-character in the regular-expression makes the 'u' optional, so ⎕SS has found a match at position 9 of length 5 ('color'), and a match at position 19 of length 6 ('colour'). (See the separate page on regular expression syntax for details). Contrast this with the simple search, which for a single search pattern returns a vector.

You can also supply multiple regular expressions. In this case, ⎕SS returns an N by 3 matrix, with the pattern number as the third column:

```
      1  ⎕SS 'A pixel color (or colour), such as light grey' ('colou?r' 'gr[ea]y')
 9 5 1
19 6 1
42 4 2
```

In this case we have searched for two separate regular-expressions, one which can match 'color' or 'colour', and one which can match 'gray' or 'grey'. The first was found at positions 9 and 19, and the second at position 42.

Note that this is different from supplying a single regular expression which itself can match either set of alternatives:

```
      1  ⎕SS 'A pixel color (or colour), such as light grey' 'colou?r|gr[ea]y'
 9 5
19 6
42 4
```

Replacing matched patterns using regular expressions

As with simple searches, you can also do search-and-replace using regular expressions. As soon as a match is found against any of the regular-expression patterns supplied, it is replaced by the corresponding sub-string supplied in the third element of the right argument to ⎕SS. For example, the regular-expression `'<[^>]+>'` can be used to match against HTML tags (it finds any pattern starting with a left angle bracket, followed by any number of characters EXCEPT a right angle bracket, and then followed by a right angle bracket):

```
      1 ⎕SS '<P>This is <B>bold</B></P>' '<[^>]+>'
 1 3
12 3
19 4
23 4
```

The pattern has matched in four places. By providing a replacement string, we can replace any HTML tags found with our own text, or strip them out completely by replacing with an empty vector:

```
      1 ⎕SS '<P>This is <B>bold</B></P>' '<[^>]+>' '∆'
∆This is ∆bold∆∆
      1 ⎕SS '<P>This is <B>bold</B></P>' '<[^>]+>' ''
This is bold
```

When replacing regular expressions, you can also extract sub-strings which matched the pattern, and use those in your replacement pattern. Wherever the replacement contains a backslash character `'\'` followed by a digit 0 to 9, the text which gets replaced is extracted from the matched text in the original string. `'\0'` always represents the whole of the original part of the string which matched the pattern:

```
      1 ⎕SS '<P>This is <B>bold</B></P>' '<[^>]+>' '[HTML: \0]'
[HTML: <P>]This is [HTML: <B>]bold[HTML: </B>][HTML: </P>]
```

`'\1'` to `'\9'` are replaced by sub-strings in extracted from the matched patterns. These sub-strings are delimited (in the regular-expression pattern) by being placed in parentheses, and the corresponding part of the matched string is used to replace them. Thus, if '\1' appears in the replacement sub-string, it is replaced by that part of the original string which matched against the first parenthesized part of the pattern:

```
      1 ⎕SS '<P>This is <B>bold</B></P>' '<([^>]+)>' '[HTML: \1]'
[HTML: P]This is [HTML: B]bold[HTML: /B][HTML: /P]
```

If there is no such sub-string (for example, if we used `'\2'` in the above replacement expression), the inserted sub-string would be empty. If you want to include a literal `'\'` in the replacement string, you need to double it up as `'\\'`.

Search type 2: Extracting sub-strings from matched regular expressions

If you supply 2 as the search type, `⎕SS` again does a regular-expression search. But instead of returning the *positions* of matches, it returns the *text* of the matches. This can be used to rapidly extract all the strings which match a certain pattern. For example, suppose you want to extract all valid e-mail addresses from a string, but you don't care where they appear. For example, this regular expression can be used to match against most e-mail addresses, if used in an case-insensitive search: `'\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b'`. It matches against sequences which begin at a word boundary (`'\b'`), and comprise a series of letters, digits and other characters followed by an `'@'` sign, then have a series of words separated by periods, and end with a two to four letter top-level domain name and a word boundary. If we use it in a case-insensitive search, it will find what we want:

```
      MAIL←'Try e-mailing bill.gates@microsoft.com or, better, jim@microapl.co.uk'
      1 1  ⎕SS MAIL '\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b'
15 24
52 18
```

The search has found the two e-mail addresses, one (of length 24) at position 15, and the other (of length 18) at position 52.

By using search-type 2, we can extract the actual strings immediately:

```
      2 1  ⎕SS MAIL '\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b'
  bill.gates@microsoft.com   jim@microapl.co.uk
```

What gets returned is a nested vector, with one element per match. Each element in turn is a nested vector of character vectors. The first is the whole matched pattern, i.e. the equivalent of `'\0'`. There then follows, for each match, the extracted sub-strings for any parenthesized sub-patterns in the regular expression, i.e. the equivalent of `'\1'`, `'\2'` etc.

In our example above, there are no parenthesized sub-patterns, so we only get one character vector for each match:

```
    ⎕DISPLAY 2 1  ⎕SS MAIL '\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b'
┌→─────────────────────────────────────────────────────────────┐
│ ┌→───────────────────────────┐ ┌→─────────────────────┐      │
│ │ ┌→───────────────────────┐ │ │ ┌→─────────────────┐ │      │
│ │ │bill.gates@microsoft.com│ │ │ │jim@microapl.co.uk│ │      │
│ │ └────────────────────────┘ │ │ └──────────────────┘ │      │
│ └∊───────────────────────────┘ └∊─────────────────────┘      │
└∊─────────────────────────────────────────────────────────────┘
```

(*Caution:* There is one more level of nesting than you might expect here.) So far, this does not do anything more than you could do by extracting the matches yourself from the match positions and lengths. But if you use parentheses in the pattern, you can use this mechanism to extract specific parts of the matches. For example, if we put parentheses around the parts of the pattern before and after the `'@'` sign, we can easily distinguish the name part from the server/domain part of the address. Each extracted match will comprise a length three-vector:

```
    ⎕DISPLAY 1⊃¨2 1  ⎕SS MAIL '\b([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,4})\b'
┌→─────────────────────────────────────────────────┐
│ ┌→───────────────────────┐ ┌→─────────────────┐  │
│ │bill.gates@microsoft.com│ │jim@microapl.co.uk│  │
│ └────────────────────────┘ └──────────────────┘  │
└∊─────────────────────────────────────────────────┘
    ⎕DISPLAY 2⊃¨2 1  ⎕SS MAIL '\b([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,4})\b'
┌→─────────────────────┐
│ ┌→─────────┐ ┌→──┐   │
│ │bill.gates│ │jim│   │
│ └──────────┘ └───┘   │
└∊─────────────────────┘
    ⎕DISPLAY 3⊃¨2 1  ⎕SS MAIL '\b([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,4})\b'
┌→─────────────────────────────────────┐
│ ┌→────────────┐ ┌→────────────┐      │
│ │microsoft.com│ │microapl.co.uk│     │
│ └─────────────┘ └──────────────┘     │
└∊─────────────────────────────────────┘
```

You can also supply multiple regular-expression patterns when using this form. In this case, the result is of the same form; for each match, you get the extracted sub-string or sub-strings for the pattern which matched.

Using modifier flags to specify how the search should be carried out

You can modify the way the search is done by providing the second element of the left argument as the sum of various modifier flags. These work as follows:

Case-insensitive search (modifier 1): By default, `⎕SS` does a case-sensitive search. If you supply a modifier flag of 1 (for either a simple or regular-expression search), the search will be case-insensitive. This applies both to the ordinary unaccented letters (A to Z and a to z), and to accented letters, so that for example È will match against è

```
      TEXT←'Potatoes are bad for you, very bad.'
      0 ⎕SS TEXT 'potatoes'        ⍝ Case-sensitive search, empty result

      0 1 ⎕SS TEXT 'potatoes'      ⍝ Case-insensitive search, one match
1
```

Stop at first match position (modifier 2): This flag causes the search to terminate after the first match position has been found. In a search operation, this normally means that only one match will be returned, but if you have also set flag 8, it is possible that more than one match will be returned because more than one pattern matches at the first hit position. For a replace operation, this flag has the effect that only the first matched pattern is replaced.

```
      0 ⎕SS TEXT 'bad'
14 32
      0 2 ⎕SS TEXT 'bad'
14
      0 ⎕SS TEXT 'bad' 'good'
Potatoes are good for you, very good.
      0 2 ⎕SS TEXT 'bad' 'good'
Potatoes are good for you, very bad.
```

After a match, advance the search position by 1 (modifier 4): Without this flag, once it has found a match at a given position, the dyadic form of `⎕SS` advances the starting position for the next search to beyond the end of the matched pattern, and starts searching from there, so that any given position in the string can appear within at most one match. If you set modifier flag 4, it will instead advance the search position by 1. This means that it may find further matches which overlay the first match, either from the same pattern or a later pattern in a list of multiple patterns. Consider this example:

```
      COMPLAINT←'Even my sandwich was sandy.'
      0 0 ⎕SS COMPLAINT ('sand' 'sandy' 'and')
 9 1
22 1
```

In this example, without the modifier flag 4, the first match it finds is against the pattern 'sand' at position 9. It then increments the search position to point at the 'w' of 'sandwich', and continues searching. It finds 'sand' again at position 22, and then starts searching from the final 'y'. As a result, it does not return a match for 'and' at positions 10 and 23. In addition, it can never return a match for 'sandy', because it will always find a match for 'sand' first at any position which otherwise would match 'sandy'.

By setting modifier 4, you can cause `⎕SS` to find further matches of other sub-patterns starting within matches already found, in this case the 'and' within 'sand':

```
      0 4 ⎕SS COMPLAINT ('sand' 'sandy' 'and')
 9 1
10 3
22 1
23 3
```

Find all matching patterns everywhere in the string (modifier 8): Normally, once it has found a match at a given position, `⎕SS` stops looking for any further matches *at the same position* against any later patterns in the list. Thus, in the previous example, it still did not regard 'sandy' as a match at position 22. If you set modifier flag 8, it will also see whether any of the other patterns in the supplied list also match at the current position (because they start with the same prefix), and it will then advance the search position by 1. The effect is to find every possible independent match, including both 'sand' and 'sandy' at position 22:

```
      0 8 ⎕SS COMPLAINT ('sand' 'sandy' 'and')
 9 1
10 3
22 1
22 2
23 3
```

Note: Modifiers 4 and 8 have no effect if you are doing a search-and-replace operation, because the next search point is always advanced to the character after the replacement. For upwards compatibility with previous versions of APLX and APL.68000, the monadic form of `⎕SS` acts as though modifier 4 were set.

Modifiers 4 and 8 can be used with regular-expression searches, but the results may be very strange if you use repeat operations in the pattern. For example, using our e-mail matching pattern:

```
    2 5  ⎕SS MAIL '\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b'
  bill.gates@microsoft.com   .gates@microsoft.com   gates@microsoft.com
jim@microapl.co.uk
```

Because the `'.'` in `'bill.gates'` is valid as a word delimiter, this has matched at three separate places within the same e-mail address, which is probably not what you want!

Treat string as multi-line, so ^ matches start of each line (modifier 16): This flag applies only to regular-expression searches. It is relevant when the string being searched contains multiple, independent lines of text delimited by newline characters. If you use a pattern containing the `'^'` or `'$'` metacharacters (match start or end of line), this flag will cause the match to occur on every line, not on the whole string. For example, suppose we have a carriage-return delimited list of names. If we don't set this flag, then the string is considered as a single piece of text:

```
      Composers←'Ludwig Van Beethoven',⎕R,'Richard Wagner',⎕R,'Gustav Mahler'
```

```
     Composers
Ludwig Van Beethoven
Richard Wagner
Gustav Mahler
      1 1 ⎕SS Composers '^[A-Z]+'
1 6
      1 1 ⎕SS Composers '\b[A-Z]+$'
44 6
      2 1 ⎕SS Composers '^[A-Z]+'
  Ludwig
      2 1 ⎕SS Composers '\b[A-Z]+$'
  Mahler
```

The `'^'` and `'$'` meta-characters have matched only at the start and end of the whole string. By adding modifier flag 16, we can treat the three lines individually:

```
      1 17 ⎕SS Composers '^[A-Z]+'
 1 6
22 7
37 6
      1 17 ⎕SS Composers '\b[A-Z]+$'
12 9
30 6
44 6
      2 17 ⎕SS Composers '^[A-Z]+'
  Ludwig   Richard   Gustav
      2 17 ⎕SS Composers '\b[A-Z]+$'
  Beethoven   Wagner   Mahler
```

By default, any of CR, LF, or CR-LF pairs are treated as newlines, so text originating on Windows, MacOS and Unix-style systems can be handled automatically. You can force the search to treat only one of these possibilities as a valid newline by including modifier 64 (CR only), 128 (LF only) or 256 (CR-LF pairs only).

Cause a* `'.'` *in the pattern to match new-line delimiters (modifier 32): Normally, the period (`'.'`) wild-card character does not match a newline character. This flag forces it to do so.

Technical considerations

Performance

For simple searches with a single search pattern, `⎕SS` uses the Boyer-Moore-Horspool algorithm, which is extremely fast for most combinations of strings and patterns. Long patterns are generally faster to search for.

For simple searches with multiple search patterns, `⎕SS` uses a simple search algorthm with optimizations, so performance should remain good for reasonable numbers of search patterns.

Case-insensitive searching will be slightly slower than case-sensitive searches. Regular-expression searching is slower than the equivalent simple searches.

Regular-expression search engine

The regular-expression search engine used in APLX is based on version 7.1 of Perl Compatible Regular Expressions (PCRE), an open-source product written by Philip Hazel of the University of Cambridge. See the Legal Notice which applies to PCRE.

The maximum length of string which can be searched using regular expressions is 2GB (on 64-bit versions of APLX, simple string searches can be done on strings longer than 2GB).

⎕STOP Stop List

Nomadic system function to stop functions or operators by setting 'break points'. Used with a numeric left argument, and the text name of a function, operator or method as the right argument, it will halt the function or operator before executing the line number(s) given in the list.

A left argument of 0, or an empty vector, removes stop control.

Used without a left argument, the current stop settings are returned.

To use `⎕STOP` on a method of a class, the right argument should be the fully-qualified method name (i.e. the class name, a period, and the method name).

Note that, under desktop editions of APLX, the interpreter will bring up the Debugger window when a stop point is reached.

⎕SVC Shared Variable Control

This system function is used to control access to a shared variable by the two processors sharing it, or to monitor the access control vector.

One-argument form

The right argument is a character matrix of names, or a vector containing one name. The result is the effective value of the 'control vector' for each name. The 'control vector' is a 4 element boolean. A 1 in the 'control vector' has the following effects:

Position	Effect
1	You cannot set a new value for the shared variable until the other processor has set or used the variable
2	The other processor cannot set a new value for the shared variable until you have set or accessed it

```
      3       You cannot use the value of the shared variable until it is set by
              the other processor
      4       The other processor cannot use the shared variable until you set it
```

The default setting for APLX is 1 1 1 1.

Two-argument form

The right argument is as above, but the left argument is a 4 column boolean matrix (or vector, for a vector right argument) which contains the proposed new values for the 'control vector'. A vector left argument is applied to all names in the right argument. The proposed new values are 'OR-ed' with the most recent values set by the other processor. The result is the new 'control vector' for the variables named. If the name is that of a variable that is not shared, the 'control vector' remains at 0 0 0 0.

⎕SVO Shared Variable Offer

This system function allows the user to establish a shared variable for use with a nominated Auxiliary Processor, or to interrogate about the degree of coupling established. An Auxiliary Processor is an external shared library which extends the facilities of the APLX language. The Shared Variable is the route by which the user communicates with this subroutine. Further details about APLX's Auxiliary Processor system are given in a separate section.

One-argument form

The right argument is a character matrix of names, or a single name. The result is the 'degree of coupling' of each of the names. The values for the 'degree of coupling' are:

```
      0       No coupling - no offer to share has been made, or the name is
              ill-formed.
      1       An offer to share has been made but not accepted
      2       An offer to share has been made and accepted.
```

Two-argument form

The right argument is a character matrix of names to be shared. Each row of the matrix may contain one or two names. The second in each pair is known as the 'surrogate' name. The 'surrogate' name is used to match a name offered by the other processor. If only one variable is to be shared the right argument may be a vector. The left argument is the number (or numbers) of the processor numbers with which the share offers are being made. The result is again the 'degree of coupling' produced.

```
            1 ⎕SVO 'NAME SURROGATE' (Shares NAME with processor 1, using
      2                              SURROGATE as the surrogate name).
```

⎕SVQ Shared Variable Query

This system function is used to query the processors available, or to establish the existence of any names offered by the processor identified. If used with an empty vector right argument, the result is a vector of processors with share offers outstanding. Used with a right argument which is the number of a processor, the result is a matrix of names offered by that processor.

```
            ⎕SVQ ''
      1                             (Processor 1 is available)
```

⎕SVR Shared Variable Retract

The dyadic system function ⎕SVR is used to retract a shared variable after use (see ⎕SVO). The right argument is a character array of names, or a vector containing one name. The result is the 'degree of coupling' in effect before the retraction command was issued.

```
            1 ⎕SVR 'NAME'
      2
```

⎕SYMB Symbol Table Used/Total Count

The niladic system function ⎕SYMB returns a two-element integer vector. The first element is the number of symbol table entries currently in use in the workspace. The second is the total size of the symbol table:

```
      )SYMBOLS
IS 1026, USED 68
      ⎕SYMB
68 1026
```

For more information on the symbol table, see the description of the system command)SYMBOLS.

⎕T Tab Character

The niladic system function `⎕T` returns a Horizontal tab character.

⎕TC and ⎕TCxx Terminal Control Characters

The niladic system function `⎕TC` returns a three-element vector containing the three control characters: backspace, carriage return and linefeed. This is compatible with IBM's APL2 equivalent.

In addition, to assist migration of APL code from APL*Plus, APLX also supports the following niladic system functions which return specific 'terminal-control' characters as scalars:

```
⎕TCBEL  Bell character, equivalent to ⎕C[⎕IO+7]
⎕TCBS   Backspace, equivalent to ⎕B or ⎕TC[⎕IO+0]
⎕TCDEL  Delete character, equivalent to ⎕C[⎕IO+32]
⎕TCESC  Escape character, equivalent to ⎕C[⎕IO+27]
⎕TCHT   Horizontal tab character, equivalent to ⎕T or ⎕C[⎕IO+9]
⎕TCFF   Formfeed character, equivalent to ⎕C[⎕IO+12]
⎕TCHT   Horizontal tab character, equivalent to ⎕T or ⎕C[⎕IO+9]
⎕TCLF   Linefeed character, equivalent to ⎕L or ⎕TC[⎕IO+2]
⎕TCNL   Newline (carriage return) character, equivalent to ⎕R or ⎕TC[⎕IO+1]
⎕TCNUL  Null character, equivalent to ⎕AV[⎕IO+0]
```

⎕TF Transfer Form

The dyadic system function `⎕TF` returns the Transfer Form of the item whose name is its right argument. Alternatively it will decode any transfer form supplied as its right argument and return the name of the successfully decoded object. The transfer form of an object is a text representation of an APL variable, function or operator, suitable for transmission between dissimilar computer systems or implementations of APL. Two forms are allowed, the standard and extended forms. The standard form consists of a header portion and a data portion, with the header having the following format:

Name	Description
Type Code	'F' for function 'N' for a numeric array 'C' for a character array
Name	Name of the object, followed by a blank
Rank	Character form of the rank of the object
Shape	Character form of the shape of the object followed by a blank
Data	Character form of the object

The standard transfer form cannot be used for mixed or nested variables or for user-defined operators.

The extended transfer form is a character string which, if executed, would reconstitute the item in question.

If the `⎕TF` function is unsuccessful, an empty vector is returned.

See also the `)OUT` and `)IN` system commands which will decode or create Transfer Forms for a list of objects.

```
      SIMPLE←'TEXT STRING'    (Sample data)
      MIXED←'ABC' (2 2⍴⍳4)
      ∇R←A FN B
[1]   R←A ÷ B
[2]   ∇
```

Encoding items:

```
      1 ⎕TF 'SIMPLE'          (Standard transfer form)
CSIMPLE 1 11 TEXT STRING      (Character, rank 1, shape 11)
      1 ⎕TF 'MIXED'           (Standard transfer form won't work on mixed)

      1 ⎕TF 'FN'
FFN 2 2 8 R←A FN BR←A÷B       (Function called FN, rank 2, shape 2 8)

      2 ⎕TF 'SIMPLE'          (Extended transfer form)
SIMPLE←'TEXT STRING'
      2 ⎕TF 'MIXED'
MIXED←('ABC') (2 2⍴1 2 3 4)
      2 ⎕TF 'FN'
⎕FX 'R←A FN B' 'R←A÷B'

      1 ⎕TF '⎕WA'             (Similar format to encode system variables)
N⎕WA 0 740322
      2 ⎕TF '⎕WA'
⎕WA←740322
      1 ⎕TF '⎕PW'
N⎕PW 0 80
      2 ⎕TF '⎕TS'
⎕TS←1990 5 31 0 36 52 734
```

Decoding objects:

```
      TF1←1 ⎕TF 'SIMPLE'
      TF2←2 ⎕TF 'MIXED'
      )ERASE SIMPLE MIXED     (Remove the original objects)
      )VARS
TF1     TF2
      1 ⎕TF TF1               (Successful decode - name returned)
SIMPLE
      )VARS
SIMPLE  TF1     TF2           (Object reappears)
      1 ⎕TF TF2               (Attempt to decode an extended version)
                              (Empty vector and no object)
      )VARS
SIMPLE  TF1     TF2
      2 ⎕TF TF2               (Decode using the correct, extended form)
MIXED
```

⎕THIS Reference to current object

The niladic system function `⎕THIS` returns a reference to the user-defined object whose method is currently being executed. If no method is currently executing, it returns a Null object.

```
      ∇MyClass{
{MyClass}:
      ∇RunMe
[1]   'I am an instance of ',⎕THIS.⎕CLASSNAME
[2]   'In fact, I am an instance of ',⎕CLASSNAME
[3]   ⎕CLASS ⎕THIS
[4]   ∇
{MyClass}:
      }

      M←⎕NEW 'MyClass'
      M.RunMe
I am an instance of MyClass
In fact, I am an instance of MyClass
{MyClass}
      ⎕THIS
[NULL OBJECT]
```

When a method is running, `⎕THIS` is implicit when accessing class members or system methods. That is why `⎕THIS.⎕CLASSNAME` and `⎕CLASSNAME` give the same result in the above example. For this reason, you normally do not need to use `⎕THIS` except when you want to pass a reference for the current object to another object or to a global function.

⎕TIME Time/Date Text

The niladic system function `⎕TIME` returns, as a simple text string, the time and date in the format currently selected by 6 `⎕CONF` (see `⎕CONF`).

⎕TR Translate Text to/from External

The dyadic system function ⎕TR translates character data from internal (⎕AV) representation to the external representation used in non-APL applications, and vice versa. Normally, this translation is done automatically when text data is transferred to/from APL, but sometimes you may need to translate data explicitly.

The right argument is a character array of any size and shape. The left argument is an integer code indicating in which direction you wish to translate. This is 1 for External (Extended ASCII) to Internal, and ¯1 for Internal to External. The result is a character array of the same size and shape as the right argument, with the characters translated as requested.

```
      ⎕AF 'hello'
232 229 236 236 239
      ⎕AF ¯1 ⎕TR 'hello'
104 101 108 108 111
```

⎕TS Timestamp

The niladic system function ⎕TS returns a seven element vector showing the current time as:

```
          Year, month, day, hour, minute, second, millisecond
```

⎕TT Terminal Type

The niladic system function ⎕TT returns terminal type as a code number. Normally ⎕TT returns 0. The meaning of any other codes returned by ⎕TT depends on the hardware on which APLX is installed.

⎕TRACE Trace

The nomadic system function ⎕TRACE displays the values of the assignments made on the lines specified in the left argument when the function, operator or method named in the right argument is executed.

Used without a left argument, the current trace settings are returned.

A left argument of 0, or an empty vector, removes trace control. For further details, see the earlier chapter on Stop and Trace control.

To use ⎕TRACE on a method of a class, the right argument should be the fully-qualified method name (i.e. the class name, a period, and the method name).

⎕UCS Convert text to/from Unicode

Unicode is a worldwide standard which encodes characters as integers or 'code points'. It includes representations of all the international and special characters used in modern computer applications. APL characters, including all those used in APLX, are defined in the Unicode standard, although there are some ambiguities about a few of them. This makes it possible to exchange character data between APLX and other applications (including other APL interpreters which support Unicode), without encountering problems with character translation and 'code tables', provided that the text being transferred can be represented in the APLX character set.

The system function ⎕UCS translates Unicode values to the equivalent character in the APLX character set (if there is one), and vice versa. It takes a right argument, which must be a simple character or integer array.

If the argument is a character array, ⎕UCS returns an integer array of the same shape, containing the Unicode representation of each character. This will be a number within the range 0 to 65535, because all of the characters supported in APLX fall into the basic Unicode range.

If the argument is an integer array, ⎕UCS returns a character array of the same shape, containing the APLX character corresponding to each Unicode value provided. Unicode values which have no equivalent in the APLX character set are converted to the current value of ⎕MC (Missing Character). By default, this is a question mark.

For example:

```
      ⎕UCS 'X←⍳10'
88 8592 9075 49 48
      ⎕UCS 88 8592 9075 49 48
X←⍳10
```

```
          ⎕UCS 937 8364 223
      ?€β
```

In the last example, the Unicode value 937 (hex `03A9`, representing the Greek capital omega character) was translated to the 'missing character' value (question mark) because it has no equivalent in the APLX character set.

A different 'missing character' can be set using `⎕MC`:

```
          ⎕MC←'$'
          ⎕UCS 937 8364 223
      $€β
```

See the section on the APLX Character Set for details of the mapping between APLX characters and Unicode.

⎕UL User Load

The niladic system function `⎕UL` gives the number of users or APLX tasks active on the system. Returns 1 on a single-tasking system. The exact meaning on multi-user or multi-tasking systems is implementation dependent. Under MacOS, Linux and Windows, it returns the number of APLX tasks running.

⎕VI Verify formatted input

The monadic system function `⎕VI` is used in conjunction with `⎕FI` to validate a text string and convert it to numeric form. In both cases, the argument is a character vector (or scalar), containing one or more sub-strings of characters separated by blanks. For each non-blank sub-string, `⎕VI` returns a 1 if the sub-string represents a valid number, and 0 if it does not. `⎕FI` returns the numeric representation of the sub-string, or 0 if it is not a valid number. Numbers are validated and converted in the same way as normal APL input. Scientific notation is supported, and negative numbers are prefixed by the high minus (¯) character. For example:

```
      ⎕FI '100.32 $4 2,,3 0 12.2 ¯3 +2 -2'
100.32 0 0 0 12.2 ¯3 0 0
      ⎕VI '100.32 $4 2,,3 0 12.2 ¯3 +2 -2'
1 0 0 1 1 1 0 0
```

`⎕VI` and `⎕FI` are usually used to validate and convert user input, or to convert text files to numeric form. To allow users to enter negative numbers using the ordinary (non-APL) minus sign, you can use `⎕SS` to translate minus to high-minus first. To allow comma-delimited input, use `⎕SS` to translate comma to space:

```
      STRING←'3,-2,45.5,-0.08'
      ⎕VI STRING
0
      ⎕SS (STRING; ('-';','); ('¯';' '))
3 ¯2 45.5 ¯0.08
      ⎕VI ⎕SS (STRING; ('-';','); ('¯';' '))
1 1 1 1
      ⎕FI ⎕SS (STRING; ('-';','); ('¯';' '))
3 ¯2 45.5 ¯0.08
```

⎕W Weekdays

The niladic system function `⎕W` returns a character matrix containing the days of the week.

```
      ⎕W
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
```

⎕WA Workspace Available

The niladic system function `⎕WA` returns the number of bytes remaining in the workspace.

See also `⎕WSSIZE` which returns the total size of the workspace, including the used area.

⎕WARG Argument to event callback function

Obsolescent: Use `⎕EVA` *instead*.

The `⎕WARG` niladic system function is valid only inside a `⎕WE` callback function, run by APLX as the result of an event occurring in one of your windows or other objects. It is used to pass data associated with the event from an external control or server, or another APL task. See `⎕EVA`.

⎕WE Wait for Event

The ⎕WE system function takes a right argument, which is either a numeric scalar, or a character vector, or a scalar reference to a window. It can optionally also take a left argument.

If the right argument is a numeric scalar, it represents a timeout value in seconds. This timeout value represents (approximately) the maximum time that ⎕WE will wait if no events for which callbacks have been defined occur. The timer is reset when a callback runs. If the timeout value if negative, ⎕WE will never return (unless you interrupt it). If the timeout value is zero, ⎕WE will execute at most one callback and then return, so that if no callbacks are pending it will return immediately.

For example:

Sit in a loop executing callbacks. If none occur for 2.5 seconds, return:

```
    R ← ⎕WE 2.5
```

Sit in a loop executing callbacks. If none pending, wait forever:

```
    R ← ⎕WE ¯1
```

Quickly check for events; if a callback is pending, run it, else return immediately:

```
    R ← ⎕WE 0
```

If the right argument is a character vector, ⎕WE will interpret the character vector as the name of a window (an instance of one of the System Classes: `Window`, `Form`, `Document` or `Dialog`). Alternatively, the right argument can be a scalar reference to a window created using ⎕NEW. In either case, ⎕WE will not return as long as that window exists, is open, and is visible. (If the window is not open and visible when ⎕WE is called, or if the window name is invalid, it will return immediately). In the meanwhile it will execute callback functions as events occur. This means that execution of your program will wait until the window has been closed or hidden, and is thus very useful for making your program flow pause until a modal dialog has ended. You can also use this syntax of ⎕WE for your highest-level window, so that when the main window is closed control automatically returns to APL desk-calculator mode. For example, the function run by the latent expression of an APL workspace which uses user-interface objects might be something like this:

```
    ∇START;X;Main
[1] ⍝ Create top-level window and wait for events
[2]  Main←'⎕' ⎕NEW 'Window'      ⍝ Create the main window
[3]  CreateControls              ⍝ Create controls
[4]  X←1 ⎕WE Main                ⍝ Process events
    ∇
```

In this example, the function will not return until the main window has been closed.

The optional left argument to `⎕WE` is a single 0 or 1. If it is 0 or omitted, `⎕WE` will execute any callbacks which correspond to events which have accumulated since it was previously called. If the left argument is 1 (as in the example above), it will first flush any old events from the event queue. This is recommended for the top-level event loop, since it means that if you restart the program after an error, old stale events will not be waiting in the queue to be executed (possibly referring to objects which no longer exist).

The result of `⎕WE` (if it returns at all) is usually an empty vector. However, it is possible for `⎕WE` to return a result which is an event record for which no callback has been defined. This only occurs if you have explicitly set the eventmask for an object in such a way that it reports events for which there are no callbacks (this is described below). If `⎕WE` does return an event record, it will be a length 9 integer vector which has the same form as `⎕EV`.

Typically, therefore, your application will include a top-level event loop which (after initialisation) will call `⎕WE` with an argument of `¯1` or a window name. Any callbacks invoked which might take a long time processing can call `⎕WE` with an argument of 0 to ensure that events are handled promptly. Any modal dialogs you call will usually be processed by calling `⎕WE` with a character right argument. Be careful, however, not to get caught in a recursive loop which will create a large `)SI` stack and fill your workspace.

To help in debugging, `⎕WE` can be interrupted (using the Interrupt menu item, or the keyboard interrupt key Ctrl-Break or Command-Period, when the session window is active). Once you have fully debugged your application, you may wish to switch off interrupt handling using `⎕CONF` so that the user does not accidentally interrupt it.

⎕WI Windowing Interface

(Obsolescent - Use ⎕NEW and dot notation instead)

The older-style of interface from APL to the object-based programming environment (System Classes) is built around the system function `⎕WI`. This takes a left argument which is the name of the object, and a right argument which names the property or method you wish to access, and if appropriate the value you wish to assign to the property or pass to the method. When you create an object, you can optionally set properties at creation time, otherwise the system chooses defaults. The following examples show how this works:

Create a top-level object called Example, in this case a window with the standard Dialog border and appearance:

```
      'Example' ⎕WI 'New' 'Dialog'
```

Create an object called Lst1, of class List Box, on the window Example (the system chooses defaults for things like size and position):

```
      'Example.Lst1' ⎕WI 'New' 'List'
```

Set the size property of the list you just created to be 5 standard rows high and 30 columns wide (the object will immediately be re-sized):

```
      'Example.Lst1' ⎕WI 'size' 5 30
```

Create an object called Lst1, as above, but this time set the size property at the time you create it:

```
      'Example.Lst1' ⎕WI 'New' 'List' ('size' 5 30)
```

Put a set of choices (contained in the variable NAMES) into the list box by setting its list property:

```
      'Example.Lst1' ⎕WI 'list' NAMES
```

Read back the selection which the user has made by reading the value property of the list box:

```
      'Example.Lst1' ⎕WI 'value'
2
```

Syntax of ⎕WI

The exact syntax of ⎕WI for the three possible cases is as follows:

Setting a property:

```
     ObjectName ⎕WI PropertyName Value
```

ObjectName and PropertyName are character vectors, and Value can be any APL array valid for the particular property in question. The right argument to ⎕WI is thus a two-element nested vector. However, as a convenience ⎕WI accepts any length of nested array vector and treats the first element as the property name and the rest of the argument as the property value, so that the following two statements are both valid and do exactly the same thing:

```
      'Win1.But' ⎕WI 'where' (12 20 3 8)       ⍝ Two-element vector
             'Win1.But' ⎕WI 'where' 12 20 3 8                ⍝ Five-element vector
```

Reading a property:

```
      Value ← ObjectName ⎕WI PropertyName
```

ObjectName and PropertyName are character vectors as before, and the current value for the property is returned as the explicit result of ⎕WI. For example:

```
      'Win1.But' ⎕WI 'where'
12 20 3 8
      'Win1.But' ⎕WI 'caption'
Cancel
```

Invoking a method:

```
      ObjectName ⎕WI MethodName {Argument}
```

ObjectName and MethodName are character vectors, and Argument is an optional argument to the method. It can be any APL array valid for the particular method in question. Again, `⎕WI` accepts any length of nested array vector and treats the first element as the method name and the rest as the argument to the method.

```
      'Win1.Movie' ⎕WI 'Rewind'              ⍝ Method with no argument
      'Win1.But' ⎕WI 'New' 'Button'          ⍝ Argument is 'Button'
```

See the separate documentation on *System Classes and User-Interface Programming* for full details.

Console and Server versions of APLX

In Console versions of APLX, there is no GUI interface and therefore nearly all of `⎕WI` is not implemented. Only the Socket object, and a subset of the System object, are currently available. In Client-Server editions of APLX, `⎕WI` is fully implemented, and runs on the Client machine.

`⎕WSELF` Object Name

Just before your callback is called, the niladic system function `⎕WSELF` is set to contain the fully-qualified name of the object to which the callback refers. This helps you to use the same callback function for several related objects. For example, you might have a general handler called `LimitHit` for several Edit fields, and you might wish to put up an alert when this is run (i.e. when the user tries to enter more characters than are allowed). This is how you might write the `LimitHit` function to cater for several different Edit objects:

```
       ∇LimitHit;max;msg
    [1]  ⍝ Put up alert if limit reached in Edit field
    [2]  max←⎕WSELF ⎕WI 'Limit'
    [3]  msg←'The maximum number of characters',⎕R
    [4]  msg←msg,'you can enter in this field is ',⍕max
    [5]  ALERT msg
       ∇
```

The value returned by `⎕WSELF` outside callback functions is not reliable. In addition, if the object has been deleted in between the event occurring and your callback running, `⎕WSELF` will give an empty vector.

⎕WSSIZE Size of Workspace

The niladic system function ⎕WSSIZE returns the total size of the workspace (including both the used and free area) in bytes.

See also ⎕WA which returns the unused area only.

⎕XML Convert to/from XML

Extensible Markup Language (XML) is a widely used standard for storing data in a text format that many different programs can access. It combines the actual data with 'mark-up' which indicates how the data should be interpreted.

The ⎕XML system function can be used to extract data from XML format into an APL array, and to generate XML from an APL array. The direction of conversion is determined by the type of the right argument. (See also the ⎕IMPORT and ⎕EXPORT functions, which allow data to be transferred to/from XML files in a single step.)

An Example of XML format

A full description of XML is beyond the scope of this document. However, the following simple but complete XML example demonstrates some of the main features:

```
<?xml version="1.0" encoding="utf-8"?>
<sales>
    <!-- Sales by month -->
    <month>January
        <item>
            <name>Ice Cream</name>
            <amount currency="dollars">25.10</amount>
       </item>
       <item>
            <name>Fizzy Drinks</name>
            <amount currency="dollars">360.92</amount>
       </item>
    </month>
    <month>February
        <item>
            <name>Ice Cream</name>
            <amount currency="dollars">5.02</amount>
       </item>
       <item>
            <name>Fizzy Drinks</name>
            <amount currency="dollars">403.16</amount>
       </item>
    </month>
</sales>
```

The first line specifies the XML version used, and the third line ("Sales by month") is a comment. The remainder of the document consists of **elements** which contain the data. Each element begins with a **start tag** and ends with a matching **end tag**, for example:

```
    <name>...</name>
```

Element tag names are case-sensitive.

An element may contain data, or other elements nested within it, or both. In addition the start tag may include one or more **attributes** specifying how the data is to be interpreted. Each attribute is a pair of the form *name="value"*, for example:

```
    <amount currency="dollars">25.10</amount>
```

An empty element which contains no data and no other elements nested within it can be written as:

```
    <name/>
```

Within an XML document there is usually no significance in the amount of white space used, for example the number of spaces used to indent an element or the positions of line breaks. The following is valid in XML:

```
<item><name>Ice Cream</name><amount currency="dollars">25.10</amount></item>
```

Converting XML Data to an APL Array

Syntax: `R←[options] ⎕XML CHRVEC`

The right argument is a character vector (with embedded carriage returns and/or line feeds) containing the XML text to be converted. The optional left argument gives some control over the conversion process and is discussed below.

The result is an N-row, 5-column matrix containing a flattened representation of the XML data. Each element in the XML data will produce one row in the result. The columns are as follows:

Column 1:	An integer indicating the depth of nesting of the element. A value of 0 is used for the outer-most nesting level, with deeper nesting being indicated by higher numbers.
Column 2:	The element name as specified in the start tag.
Column 3:	The element data as a character vector
Column 4:	An M-row, 2-column nested matrix containing any attribute name/value pairs. Each item in the matrix is a character vector. If the element has no attributes, this matrix will have 0 rows.
Column 5:	A code to help interpret the type of data the row contains (See below)

For example, when presented with the XML sample listed above the array produced is as follows:

```
      ⎕XML xml_data
0 sales                                 3
1 month                                 7
2       January                         4
2 item                                  3
3 name   Ice Cream                      5
3 amount 25.10         currency dollars 5
2 item                                  3
3 name   Fizzy Drinks                   5
3 amount 360.92        currency dollars 5
1 month                                 7
2       February                        4
2 item                                  3
3 name   Ice Cream                      5
3 amount 5.02          currency dollars 5
2 item                                  3
3 name   Fizzy Drinks                   5
3 amount 403.16        currency dollars 5
```

```
      ⎕DISPLAY ⎕XML xml_data
┌→─────────────────────────────────────────────────────┐
↓   ┌→────┐  ┌⊖┐            ┌→────────┐                │
│ 0 │sales│  │ │            ⌽ ┌⊖┐ ┌⊖┐ │              3 │
│   └─────┘  └─┘            │ │ │ │ │ │                │
│                           │ └─┘ └─┘ │                │
│                           └∊────────┘                │
│   ┌→────┐  ┌⊖┐            ┌→────────┐                │
│ 1 │month│  │ │            ⌽ ┌⊖┐ ┌⊖┐ │              7 │
│   └─────┘  └─┘            │ │ │ │ │ │                │
│                           │ └─┘ └─┘ │                │
│                           └∊────────┘                │
│   ┌⊖┐      ┌→──────┐      ┌→────────┐                │
│ 2 │ │      │January│      ⌽ ┌⊖┐ ┌⊖┐ │              4 │
│   └─┘      └───────┘      │ │ │ │ │ │                │
│                           │ └─┘ └─┘ │                │
│                           └∊────────┘                │
│   ┌→───┐   ┌⊖┐            ┌→────────┐                │
│ 2 │item│   │ │            ⌽ ┌⊖┐ ┌⊖┐ │              3 │
│   └────┘   └─┘            │ │ │ │ │ │                │
│                           │ └─┘ └─┘ │                │
│                           └∊────────┘                │
│   ┌→───┐   ┌→────────┐    ┌→────────┐                │
│ 3 │name│   │Ice Cream│    ⌽ ┌⊖┐ ┌⊖┐ │              5 │
│   └────┘   └─────────┘    │ │ │ │ │ │                │
│                           │ └─┘ └─┘ │                │
│                           └∊────────┘                │
│   ┌→─────┐ ┌→────┐        ┌→─────────────────────┐   │
│ 3 │amount│ │25.10│        ↓ ┌→───────┐ ┌→──────┐ │ 5 │
│   └──────┘ └─────┘        │ │currency│ │dollars│ │   │
│                           │ └────────┘ └───────┘ │   │
│                           └∊─────────────────────┘   │
│   ┌→───┐   ┌⊖┐            ┌→────────┐                │
│ 2 │item│   │ │            ⌽ ┌⊖┐ ┌⊖┐ │              3 │
│   └────┘   └─┘            │ │ │ │ │ │                │
│                           │ └─┘ └─┘ │                │
│                           └∊────────┘                │
│   ┌→───┐   ┌→───────────┐ ┌→────────┐                │
│ 3 │name│   │Fizzy Drinks│ ⌽ ┌⊖┐ ┌⊖┐ │              5 │
│   └────┘   └────────────┘ │ │ │ │ │ │                │
│                           │ └─┘ └─┘ │                │
│                           └∊────────┘                │
│                                                      │
│   ┌→─────┐  ┌→─────┐        ┌→─────────────────────┐     │
│ 3 │amount│  │360.92│        ↓ ┌→───────┐ ┌→──────┐ │   5 │
│   └──────┘  └──────┘        │ │currency│ │dollars│ │     │
│                             │ └────────┘ └───────┘ │     │
│                             └∊─────────────────────┘     │
│   ┌→────┐   ┌⊖┐             ┌→────────┐                  │
│ 1 │month│   │ │             ⌽ ┌⊖┐ ┌⊖┐ │              7   │
│   └─────┘   └─┘             │ │ │ │ │ │                  │
│                             │ └─┘ └─┘ │                  │
│                             └∊────────┘                  │
│   ┌⊖┐       ┌→───────┐      ┌→────────┐                  │
│ 2 │ │       │February│      ⌽ ┌⊖┐ ┌⊖┐ │              4   │
│   └─┘       └────────┘      │ │ │ │ │ │                  │
│                             │ └─┘ └─┘ │                  │
│                             └∊────────┘                  │
│   ┌→───┐    ┌⊖┐             ┌→────────┐                  │
│ 2 │item│    │ │             ⌽ ┌⊖┐ ┌⊖┐ │              3   │
│   └────┘    └─┘             │ │ │ │ │ │                  │
│                             │ └─┘ └─┘ │                  │
│                             └∊────────┘                  │
│   ┌→───┐    ┌→────────┐     ┌→────────┐                  │
│ 3 │name│    │Ice Cream│     ⌽ ┌⊖┐ ┌⊖┐ │              5   │
│   └────┘    └─────────┘     │ │ │ │ │ │                  │
│                             │ └─┘ └─┘ │                  │
│                             └∊────────┘                  │
│   ┌→─────┐  ┌→───┐          ┌→─────────────────────┐     │
│ 3 │amount│  │5.02│          ↓ ┌→───────┐ ┌→──────┐ │   5 │
│   └──────┘  └────┘          │ │currency│ │dollars│ │     │
│                             │ └────────┘ └───────┘ │     │
│                             └∊─────────────────────┘     │
│   ┌→───┐    ┌⊖┐             ┌→────────┐                  │
│ 2 │item│    │ │             ⌽ ┌⊖┐ ┌⊖┐ │              3   │
│   └────┘    └─┘             │ │ │ │ │ │                  │
│                             │ └─┘ └─┘ │                  │
│                             └∊────────┘                  │
│   ┌→───┐    ┌→───────────┐  ┌→────────┐                  │
│ 3 │name│    │Fizzy Drinks│  ⌽ ┌⊖┐ ┌⊖┐ │              5   │
│   └────┘    └────────────┘  │ │ │ │ │ │                  │
│                             │ └─┘ └─┘ │                  │
│                             └∊────────┘                  │
│   ┌→─────┐  ┌→─────┐        ┌→─────────────────────┐     │
│ 3 │amount│  │403.16│        ↓ ┌→───────┐ ┌→──────┐ │   5 │
│   └──────┘  └──────┘        │ │currency│ │dollars│ │     │
│                             │ └────────┘ └───────┘ │     │
│                             └∊─────────────────────┘     │
└∊─────────────────────────────────────────────────────────┘
```

Options for converting XML to an APL array

The conversion from XML to an APL array described above can be controlled by an optional left argument which consists of one or more keyword/value pairs, for example:

```
      R←('markup' 'preserve') ('whitespace' 'preserve') ⎕XML xml_data
```

The supported keywords are:

- **'markup'**: possible values **'preserve'** and **'strip'**

  By default ⎕XML strips out all XML statements which are not data elements. In the example above, the following two lines were stripped out:

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <!-- Sales by month -->
  ```

  The first one is a processing instruction and the second is a comment. Neither of them contain any data.

  However it is sometimes necessary to have access to the complete content of the XML document, for example if you need to do special processing of entity declarations like *<!DOCTYPE>* and *<!ELEMENT>*. By specifying 'markup' 'preserve' you can tell ⎕XML that all elements in the XML should produce corresponding rows in the APL array.

- **'whitespace'**: possible values **'preserve'**, **'strip'** and **'trim'**

  By default ⎕XML strips all leading and trailing white space from element data, and compresses runs of white space within the data into a single space. You can modify this behaviour by specifying that all white space should be preserved, or that only leading and trailing spaces which enclose the data should be trimmed.

  There is one exception to this behaviour. If an XML element has the attribute **xml:space="preserve"** then white space is always retained.

- **'unknown-entity'**: possible values **'preserve'** and **'replace'**

  XML data can include a number of predeclared entity references like "&amp;" to represent the "&" character, or "&amp#9017;" for Unicode character 9017. These are always converted by ⎕XML to their single-character forms.

  However, additional entity references can be declared in the XML Document Type Definition (DTD) and then used in the text. APLX does not currently parse the DTD and so does not know how to substitute for these references. Instead, the default behaviour of ⎕XML is to substitute the character specified by ⎕MC (by default, a question mark).

  This behaviour can be changed so that ⎕XML preserves unknown entity references, in which case they are passed to the APL array. The leading '&' is converted to an <ESC> character (⎕TCESC) so that the entity reference can be detected by the APL program, e.g. "&ref;" becomes "<ESC>ref"

Type code returned by ⎕XML

The fifth column of the array produced by ⎕XML contains a type code which can be used to interpret the row. Its value depends on whether the XML element has any children.

Possible children can be of the following types. (Note that if markup is stripped only the first of these types can occur in the final result).

- A nested XML element

```
<Parent>
    <Child>...</Child>
</Parent>
```

- A nested XML comment

```
<Parent>
    <!--Comment-->
</Parent>
```

- A nested XML Processing Instruction

```
<Parent>
    <?Processing instruction?>
</Parent>
```

- Other nested XML markup

```
<Parent>
    <!ELEMENT name (#PCDATA)>
</Parent>
```

(a) If the XML element has children its type code is formed from a sum of the following values, reflecting the types of children found on subsequent rows:

1	Element has a tag (in column 2) (Always true)
2	Element contains nested child element
4	Element contains data as well as nested items
8	Element contains nested XML markup
16	Element contains nested XML comment
32	Element contains nested XML Processing Instruction

For example, the element `<Weight>` in the following example has a type code of 21 (1 + 16 + 4) when markup and comments are preserved:

```
<Weight>
    <!-- All weights approximate-->
    100
</Weight>
```

Notice that an XML element with children always has a tag name in column 2. It never has any data in column 3 : all the data is returned in subsequent rows.

(b) The following type codes are used for XML elements which don't have any children:

1	Element is an empty XML tag, e.g. *<empty/>*. The tag name in returned in column 2, and column 3 is blank.
4	Row is data for parent (See below). The data is returned in column 3, and column 2 is blank.

5 Element has an XML tag and data, e.g. *<Tag>Data</Tag>*
The tag name is returned in column 2 and the data in column 3.

8 Element is unprocessed XML markup, e.g. *<!ELEMENT name (#PCDATA)>*.
The markup is returned in column 2, and column 3 is blank.

16 Element is XML comment, e.g. *<!--Comment-->*.
The comment is returned in column 2, and column 3 is blank.

32 Element is XML Processing Instruction, e.g. *<?xml version="1.0" encoding="utf-8"?>*.
The processing instruction is returned in column 2, and column 3 is blank.

The following example illustrates how the codes are used:

```
<Tag1>Text
      <Tag2>
      <Tag3>Text</Tag3>
      </Tag2>
      More Text
</Tag1>
```

When converted by `⎕XML` this will produce the following array

```
    ⎕DISPLAY ⎕XML xml_data
┌→───────────────────────────────────┐
↓   ┌→───┐ ┌⊖┐         ┌→────────┐   │
│ 0 │Tag1│ │ │         ⌽ ┌⊖┐ ┌⊖┐ │ 7 │
│   └────┘ └─┘         │ │ │ │ │ │   │
│                      │ └─┘ └─┘ │   │
│                      └∊────────┘   │
│                                    │
│   ┌⊖┐    ┌→───┐      ┌→────────┐   │
│ 1 │ │    │Text│      ⌽ ┌⊖┐ ┌⊖┐ │ 4 │
│   └─┘    └────┘      │ │ │ │ │ │   │
│                      │ └─┘ └─┘ │   │
│                      └∊────────┘   │
│                                    │
│   ┌→───┐ ┌⊖┐         ┌→────────┐   │
│ 1 │Tag2│ │ │         ⌽ ┌⊖┐ ┌⊖┐ │ 3 │
│   └────┘ └─┘         │ │ │ │ │ │   │
│                      │ └─┘ └─┘ │   │
│                      └∊────────┘   │
│                                    │
│   ┌→───┐ ┌→───┐      ┌→────────┐   │
│ 2 │Tag3│ │Text│      ⌽ ┌⊖┐ ┌⊖┐ │ 5 │
│   └────┘ └────┘      │ │ │ │ │ │   │
│                      │ └─┘ └─┘ │   │
│                      └∊────────┘   │
│                                    │
│   ┌⊖┐    ┌→────────┐ ┌→────────┐   │
│ 1 │ │    │More Text│ ⌽ ┌⊖┐ ┌⊖┐ │ 4 │
│   └─┘    └─────────┘ │ │ │ │ │ │   │
│                      │ └─┘ └─┘ │   │
│                      └∊────────┘   │
│                                    │
└∊───────────────────────────────────┘
```

Creating XML Data from an APL Array

Syntax: `R←[options] ⎕XML NSTMAT`

When presented with an array of APL data, `⎕XML`will convert it to XML representation. The result is a character vector with embedded line-feed characters.

The right argument must be a nested matrix with one row for each XML element, and between 3 and 5 columns as follows

Column 1:	An integer indicating the depth of nesting of the element. A value of 0 is used for the outer-most nesting level, with deeper nesting being indicated by higher numbers.
Column 2:	The element name to use for the start tag.
Column 3:	The element data (see below)
Column 4:	(Optional) An M-row, 2-column nested matrix containing any attribute name/value pairs. Each item in the matrix is a character vector. If the element has no attributes you can specify a 0-row matrix, or a pair of empty character vectors. If none of the elements have any attributes you can omit column 4 completely.
Column 5:	(Optional) An integer type code (ignored). This column is only used to facilitate round-trip conversions from XML to APL and back again.

The data specified in Column 3 will usually be a character vector or scalar. However, as a convenience `⎕XML` also allows you to specify numeric values. These are formatted as character data before copying to the XML result. Numeric values are also allowed for attribute values (but not names).

Example:

```
      array←1 4⍴0 '?xml version="1.0" encoding="utf-8"?' '' ('' '')
      array←array⍪0 'Person' '' ('' '')
      array←array⍪1 'Name' '' ('order' 'western')
      array←array⍪2 'FirstName' 'Fred' ('' '')
      array←array⍪2 'LastName' 'Smith' ('' '')
      array←array⍪1 'DateOfBirth' '' ('' '')
      array←array⍪2 'Year' 1943 ('' '')
      array←array⍪2 'Month' 12 ('' '')
      array←array⍪2 'Day' 17 ('' '')
      XML←⎕XML array
      ⎕SS XML ⎕L ⎕R    ⍝ Convert line feeds to carriage return for display
<?xml version="1.0" encoding="utf-8"?>
<Person>
    <Name order="western">
        <FirstName>Fred</FirstName>
        <LastName>Smith</LastName>
    </Name>
    <DateOfBirth>
        <Year>1943</Year>
        <Month>12</Month>
        <Day>17</Day>
    </DateOfBirth>
</Person>
```

The conversion process can be controlled by an optional left argument, for example:

```
      R←('whitespace' 'preserve') ⎕XML apl_data
```

The only supported option is:

- **'whitespace'**: possible values **'preserve'**, **'strip'** and **'trim'**

  By default `⎕XML` strips all leading and trailing white space from element data, and compresses runs of white space within the data into a single space. The XML text produced then has spaces and line-feed characters added to format it for readability. For example elements are indented to reflect their degree of nesting.

  You can modify this behaviour by specifying that all white space should be preserved, or that only leading and trailing spaces which enclose the data should be trimmed. The option of preserving all white space is most useful when you are re-creating XML data from an APL array which was itself produced by `⎕XML` with spaces preserved.

  If you specify the attribute **xml:space** with the value **preserve** on any row, all white space is retained in the corresponding XML element.

Adding the XML Prologue

To be valid, an XML file must start with a line containing an XML prologue, e.g.

```
<?xml version="1.0" encoding="utf-8"?>
```

Note that `⎕XML` does not add the prologue automatically. To ensure that the XML is valid you must do one of two things:

(a) Make sure that the first row of the array used to generate the XML contains a valid prologue, as in the example above, or

(b) Prepend the prologue after the XML has been generated:

```
      XML←⎕XML 1 'Name' 'Fred Smith'
      XML←'<?xml version="1.0" encoding="utf-8"?>',⎕L,XML
```

If you create an XML file using `⎕EXPORT`, APLX will automatically add the prologue if it is missing from the array.

Acknowledgment

This work is based on the original design concepts and implementation by Mark E. Johns, and has been designed in cooperation with Dyalog Ltd

Section 8: System Methods

⎕BASE Base (parent) class

Implemented for Internal and External classes. Not implemented for System classes.

Syntax:

```
    classref ← objref.⎕BASE
    classref ← classref.⎕BASE
    classref ← ⎕BASE        (Within user-defined method, same as ⎕THIS.⎕BASE)
```

The niladic system method ⎕BASE returns a reference to the base (parent class) of either an object, or a class. The result is always a class reference, or the Null object if there is no parent. For example, if class `Car` inherits from class `Vehicle`:

```
      )CLASSES
Car     Vehicle
      M←⎕NEW 'Car'
      M
[Car]
      M.⎕CLASSREF
{Car}
      M.⎕BASE
{Vehicle}

      M.⎕NL 2        ⍝ Child has properties of its own, plus those of parent
Marque
Owner
HasEngine
IsPublicTransport
Passengers
TopSpeed
Wheels

      M.⎕BASE.⎕NL 2  ⍝ Parent has fewer properties than the child
HasEngine
IsPublicTransport
Passengers
TopSpeed
Wheels
```

A common use for ⎕BASE is calling the parent's version of a method within the child's version of the same method. Typically, the need for this arises when the child class needs to do some extra processing in addition to what the parent does. You have full control over this; the child method (which in APLX always overrides the parent's method) can call the parent's version either at the beginning, in the middle, or at the end of its own version of the method - or indeed, not call it at all.

⎕BASE can also be used with External classes (but not System classes). For example, in the .Net System.Windows.Forms framework, the Button class inherits from the ButtonBase class, which in turn inherits from the Control class:

```
      BT←'.net' ⎕NEW 'System.Windows.Forms.Button'
      BT.⎕BASE
{.net:ButtonBase}
      BT.⎕BASE.⎕CLASSNAME
.net:System.Windows.Forms.ButtonBase
      BT.⎕BASE.⎕BASE.⎕CLASSNAME
.net:System.Windows.Forms.Control
```

Similarly, in Ruby, the DateTime Class inherits from the Date class, which in turn inherits from the fundamental Object class (from which all classes in Ruby are derived). This example shows how this looks from within APL:

```
      'ruby' ⎕SETUP 'require' 'date'
      DT←'ruby' ⎕NEW 'DateTime'
      DT                               ⍝ Reference to a DateTime object
[ruby:DateTime]
      DT.⎕BASE                         ⍝ Reference to parent class (Date)
{ruby:Date}
      DT.⎕BASE.⎕CLASSNAME
ruby:Date
      DT.⎕BASE.⎕BASE.⎕CLASSNAME
ruby:Object
      DT.⎕BASE.⎕BASE.⎕BASE
[NULL OBJECT]
```

⎕CHILDREN Child classes

Implemented for Internal (user-defined)classes only

Syntax:

```
    classrefs ← classref.⎕CHILDREN
```

The niladic system method ⎕CHILDREN returns a vector of references to the children of a given class, i.e. the classes which inherit from that class. In this example, classes `Color_Point` and `Grayscale_Point` both inherit from class `Point`:

```
      )CLASSES
Color_Point  Grayscale_Point  Point
      Point.⎕CHILDREN
{Color_Point} {Grayscale_Point}
      ⍴Point.⎕CHILDREN
2
```

⎕CLASSNAME Name of class

Implemented for Internal, External and System classes.

Syntax:

```
    name ← objref.⎕CLASSNAME
    name ← classref.⎕CLASSNAME
    name ← ⎕CLASSNAME         (Within user-defined method, same as ⎕THIS.⎕CLASSNAME)
```

The niladic system method ⎕CLASSNAME returns the fully-qualified name of the class of an object, as a character vector. It can also be used with a class reference rather than an object reference:

```
      PT←⎕NEW Point
      PT.⎕CLASSNAME
Point
      ⎕DISPLAY PT.⎕CLASSNAME
┌→────┐
│Point│
└─────┘
      Point.⎕CLASSNAME
Point
```

If the class is an External or System class, the class name is preceded by the architecture name and a colon:

```
      DT←'.net' ⎕NEW 'System.DateTime' 2007 5 30
      DT
[.net:DateTime]
      DT.⎕CLASSNAME
.net:System.DateTime
```

⎕CLASSREF Reference to object's class

Implemented for Internal and External classes. Not implemented for System classes.

Syntax:

```
    classref ← objref.⎕CLASSREF
    classref ← classref.⎕CLASSREF
    classref ← ⎕CLASSREF         (Within user-defined method, same as ⎕THIS.⎕CLASSREF)
```

The niladic system method `⎕CLASSREF` returns a reference to the class of an object, as a scalar. It can also be used with a class reference rather than an object reference, in which case it returns the class reference unchanged.

It can be used with both internal and external classes (but not system classes). However, it is an optional feature for external classes; it is implemented for .Net, Java and Ruby, but is not implemented for R.

```
      DT←'.net' ⎕NEW 'System.DateTime' 2007 5 30
      DT                   ⍝ DT references an object of the .Net DateTime class
[.net:DateTime]
      CL←DT.⎕CLASSREF
      CL                   ⍝ CL is a reference to the DateTime class itself
{.net:DateTime}
```

A class reference obtained in this way can be used to create a new object of the same class, by using the class reference as the right argument of `⎕NEW` (rather than a character vector). In this case, it is not necessary to specify the architecture as the left argument of `⎕NEW`:

```
      DT2←⎕NEW CL 2007 9 20 12 13 44  ⍝ CL can be used to make a new object
      DT2                             ⍝ of the same class as DT
[.net:DateTime]
      DT2.⎕DS
20/09/2007 12:13:44
```

`⎕CLONE` Create copies of object

Implemented for Internal and External classes. Not implemented for System classes.

Syntax:

```
    objrefs ← objref.⎕CLONE N
    objrefs ← ⎕CLONE N
```
(Within user-defined method, same as `⎕THIS.⎕CLONE N`*)*

The monadic system method `⎕CLONE` creates new copies of an object, and returns a vector of references to the new objects. The right argument N is the number of copies required, as a positive integer (or 0). The new objects have their properties initially set to the same values as the original.

In this example, we create a simple class `Point`, and create an instance of it. The instance is then cloned four times. Note that the four copies are independent objects; changing a property of one of them does not affect the others:

```
      'Point' ⎕IC ⎕BOX 'X Y Z'
1 1 1
      PT←⎕NEW 'Point'
      PT.X←34 ⋄ PT.Y←23 ⋄ PT.Z←12
      VEC←PT.⎕CLONE 4
      VEC
```

```
[Point] [Point] [Point] [Point]
      VEC.X
34 34 34 34
      VEC[1].X←11
      VEC.X
11 34 34 34
```

Contrast this with the following example, in which `VEC2` contains four references to the *same* object:

```
      VEC2←4⍴PT
      VEC2.X
34 34 34 34
      VEC2[1].X←11
      VEC2.X
11 11 11 11
```

When cloning internal objects, if the properties of the original object contain further object references these sub-objects are not cloned; instead, both the original and cloned objects will point to the same original sub-objects. This is known as a 'shallow' copy operation.

If an object contains file handles or database references, cloning it in this way may not make sense. On the other hand, for simple objects it can be a very useful way of rapidly creating new instances with an initial set of known properties.

`⎕CLONE` can be used with External classes (but not System classes). However, not all classes implement the operation, and the exact way in which it works may vary according to the architecture and the class:

.Net

In the .Net architecture, `⎕CLONE` is mapped to the `Clone` method of the `ICloneable` interface. However, many .Net classes do not implement this.

Java

For Java classes, `⎕CLONE` is mapped to the `Clone` method of the Java base object, which may be over-ridden for a particular Java class.

In this example, we create an instance of the Java `Date` class (which defaults to the current date/time), clone 12 copies of it, and set the month of each copy to a different value:

```
      date←'java' ⎕NEW 'java.util.Date'
      date
[java:Date]
      dateList←date.⎕CLONE 12
      dateList.setMonth ((⍳12)-⎕IO)
      12 1⍴dateList.⎕DS
 Wed Jan 10 14:33:12 GMT 2007
 Sat Feb 10 14:33:12 GMT 2007
 Sat Mar 10 14:33:12 GMT 2007
 Tue Apr 10 14:33:12 BST 2007
 Thu May 10 14:33:12 BST 2007
 Sun Jun 10 14:33:12 BST 2007
 Tue Jul 10 14:33:12 BST 2007
 Fri Aug 10 14:33:12 BST 2007
 Mon Sep 10 14:33:12 BST 2007
 Wed Oct 10 14:33:12 BST 2007
 Sat Nov 10 14:33:12 GMT 2007
 Mon Dec 10 14:33:12 GMT 2007
```

R

⎕CLONE is mapped to the `duplicate` function of R. It can thus be used to create multiple independent copies of R arrays, complex numbers, data frames, factors and so on.

```
      c←'r' ⎕new 'complex' 2 3
      c
[r:complex]
      c.⎕ds
2+3i
      t←c.⎕clone 5
      t.⎕ds
 2+3i 2+3i 2+3i 2+3i 2+3i
      t
[r:complex] [r:complex] [r:complex] [r:complex] [r:complex]
```

Ruby

⎕CLONE is mapped to the `dup` method of the Ruby base object. Again, this may be over-ridden for a particular Ruby class.

In this example, we create a Ruby array (referenced by `A`), and place three strings in it. We then make a clone of it (referenced by `B`), which creates an independent copy of the array. Therefore, when we add a fourth element to the array referenced by `A`, the copy is unchanged:

```
      A←'ruby' ⎕NEW 'Array'
      dummy←A.push 'First'
      dummy←A.push 'Second'
      dummy←A.push 'Third'
      A.length
3
      A.⎕VAL
 First Second Third
      B←A.⎕CLONE 1
      B.length
3
      B.⎕VAL
 First Second Third
      dummy←A.push 'Fourth'
      A.length
4
      A.⎕VAL
 First Second Third Fourth
      B.length
3
      B.⎕VAL
 First Second Third
```

⎕DESC Describe public members

Implemented for Internal, External and System classes.

Syntax:

```
    namelist ← objref.⎕DESC N
    namelist ← classref.⎕DESC N
    namelist ← ⎕DESC N          (Within user-defined method, same as ⎕THIS.⎕DESC N)
```

⎕DESC is similar to ⎕NL in that it returns a list of the public members of a class as a character matrix. However, it also provides additional information about the parameters of each method, including type information if appropriate. It is especially useful for strongly-typed architectures such as Java and .Net. It can be used as a method either of a class, or of an instance of a class.

The right argument is a scalar or vector which indicates which types of class member should be included in the result, using the same codes as ⎕NL:

Code	***Type of member***
2	Properties
3	Function methods
4	Operator methods
8	Events
10	Constructors

For Internal classes, ⎕DESC returns just the names for properties, and the function header (excluding the localised names) for function and operator methods:

```
Point {
  X
  Y

  ∇R←MAG
    R←((X*2)+(Y*2))*0.5
  ∇
}
      Point.⎕DESC 2       ⍝ Applied to a class
X
Y
      PT←⎕NEW 'Point'
      PT.⎕DESC 2 3        ⍝ Applied to an object (properties and methods)
R←MAG
X
Y
```

For External classes in Java or .Net, `⎕DESC` shows the types of properties, method parameters, and results. Because a given method name may be *overloaded* (i.e. exist in different forms, according to the number and types of parameters provided to it), a given method name may appear more than once:

```
      NetPT←'.net' ⎕NEW 'System.Drawing.Point'

      NetPT.⎕DESC 2      ⍝ Properties
System.Drawing.Point Empty
Boolean IsEmpty
Int32 X
Int32 Y

      NetPT.⎕DESC 10     ⍝ Constructors (overloaded)
Void Point(Int32, Int32)
Void Point(System.Drawing.Size)
Void Point(Int32)

      NetPT.⎕DESC 3      ⍝ Methods
System.Drawing.Point Add(System.Drawing.Point, System.Drawing.Size)
System.Drawing.Point Ceiling(System.Drawing.PointF)
Boolean Equals(System.Object)
Boolean get_IsEmpty()
Int32 get_X()
Int32 get_Y()
Int32 GetHashCode()
System.Type GetType()
Void Offset(Int32, Int32)
Void Offset(System.Drawing.Point)
System.Drawing.Point op_Addition(System.Drawing.Point, System.Drawing.Size)
Boolean op_Equality(System.Drawing.Point, System.Drawing.Point)
System.Drawing.Size op_Explicit(System.Drawing.Point)
System.Drawing.PointF op_Implicit(System.Drawing.Point)
Boolean op_Inequality(System.Drawing.Point, System.Drawing.Point)
System.Drawing.Point op_Subtraction(System.Drawing.Point, System.Drawing.Size)
System.Drawing.Point Round(System.Drawing.PointF)
Void set_X(Int32)
Void set_Y(Int32)
System.Drawing.Point Subtract(System.Drawing.Point, System.Drawing.Size)
System.String ToString()
System.Drawing.Point Truncate(System.Drawing.PointF)
```

For the R interface, `⎕DESC` can be used to produce a list of all the R functions which are available in the currently-loaded packages, together with the parameters and default values (except for primitive functions). This will usually be a large list (several thousand functions):

```
      r←'r' ⎕new 'r'
      fns←r.⎕desc 3
      ⍴fns
2037 117
      fns[1445+⍳5;]
pwilcox (q, m, n, lower.tail = TRUE, log.p = FALSE)
q (save = "default", status = 0, runLast = TRUE)
qbeta (p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qbinom (p, size, prob, lower.tail = TRUE, log.p = FALSE)
qbirthday (prob = 0.5, classes = 365, coincident = 2)
```

⎕DF Set display form

Implemented for Internal, External and System classes.

Syntax:

```
    old ← objref.⎕DF string
    old ← ⎕DF string
```
(Within user-defined method, same as `⎕THIS.⎕DF string`*)*

Normally, when you display a variable or temporary result which contains an object reference, APLX displays the reference in a standard form (the class name enclosed in square brackets). The monadic system method `⎕DF` allows you to substitute your own string to replace this. The right argument should be a character vector (it will be truncated on display if it is longer than 40 characters). If the right argument is an empty vector, the default display format is restored.

The explicit result of `⎕DF` is the previous value.

For example, if you have a user-defined class called `Queue`:

```
      Q←⎕NEW 'Queue'
      Q
[Queue]
      Q.⎕DF 'Box Office Queue'

      Q
Box Office Queue

      Q.⎕DF ''          ⍝ Restore default format
Box Office Queue
      Q
[Queue]
```

`⎕DF` may be used within a Constructor, to set the default display of an object depending on the values used to create it.

You can use `⎕DF` to set the default display form of any object, including External and System objects:

```
      W←'⎕' ⎕NEW 'Window'
      W
[⎕:Window]
      W.⎕DF 'Main window'

      W
Main window
      2 3⍴W,⍳5
Main window 1 2
          3 4 5
```

The display string which you set using `⎕DF` applies to monadic format as well as display of expressions. See also `⎕DS` which returns a text form of an object.

⎕DS Display summary of object

Implemented for Internal, External and System classes.

Syntax:

```
    string ← objref.⎕DS
    string ← classref.⎕DS
    string ← ⎕DS
```
(Within user-defined method, same as `⎕THIS.⎕DS`*)*

`⎕DS` returns a character vector containing a summary of an object in human-readable form.

For Internal classes, `⎕DS` returns a summary of properties of the object, abbreviated if they are longer than around 50 characters. For example, if the class `Contact` has properties `Name`, `Address`, `Email`, and `Recnum`, the summary display might be as follows:

```
      A←⎕NEW Contact
      A.Name←'Alphonse T. Randall'
      A.Address←'The Manor House,Main Square,Moreton-under-Stockwood,Tenby'
      A.EMail←'Alphonse.Randall@themanor.com'
      A.Recnum←34973
      A.⎕DS
Name='Alphonse T. Randall', Address='The Manor House,Main Square,Moreton-under-S
      tockwoo...', EMail='Alphonse.Randall@themanor.com', Recnum=34973
```

For System classes, `⎕DS` returns the class name.

For External classes, `⎕DS` is mapped to whichever method in the target architecture is used for the default string form of an object. This is `ToString` for .Net. `to_s` for Ruby, `toString` for Java, and `format` for R.

In this example, we create (in each of three architectures) an object representing the same date/time, and use `⎕DS` to display the date/time in human-readable form:

```
      'ruby' ⎕SETUP 'require' 'Date'
      RubyDate←'ruby' ⎕NEW 'DateTime' 2004 12 13 8 34 03
      RubyDate.⎕DS
2004-12-13T08:34:03+00:00

      NetDate←'.net' ⎕NEW 'DateTime' 2004 12 13 8 34 03
      NetDate.⎕DS
13/12/2004 08:34:03

      JavaDate←'java' ⎕NEW 'java.util.Date' 2004 12 13 8 34 03
      JavaDate.⎕DS
Fri Jan 13 08:34:03 GMT 3905
```

```
      MixedObjects←RubyDate,NetDate,JavaDate
      MixedObjects
[ruby:DateTime] [.net:DateTime] [java:Date]
      MixedObjects.⎕DS
 2004-12-13T08:34:03+00:00 13/12/2004 08:34:03 Fri Jan 13 08:34:03 GMT 3905
      ⎕DISPLAY MixedObjects.⎕DS
┌→──────────────────────────────────────────────────────────────────────────┐
│ ┌→────────────────────────┐ ┌→──────────────────┐ ┌→──────────────────────────┐ │
│ │2004-12-13T08:34:03+00:00│ │13/12/2004 08:34:03│ │Fri Jan 13 08:34:03 GMT 3905│ │
│ └─────────────────────────┘ └───────────────────┘ └───────────────────────────┘ │
└∊──────────────────────────────────────────────────────────────────────────┘
```

The actual string which is displayed depends on the individual class. The style of the display may not always be consistent for a given architecture. For example, in the .Net framework, the `ToString` method sometimes displays the data associated with the class (as in the DateTime example above), sometimes just the object's class name, and sometimes a verbose summary of an object's properties:

```
      PT←'.net' ⎕NEW 'System.Drawing.Point'
      PT.⎕DS
{X=0,Y=0}

      FD←'.net' ⎕NEW 'System.Windows.Forms.FontDialog'
      FD.⎕DS
System.Windows.Forms.FontDialog,  Font: [Font: Name=Microsoft Sans Serif, Size=8
      .25, Units=3, GdiCharSet=0, GdiVerticalFont=False]

      HT←'.net' ⎕NEW 'System.Collections.Hashtable'
      HT.⎕DS
System.Collections.Hashtable
```

In this example, in R, we create an array of complex numbers and use `⎕DS` to format it:

```
      m←'r' ⎕new 'complex' (3 2⍴(1 2) (3 4) (5 6) (7 8) (9 10) (11 12))
      m
[r:matrix]
      m.⎕ds
 1+ 2i  3+ 4i
 5+ 6i  7+ 8i
 9+10i 11+12i
```

⎕HANDLE Handle to object

Implemented for Internal, External and System classes.

Syntax:

```
    integer ← objref.⎕HANDLE
    integer ← classref.⎕HANDLE
    integer ← ⎕HANDLE              (Within user-defined method, same as ⎕THIS.⎕HANDLE)
```

For External and System classes, `⎕HANDLE` returns an integer containing a reference to a class or object, as seen by the external sub-system. For Internal classes it always returns zero. It is used for advanced low-level programming.

Depending on the external architecture, this handle can be used when you need to pass an object reference to custom low-level (non-APL) code, or use `⎕NA` to call an external library routine. The meaning of the value returned is as follows:

Architecture	***Value returned***
.Net	Integer representing a `GCHandle` for the object
Java	Result of `NewGlobalRef` call to the JVM
Ruby	The Ruby `VALUE` of the object, cast to an integer

⎕MEMBERS Details of class members

Currently implemented for Internal classes only.

Syntax:

```
    nested_matrix ← objref.⎕MEMBERS
    nested_matrix ← classref.⎕MEMBERS
    nested_matrix ← ⎕MEMBERS       (Within user-defined method, same as ⎕THIS.⎕MEMBERS)
```

The result of `⎕MEMBERS` is a 6-column matrix, describing the members (properties, methods, and constructors) for the class of an object, or for the class itself if it is used on a class reference. The six columns are as follows (in index origin 1):

[;1] Member name (character vector)

[;2] Class name in which it is defined (character vector)

[;3] Member type (using `⎕NC` codes: 2=var, 3=function, 4=operator, 10=constructor)

[;4] Scope: 0=public, 1=private

[;5] Access code: 2 if undefined property, 1 if read-only property, else 0

[;6] Summary of Method (first line of comment) if available, else empty (character vector)

⎕MIXIN Mix another class into object

Implemented for Internal classes only (but right argument can be External or System class).

Syntax:

```
    {arch} objref.⎕MIXIN Class Arg1 Arg2..
    mixin_ref ← {arch} objref.⎕MIXIN Class Arg1 Arg2..
    {arch} ⎕MIXIN Class Arg1 Arg2..   (Within user-defined method, same as ⎕THIS.⎕MIXIN)
    mixin_ref ← {arch} ⎕MIXIN Class Arg1 Arg2..    "
```

The System Method ⎕MIXIN allow you to mix another class into an object. This has the effect of adding the properties and methods of the mixin to the main object. The mixin can be another internal (APL class), or a system class (such as Window), or an external class (.Net, Java, etc).

⎕MIXIN has a similar syntax to ⎕NEW; the right argument is the class reference (or name, as a text vector), followed by any arguments to the constructor for the class you are mixing-in. The left argument can be omitted if you are mixing-in an APL class, otherwise it defines the architecture for the mix-in. For example:

```
      ⍝ Mix in an APL class, no arguments to constructor
      inv.⎕mixin Fax

      ⍝ Mix in a Java class, no arguments to constructor
      'java' inv.⎕mixin 'java.util.Date'

      ⍝ Mix in a .Net class, with arguments to constructor
      '.net' inv.⎕mixin 'DateTime' 2004 5 6

      inv.⎕mixins
[Fax] [java:Date] [.net:DateTime]
```

The explicit result of ⎕MIXIN is the underlying object reference which has been mixed in to the object, but with display potential switched off. (In other words it is a 'shy' or non-printing result). You can assign this to a variable or property of your APL class, and use this to call the underlying object directly:

```
      jd←'java' inv.⎕mixin 'java.util.Date'
      jd.⎕classname
java:java.util.Date
```

See the section on Mixins for more information.

⎕MIXINS Return list of mixins

Implemented for Internal classes only.

Syntax:

```
    mixin_refs ← objref.⎕MIXINS
    mixin_refs ← ⎕MIXINS
```
(Within user-defined method, same as `⎕THIS.⎕MIXINS`*)*

The system method `⎕MIXINS` returns a vector of references to any mixins which have been added to an object using `⎕MIXIN`, in the order in which they were added:

```
      a←⎕new class1
      a.⎕mixin class2
      a.⎕mixins
[class2]
      '.net' a.⎕mixin 'DateTime' 2004 5 6
      a.⎕mixins
[class2] [.net:DateTime]
      ⍴a.⎕mixins
2
```

If there are no mixins for the object, it returns an empty vector.

The references returned by `⎕MIXIN` can typically be used to access methods or properties specific to the mixin. For example, if a method in the main class has the same name as a method in a mixin, the reference can be used to access the version in the mixin:

```
      a.⎕mixins[2].⎕classname
.net:System.DateTime
      a.⎕classname
class1
```

See the section on Mixins for more information.

⎕NL Names of public members

Syntax:

```
    namelist ← objref.⎕NL N
    namelist ← classref.⎕NL N
```

⎕NL returns a list of the public members of a class as a character matrix. (See also ⎕DESC which provides additional information about the parameters of each method). It can be used as a method either of a class, or of an instance of a class. However, exceptionally, if it is used as a standalone system function within a user-defined method, it is NOT equivalent to ⎕THIS.⎕NL. This is because in this case, the traditional system *function* ⎕NL will be invoked, which lists ordinary variables, functions, and operators, not class members.

The right argument is a scalar or vector which indicates which types of class member should be included in the result:

Code	***Type of member***
2	Properties
3	Function methods
4	Operator methods
8	Events
10	Constructors

For example:

```
Point {
  X
  Y

  ∇R←MAG
    R←((X*2)+(Y*2))*0.5
  ∇
}
      Point.⎕NL 2      ⍝ Applied to a class reference (properties)
X
Y
      PT←⎕NEW 'Point'
      PT.⎕NL 2 3       ⍝ Applied to an object ref (properties and methods)
MAG
X
Y
```

⎕NL may also be used to list the members of External or System classes:

```
      PT←'java' ⎕NEW 'java.awt.Point'
      PT.⎕NL 2             ⍝ Properties
x
y
```

```
      PT.⎕NL 3             ⍝ Methods
clone
distance
distanceSq
equals
getClass
getLocation
getX
getY
hashCode
move
notify
notifyAll
setLocation
toString
translate
wait

      T←'⎕' ⎕NEW 'Timer'
      T.⎕NL 2             ⍝ Properties
children
class
data
enabled
events
interval
methods
name
opened
properties
self
tie
      T.⎕NL 3             ⍝ Methods
Close
Create
Delete
New
Open
Send
Set
Trigger
      T.⎕NL 8             ⍝ Events
onClose
onDestroy
onOpen
onSend
onTimer
```

For the R interface, `⎕NL` can be used to produce a list of all the R global variables in the current R session (right argument 2), or all the functions which are available in the currently-loaded packages (right argument 3). This will usually be a long list (several thousand functions).

⎕OID Object ID

Implemented for Internal, External and System classes.

Syntax:

```
    integer ← objref.⎕OID
    integer ← classref.⎕OID
    integer ← ⎕OID              (Within user-defined method, same as ⎕THIS.⎕OID)
```

⎕OID returns an integer which uniquely identifies a given object instance or class definition within the workspace. It starts at 1 for the first-ever object created in the workspace, and increments by one each time a new object is created (for example by a call to ⎕NEW or ⎕CLONE).

On 32-bit versions of APLX, ⎕OID may return a floating-point data type if you have created more object instances than will fit in a 32-bit signed integer.

⎕PARENT Base (parent) class

Implemented for Internal and External classes. Not implemented for System classes.

Syntax:

```
    classref ← objref.⎕PARENT
    classref ← classref.⎕PARENT
    classref ← ⎕PARENT           (Within user-defined method, same as ⎕THIS.⎕PARENT)
```

The niladic system method ⎕PARENT returns a reference to the parent class of either an object, or a class. The result is always a class reference, or the Null object if there is no parent. It is a synonym for ⎕BASE.

⎕REF Force reference result

Implemented for Internal and External classes. Not implemented for System classes.

Syntax:

```
    objref ← objref.⎕REF
    objref ← objref.method.⎕REF
    objref ← classref.⎕REF
    objref ← classref.method.⎕REF
    objref ← ⎕REF           (Within user-defined method, same as ⎕THIS.⎕REF)
```

⎕REF is a special modifier method which can be used to force another method to return an object reference rather than 'unbox' the data before it is returned to APL. It can be used with either niladic or monadic external system methods, or on external objects directly.

When used on an object which would normally be returned to APL as an object reference, the niladic system method ⎕REF does nothing; it simply returns the reference to the object (or class definition) unchanged, as a scalar.

Where ⎕REF becomes useful is for cases where the data would normally be converted from an object in an external architecture, to a native APL data type. The two most common cases are strings and arrays. For example, if a method in an external class returns a string object, it is normally converted to an APL character vector and this data is returned to APL:

```
      DT←'.net' ⎕NEW 'System.DateTime' 2005 12 3 13 45 21
      DT.ToLongDateString
03 December 2005
      ⍴DT.ToLongDateString
16
      ⎕DR DT.ToLongDateString
4
```

Whilst this default behavior is usually desirable, there are occasionally cases where it is better to leave the string as an external object, and return a reference to it instead. ⎕REF forces this to happen:

```
      ExtString←DT.ToLongDateString.⎕REF
      ExtString
[.net:String]
      ExtString.Length
16
```

In the above example, `ExtString` is a scalar reference to a `String` object which exists in the .Net environment.

There are several reasons why you might want to use ⎕REF:

- If the data is about to be passed to another external method, and you are not interested in the contents of the intermediate variable, it is more efficient to pass just a reference back to APL, rather than copying the data contents twice.
- Some values or precision may be lost in converting the data from the external subsystem to APL. For example, the external data may comprise 64-bit integers; if you are running a 32-bit version of APLX, these will by default be converted to floating-point form, and may therefore lose precision.
- You may want to keep it as object so that you can run methods, defined for the external object class, directly on it.

See also `⎕VAL` which does the opposite: it attempts to 'unbox' data which would otherwise be returned as an object reference.

`⎕STATE` Property names and values

Currently implemented for Internal classes only.

Syntax:

```
    objref ← objref.⎕STATE B
    objref ← ⎕STATE B
```
(Within user-defined method, same as `⎕THIS.⎕STATE`*)*

The monadic system method `⎕STATE` returns a nested matrix of two columns, containing the names of each property in the first column, and the current value of each property in the second. It takes a right argument (0 or 1) which indicates which properties should be included in the result. If the right argument is 0, only values which differ from the default value are included. If it is 1, all properties which have a defined value are included:

```
      ⎕CR 'Circle'
Circle {
    X←0
    Y←0
    COLOR←'Pale green'
    RADIUS←100

    ∇R←Area
     R←(○1)×RADIUS*2
    ∇
}

      C←⎕NEW Circle
      C.X←23
      C.Y←12
      C.⎕STATE 0
 X 23
 Y 12
```

```
      C.⎕STATE 1
 X              23
 Y              12
 COLOR  Pale green
 RADIUS        100

      ⎕DISPLAY C.⎕STATE 1
┌→───────────────────────┐
↓ ┌→┐                    │
│ │X│      23            │
│ └─┘                    │
│ ┌→┐                    │
│ │Y│      12            │
│ └─┘                    │
│ ┌→────┐  ┌→─────────┐  │
│ │COLOR│  │Pale green│  │
│ └─────┘  └──────────┘  │
│ ┌→─────┐               │
│ │RADIUS│ 100           │
│ └──────┘               │
└∊───────────────────────┘
```

This operation is known as *serialization* of an object. Note that, if any of the properties include a reference to another object, the state of that object will not be expanded (serialized) in the returned matrix. The operation is therefore a 'shallow' copy of the data.

⎕UNMIX Remove mixins from object

Implemented for Internal classes only.

Syntax:

```
    binary_vec ← objref.⎕UNMIX mixin_refs
    binary_vec ← ⎕UNMIX mixin_refs
```
(Within user-defined method, same as ⎕THIS.⎕UNMIX*)*

The System Method ⎕UNMIX can be used to remove one or more mixins from an object. It takes a right argument which is a scalar or vector list of mixin-references to delete, and returns a binary vector with 1 for each mixin removed, and 0 if the mixin reference could not be found:

```
      inv.⎕mixins
[Fax] [java:Date]
      inv.⎕unmix inv.⎕mixins
1 1
      inv.⎕mixins
```

Note that you don't normally need to delete mixins explicitly; they will be deleted automatically when the object which owns them is deleted.

See the section on Mixins for more information.

⎕VAL Force value result

Implemented for External classes only.

Syntax:

```
    data ← objref.⎕VAL
```

The niladic system method ⎕VAL attempts to force data within an external object to be 'unboxed', so that the value of the data, rather than an object reference, is returned to APLX.

When used on an object which would normally be returned to APL as data, it does nothing; it simply returns the data as normal. Similarly, when used on an object which cannot be converted to simple APL data types, it also does nothing; the object reference is returned.

Where ⎕VAL becomes useful is for cases where you have created an object (using typically ⎕NEW or ⎕REF) in the external environment, and where the object *can* be represented as ordinary APL data. The two most common cases are strings and arrays, but it can also be used with numeric types (which can be converted to integers or floating-point values).

In this example, we create an array of three strings in .Net:

```
      StringType←'.net' ⎕GETCLASS 'System.String'    ⍝ Get Type of System.String
      ArrayType←'.net' ⎕GETCLASS 'System.Array'      ⍝ Get Type of System.Array
                                                     ⍝ so we can call static
                                                     ⍝ method 'CreateInstance'
      ArrayType.CreateInstance StringType 3          ⍝ Create array of 3 strings
[NULL OBJECT] [NULL OBJECT] [NULL OBJECT]            ⍝ Oops! Array was unboxed

      A←ArrayType.CreateInstance.⎕REF StringType 3   ⍝ Keep array as object
      A
[.net:String[]]                                      ⍝ Reference to .Net array
      A.SetValue 'First Item' 0                      ⍝ Insert some values
      A.SetValue 'Second Item' 1
      A.SetValue 'Third Item' 2

      A                                              ⍝ Still a reference
[.net:String[]]
      A.⎕VAL                                         ⍝ Unbox array to get contents
 First Item Second Item Third Item
      ⎕DISPLAY A.⎕VAL
┌→──────────────────────────────────────────┐
│ ┌→─────────┐ ┌→──────────┐ ┌→─────────┐ │
│ │First Item│ │Second Item│ │Third Item│ │
│ └──────────┘ └───────────┘ └──────────┘ │
└∊──────────────────────────────────────────┘
```

This example in R creates a 'factor' and uses ⎕VAL to extract from it the underlying vector of indices:

```
      r←'r' ⎕new 'r'
      f←r.factor (⊂'abc' 'def' 'abc')
      f
[r:factor]
      f.⎕val
1 2 1
```

See also ⎕REF which does the opposite: it attempts to force data, which would otherwise be 'unboxed', to be returned as an object reference.

Section 9: Interfacing to other languages

Overview of interfacing to other languages

Interfacing to .Net, Java, and other object-oriented software

APLX allows you to use classes written in modern object-oriented languages such as the .Net languages (C#, Visual Basic, and other languages which generate code for the .Net Common Language Runtime), Java, and Ruby, and also the R statistical language. You can create and use instances of external classes, call 'static' (shared) methods, and in some cases evaluate statements in the external language directly. See the next section on Using External Classes for full details.

Interfacing to procedural languages and shared libraries

APLX includes two main facilities for calling external procedural (i.e. not object-oriented) code from within APL, for example operating-system libraries (DLLs) or code written in C. These are:

- The `⎕NA` system function, which allows you to define 'external functions' (i.e. calls to shared libraries or DLLs) and call them directly from APL
- The Auxiliary Processor (shared variable) mechanism, which allows you to interface to custom code written in C or other low-level languages. The details of writing an Auxiliary Processor are discussed in a later section. See the `⎕SVO` system function for more information on using auxiliary processors

The choice between these two mechanisms depends on what you are trying to do. To interface to an existing shared library or DLL, or to call the operating system directly, `⎕NA` is usually the better choice. To write a full-featured sub-system which provides extensive functionality to APLX, the auxiliary processor mechanism may be preferred.

Interfacing via OLE and OCX (*Windows only)*

Under Windows, Microsoft provide a generalized means for interfacing to external applications and using external controls. This mechamism is variously known as OLE, OCX, ActiveX, and COM. Although to some extent it has been superseded by the .Net architecture, it remains important because it is supported by a wide range of software. There are three ways in which you can use this feature:

(a) You can place external OCX or ActiveX controls directly on your APLX user-interface windows. For example, you can use an Adobe® Acrobat™ control as though it were a built-in APLX control. This is done simply by creating a window and then creating an external control on it.

(b) You can embed OLE (Object Linking and Embedding) documents in your windows. For example, you could include a Microsoft Word document as part of a more complex window, merging the Word menus and toolbar with your own. To allow this, APLX includes a control called an 'OLEContainer' which you place on your window. You can then specify the document to be embedded or linked, or allow the user to select one.

(c) You can invoke and exchange data and commands with external OLE Server Applications. This is somewhat similar to (b), but the application runs independently of APLX and is not embedded in one of your windows. For example, you could invoke Excel, cause it to load a spreadsheet, and extract the data from the spreadsheet as an APL nested array, without the user being involved.

See the section on OCX/ActiveX Controls and OLE Automation in the separate manual *APLX System Classes and User-Interface Programming* for full details.

Using External Classes

Introduction

Modern object-oriented language frameworks such as Java, .Net and Ruby provide a consistent mechanism for exploring class libraries, discovering what classes are defined in the library, and detailing the class methods and properties. This allows objects to be created and used from outside the specific environment in which they were written. (These facilities, which provide so-called *metadata* about a given class together with the ability to create new classes at runtime, are sometimes described by the term *Reflection*.)

The underlying object-oriented model in APLX is broadly similar to that of the .Net languages, Java, or Ruby. This makes it possible to access all the powerful facilities of the .Net and Java class libraries, as well as custom software written in the mainstream object-oriented languages in use today, directly from APLX.

Creating instances of external classes

You can create instance of external classes in much the same way as you create instances of APL (user-defined) classes, by means of the system function `⎕NEW`. The right argument of `⎕NEW` can be either a class reference (which is typically the case when an APL user-defined class is being used), or a class name as a character vector. In the latter case, the optional left argument of `⎕NEW` determines the environment in which the class is to be found. For example:

Create an instance of the .Net `DateTime` class, defined in the .Net class libraries, specifying the initial date to the constructor of that class:

```
      NETDATE←'.net' ⎕NEW 'System.DateTime'  2007 6 20 9 32 3
```

Create an instance of the Ruby DateTime class, defined in the Ruby class libraries:

```
      RUBYDATE←'ruby' ⎕NEW 'DateTime'  2007 6 20 9 32 3
```

Create an array of complex numbers in the R statistical language:

```
      C←'r' ⎕NEW 'complex' (3 2⍴(1 2) (3 4) (5 6) (7 8) (9 10) (11 12))
```

The left argument to `⎕NEW` is the environment identifier. For external classes, it corresponds to a shared library or DLL, as follows:

For 32-bit implementations of APLX:

Left arg	***Environment***	***Windows DLL***	***Macintosh bundle***	***Linux shared library***
(Omitted)	User-defined APL class	None	None	None
`'⎕'`	System class	None	None	None

`'.net'`	Microsoft .Net	`aplxobj_net.dll`	Not supported	Not supported
`'java'`	Java	`aplxobj_java.dll`	`aplxobj_java.bundle`	`aplxobj_java.so`
`'r'`	R	`aplxobj_r.dll`	`aplxobj_r.bundle`	`aplxobj_r.so`
`'ruby'`	Ruby	`aplxobj_ruby.dll`	`aplxobj_ruby.bundle`	`aplxobj_ruby.so`
Other	Customized environment	`aplxobj_XXX.dll`	`aplxobj_XXX.bundle`	`aplxobj_XXX.so`

For 64-bit implementations of APLX:

Left arg	***Environment***	***Windows DLL***	***Linux shared library***
(Omitted)	User-defined APL class	None	None
`'⎕'`	System class	None	None
`'.net'`	Microsoft .Net	`aplx64obj_net.dll`	Not supported
`'java'`	Java	`aplx64obj_java.dll`	`aplx64obj_java.so`
`'r'`	R	`aplx64obj_r.dll`	`aplx64obj_r.so`
`'ruby'`	Ruby	`aplx64obj_ruby.dll`	`aplx64obj_ruby.so`
Other	Customized environment	`aplx64obj_XXX.dll`	`aplx64obj_XXX.so`

What actually happens 'under the hood' here is that user-defined and system classes are handled directly by the APLX interpreter. Operations to create and use object classes written in other environments are passed to the external library (Windows dynamic link library, Macintosh bundle, or Linux shared library) whose name is given in the table. This provides an extensible interface, allowing further environments to be added in the future. For example, to add an interface to Mono (the open-source equivalent of .Net) it would not be necessary to change the APLX interpreter at all; all that would be required is to supply a new interface library `aplxobj_mono.dll`.

It is also possible to create instances of external classes by using a class reference rather than a character vector as the right argument to `⎕NEW`, but first you need to obtain the reference from the external system using the `⎕GETCLASS` system function:

```
      NETDATECLASS←'.net' ⎕GETCLASS 'System.DateTime'
      NETDATE←⎕NEW NETDATECLASS 2007 6 20 9 32 3
```

Obtaining and using a class reference can be much more efficient than supplying a class name to `⎕NEW` if you need to create a large number of objects.

Customized interfaces

As well as being extensible to further public object-oriented environments, the mechanism also allows the same APL syntax to be used for accessing custom class libraries written in languages (such as C++) which do not support metadata and Reflection. For example, if a financial institution wanted to make use of a timeseries analysis class library written in C++, it could write a simple interface DLL `aplxobj_ts.dll` which would allow classes contained in that library to be used from APL:

```
    TS←'ts' ⎕NEW 'TimeSeries'
```

The APLX interpreter, seeing this line, would pick up the environment identifier `'ts'` which would cause it to search for `aplxobj_ts.dll` to handle the creation and use of the classes within the custom library. Unlike the generalized interfaces to .Net, Java, and Ruby, this custom interface would of course support only the specific set of classes for which it had been written, with the names and other details of the supported classes and their methods hard-coded into the interface DLL rather than being obtained at runtime. Nonetheless, it is a powerful extension of the ability of APL to use external code.

Calling methods and accessing properties

Once you have a reference to an object, you can use a consistent syntax to access its methods and properties. It makes no difference whether the object is an instance of an internal APL class, or of an external Ruby, Java, or .Net class. For external calls, the APL interpreter automatically marshals any parameters supplied to the form required by the external class method (assuming that such a conversion is possible). For example, if the method requires a parameter which is of type signed 16-bit integer, the APL interpreter will convert any supplied binary, integer, or floating-point data type to the required 16-bit form, provided that the number is integral and is in range. Where the required parameter is a string, APLX will automatically convert from an APL character vector. Where the required parameter is itself an object reference, the user can supply an APL reference to an object (in the same environment, of course - you cannot pass a Ruby object reference to a .Net method).

The information which allows this conversion to happen successfully is the metadata which describes the external class. Depending on the external environment, you can see this metadata in human-readable form by using the system method `⎕DESC`, which is like `⎕NL` except that it returns types for properties, and the prototypes of methods:

```
      NETDATE.⎕DESC 3
System.String ToString()
System.String ToString(System.String)
System.String ToString(System.IFormatProvider)
System.String ToString(System.String, System.IFormatProvider)
System.Type GetType()
System.DateTime Add(System.TimeSpan)
System.DateTime AddDays(Double)
System.DateTime AddHours(Double)
... etc
```

This means, for example, that the AddDays method of the .Net `System.DateTime` class takes a double-precision floating point value as an argument, and returns a new `DateTime` object which represents the original date-time value plus the number of days (or parts of days). We can see this by calling the method with a suitable parameter, but without assigning the result:

```
      NETDATE.AddDays 1
[.net:DateTime]
```

What has happened here is that the .Net class library has created a new `DateTime` object, and a reference to it has been passed back to APL. Because we have not assigned or used the object reference, the default display form of the object has been written to the session window, and the object has then been deleted.

To display the date/times for exactly one, two, and three days following, we can use the APL 'each' operator to run the `AddDays` method three times, with three separate arguments, and then run the `⎕DS` method on each of the three new temporary `DateTime` objects which will be returned:

```
      (NETDATE.AddDays¨⍳3).⎕DS
 21/06/2007 09:32:03 22/06/2007 09:32:03 23/06/2007 09:32:03
```

Properties are handled in a similar way to methods, although not all external classes really have properties as such; some only have 'getter' and 'setter' functions. You can read properties back directly, and assign to them using the normal APL assignment arrow.

Calling 'static' methods

Object-oriented languages like C# and Java include support for so-called *static* or *shared* class methods. These are methods which belong to a class but which do not manipulate an individual object. You can call these static methods from APLX in one of two ways:

- If you have an object of the appropriate class, or a reference to the class, you can call the static method as though it were a normal object method. The object is just ignored when the call is made.
- Alternatively, you can use the `⎕CALL` system function. As an example, consider the Java class 'java.lang.Integer', which includes a static method to convert an integer into a binary string. You can call it from APLX as follows:
-
-
```
      'java' ⎕CALL 'java.lang.Integer.toBinaryString' 37
100101
```

Overloaded methods and syntactic ambiguity

One common feature of modern object-oriented languages is that they support *overloaded* methods, that is to say the same method name is used for more than one method, but with different numbers or types of arguments. An example is shown above in the `⎕DESC` listing of methods for the .Net `System.DateTime` object; there are four versions of `ToString()`, one of which takes no arguments, two of which take a single argument of different types, and one of which takes two arguments.

Normally, APLX is able to handle this unambiguously by examining the parameters supplied, and choosing the correct match amongst the possible overloaded methods. Ambiguities are sometimes possible, however. The most important of these is the use of strand notation with niladic methods.

Consider the following line of traditional APL:

```
      FORMAT 'THIS STRING'
```

It is impossible, looking at this line in isolation, to know whether this is a call to a monadic function `FORMAT` with `'THIS STRING'` as the argument, or is a reference to a variable `FORMAT`, or is a call to a niladic function `FORMAT`. In the latter two cases, the returned data would be joined with the character vector `'THIS STRING'` to produce a length-2 nested vector.

In traditional APL, the interpreter is able to resolve this ambiguity by looking at the type of the symbol `FORMAT`. The same name cannot simultaneously refer both to a monadic function, and to a niladic function or a variable.

When calling an external method, it would in many cases be possible to resolve the ambiguity in the same way. However, if the method is overloaded and exists in a form which take no arguments as well as in a form which takes arguments, it may not be possible to know which was intended.

APLX therefore assumes that ***if an external method call syntactically could take arguments, then it does take arguments***. Hence, the line:

```
      NETDATE.ToString TEXT
```

(where `TEXT` is a character vector) is always assumed to be a call to the version of `ToString()` which takes a `String` argument (the second form in the list), not to any niladic version of `ToString()`.

If you really want to call the niladic version of the method, and join the result to the variable `TEXT`, then you can force this behavior by using parentheses:

```
      (NETDATE.ToString) TEXT
```

Note that you cannot, in APL, tell the difference syntactically between a call to a niladic function or method, and a reference to a variable or property. The interface code will examine the metadata to make the right call. Note also that only internal APL user-defined methods can be dyadic functions, or APL operators.

Object lifetimes and garbage collection

Internally, APLX uses a reference-count method to ensure that objects are deleted when they are no longer needed (this is supplemented by special code to handle the problem of circular references, which might otherwise make it impossible to delete certain objects). Most of the external object-oriented environments which APLX interfaces to, including .Net, Ruby and Java, use a mark-and-sweep garbage collect system. In this system, a periodic sweep through memory is carried out to find all objects which are no longer referenced, and which therefore can be deleted.

The approach taken is for APLX to be the arbiter of object lifetimes. When an external object is created (either explicitly using `⎕NEW`, or as a result of some other operation), a reference to the external object is retained in the external environment (for example, the .Net runtime), and a copy of this reference is passed back to the APL interpreter. Because there is a reference to the object held in memory in the external environment, it will not be deleted by the mark-and-sweep garbage collect.

When APL's own reference count for the object falls to zero, APL deletes from the workspace its own data structure which describes the external object. Just before doing this, it calls the external sub-system to indicate that the object is no longer needed. Any cleaning-up required before deletion is then done, and the local reference to the object is deleted or replaced by a NULL. As a result, when the next garbage collect takes place in the external system, the memory used by the object will be reclaimed (unless of course there is another reference to the same object somewhere else in the external system).

For cases where the external environment does not use a garbage-collect system (for example, a custom interface to a class library in C++), the mechanism is similar; the only difference is that when APL notifies the external interface that the object is no longer needed, the external interface carries out an explicit delete rather than relying on replacing the reference with a NULL.

A special case arises if a reference to an external object is saved, for example when you `)SAVE` the workspace. When you re-load the workspace, the saved reference is no longer valid. APLX will issue a warning and set it to NULL.

Issues with naming conventions

One problem which arises with trying to unify the way in which external classes are accessed is that of naming conventions. For example, Ruby allows question marks and exclamation marks in method names. To work around this problem, the $ character can be used as an escape character in external names. It has the effect of treating the next character as part of the name. (If you are interfacing to a language in which $ itself is a valid character in a name, you can escape the dollar itself by writing $$).

Errors reported from the external environment

If the external environment raises an error (or *exception*), APLX will normally try to print the error message or exception text on the session window, and then raise an APL error (typically DOMAIN ERROR). For example:

```
      f←'ruby' ⎕new 'ftp'
#<NameError: uninitialized constant ftp>
DOMAIN ERROR
      f←'ruby' ⎕new 'ftp'
        ^

      f←'.net' ⎕NEW 'ftp'
Class ftp not found in current search list
DOMAIN ERROR
      f←'.net' ⎕NEW 'ftp'
        ^
      dt←'.net' ⎕NEW 'DateTime' 'Bastille day'
Constructor on type 'System.DateTime' not found.
DOMAIN ERROR
      dt←'.net' ⎕NEW 'DateTime' 'Bastille day'
         ^
```

For most external environments, you can find out more about what caused the the exception by using the `⎕LE` Last Exception system function.

Inheriting from external classes

Your own classes (written in APL) cannot inherit directly from an external class. However, you can achieve much the same result by using `mixins`, which allow you to 'mix in' the properties and methods of one or more external classes into your APL objects.

Interfacing to .Net

Specifying a call to the .Net environment

You can interface to .Net by supplying `'net'` or `.net` as the environment string (left argument) for `⎕NEW`, `⎕GETCLASS`, or `⎕CALL`. These system functions will allow you to create an instance of a .Net class, or to call a .Net static (shared) method.

Specifying .Net class names

.Net classes are defined in *Assemblies* (each assembly usually corresponds to a single .Net library DLL, for example `mscorlib.dll` which contains the basic .Net utility classes), and are organized into *Namespaces* (such as `System`, `System.Text`, `System.IO` and so on). For example, Microsoft provide a class called `Font` for representing fonts. This is defined in the `System.Drawing` namespace, so its fully-qualified name is `System.Drawing.Font`. The code for this class is held in the System.Drawing assembly, in the shared library `system.drawing.dll`.

When you use `⎕NEW` (or `⎕GETCLASS` or `⎕CALL`) to refer to a .Net class, you can specify the class name in one of the following ways:

- As an undecorated class name, without the namespace qualifier, for example: `'Font'`. In this case, the assembly must be one of those loaded by default or already loaded by a previous call, and the namespace must be included in the namespace search list (see below).
- As a fully (or partially) qualified namespace and class name, for example `'System.Drawing.Font'`. In this case, the assembly must be one of those loaded by default, or already loaded by a previous call, but the namespace does not have to be included in the namespace search list.
- As an undecorated or fully-qualified class name, and a simple file name specifying the DLL in which the assembly can be found, for example: `'System.Data.SqlClient,system.data.dll'`. A comma is used as the delimiter between the class and file name. In this case, the DLL must exist in the standard location known as the *Global Assembly Cache* (GAC).
- As an undecorated or fully-qualified class name and a full path name specifying the DLL, for example: `'MyBase.MyFirstClass,c:\devt\myclasses.dll'` Again a comma is used as a delimiter between the class and file name.

Setting the search paths for namespaces and .Net assemblies

When you use `⎕NEW` (or `⎕GETCLASS` or `⎕CALL`) to refer to a .Net class, the system searches through the namespaces and assemblies which are in its search path to resolve the class name. If the class is not found, you will get an error:

```
      SW←'.net' ⎕NEW 'StringWriter'
Class StringWriter not found in current search list
DOMAIN ERROR
      SW←'.net' ⎕NEW 'StringWriter'
         ^
```

The above example has failed because `StringWriter` is a class defined in the `System.IO` namespace, which is not included in the default search list. You can set the current search path for .Net namespaces and DLLs by using the system function `⎕SETUP` and the `'using'` keyword. For example, we can tell the .Net subsystem to include `System.IO` in the search list:

```
      '.net' ⎕SETUP 'using' 'System' 'System.IO'
      SW←'.net' ⎕NEW 'StringWriter'
      SW
[.net:StringWriter]
```

The namespaces (and optionally DLLs in which they are located) should be supplied as character vectors after the keyword:

```
      '.net' ⎕SETUP 'using' 'System' 'System.Text' 'QMath.Geom,c:\dev\qmath.dll'
```

Each element comprises either just a namespace (such 'System.Text'), or a namespace followed by the name of the DLL in which it is located. This can be a full path name, or just the name of the DLL (in which case the DLL should be in the Global Assembly Cache).

For convenience, the path is set by default to include the most important .Net libraries, as follows:

```
System
System,system.dll
System.Text
System.Collections
System.Windows.Forms,system.windows.forms.dll
System.Drawing,system.drawing.dll
```

You can read the current search list by supplying the 'using' keyword to `⎕SETUP` with no arguments.

Conversion of .Net data types to APL data

When your read a .Net property, or call a method which returns a result, APLX by default applies the following data conversion rules:

- Any .Net numeric types (Int32, Int64, single- or double-precision floats, decimal, etc) are converted to APL integers or floats. 64-bit integers are converted to APL floats on 32-bit platforms, to APL integers on 64-bit platforms.
- .Net Booleans are converted to APL binary values 0 or 1
- .Net Strings and Chars are converted to APL character arrays, translated from Unicode to APLX internal representation. (Any characters which do not appear in the APLX character set are converted to question marks.)
- .Net Enumeration types are converted to APL integers *(see below)*
- Simple .Net arrays are converted to APL arrays, with individual elements converted as above.

Anything else is left as an Object in the .Net environment, and a reference to the object is returned to APL.

There are some special cases to consider. The data might not be convertible at all, or it might lose precision in the conversion. For example, a .Net `Decimal` might have a higher precision than an APL double-precision floating point can represent. To handle cases like this, APLX provides the `⎕REF` system method. This forces the data to remain as a .Net object. You can then call `ToString`, or other .Net methods appropriate to the `Decimal` data type, to manipulate the data without losing precision.

An example which cannot be represented at all is where a .Net `Double` contains a NaN (Not A Number). APL does not handle NaNs, so it cannot be converted to an APL floating-point value. Instead, NaNs are left as Objects. If you try to use the data in an APL expression, you will get a DOMAIN ERROR, but you can see that it is a NaN and use `ToString` and other operations on it.

Enumeration Types

Enumeration types are special classes in .Net, which contain sets of alternative values which a particular type of variable can hold (somewhat similar to `enum` types in the C language). When a .Net nethod or property returns an Enumeration type, APLX converts it to the equivalent integer. Equally, calls which expect an Enumeration type can be passed the equivalent integer from APL (the type is converted automatically). This means that Enumerations which are sets of bits can be combined in APL using the + primitive. However, for readability of code, and to avoid having to check the specific value of an Enumeration, you can use `⎕GETCLASS` to fetch the Enumeration type, and use that directly.

For example, most .Net dialogs (such as OpenFileDialog, which puts up a dialog inviting the user to select an existing file) return a Enumeration type called `DialogResult`, indicating which button (OK, Cancel, etc) was clicked to end the dialog. We can access this class to see what the various enumeration values are:

```
      DR←'.net' ⎕GETCLASS 'DialogResult'
      DR.OK
1
      DR.OK.ToString
OK
      DR.Cancel
2
      ⎕BOX DR.⎕NL 2
Abort Cancel Ignore No None OK Retry value__ Yes
```

This means that, if you call the `ShowDialog` method of the `OpenFileDialog`, you can test which button ended the dialog either by checking the numeric value returned, or by seeing if it is equal to the named enumeration value:

```
      ∇R←FileDialog;DR;DLG
[1]   ⍝ Put up file dialog, return file selected or empty vector if none
[2]   DR←'.net' ⎕GETCLASS 'DialogResult'
[3]   DLG←'.net' ⎕NEW 'OpenFileDialog'
[4]   :If DLG.ShowDialog=DR.OK
[5]     R←DLG.FileName
[6]   :Else
[7]     R←''
[8]   :End
      ∇
```

Parameters passed by reference

To handle the (relatively uncomon) cases where a .Net method takes an argument 'by reference', and modifies one or more of the arguments in-situ, you can use `⎕SETUP` and the `'byref'` keyword. See the documentation for `⎕SETUP`.

Using the .Net interface from multiple APL tasks

There are no special restrictions on using the .Net interface from multiple APL tasks. Each task will have an independent copy of the interface, which will be deleted on `)CLEAR`, `)LOAD`, or `)OFF`.

Inheriting from .Net classes

Your own classes (written in APL) cannot inherit directly from a .Net class. However, you can achieve much the same result by using mixins.

In this example, the constructor of a user-defined class called `APLDate` 'mixes-in' the .Net class `DateTime`

```
      ⎕cr 'APLDate'
APLDate {
∇APLDate b
⍝ Constructor for APLDate class
⍝ Argument is vector of Year Month Day Hour Sec
⍝ (or just Year Month day)
⍝ If no argument supplied, use current ⎕ts value
:If 0=⍴b
  '.net' ⎕mixin(⊂'DateTime'),6↑⎕ts
:Else
  '.net' ⎕mixin(⊂'DateTime'),b
:End
∇

∇r←ts
⍝ Return .Net DateTime in ⎕TS format
r←Year,Month,Day,Hour,Minute,Second,Millisecond
∇

∇r←ts_gmt;gmt
⍝ Return the date/time adjusted to GMT, in ⎕ts form
gmt←ToUniversalTime
r←gmt.Year,gmt.Month,gmt.Day,gmt.Hour,gmt.Minute,gmt.Second,gmt.Millisecond
∇
}
      dt←⎕new APLDate
```

As well as the methods written in APL, all the public properties and methods of the .Net `DateTime` class become available in the APL class `APLDate`:

```
      dt←⎕new APLDate
      dt.⎕nl 2
Date
Day
DayOfWeek
DayOfYear
Hour
```

```
Kind
MaxValue
Millisecond
MinValue
Minute
Month
Now
Second
Ticks
TimeOfDay
Today
UtcNow
Year

      dt.ToLongDateString    ⍝ Method 'mixed-in' from .Net DateTime
19 March 2009
      dt.ts                  ⍝ Method written in APL
2009 3 19 16 0 5 0
      dt.ts_gmt
2009 3 19 21 0 5 0
```

Exceptions

If the .Net environment raises an error (or *exception*), APLX will normally try to print the error message or exception text on the session window, and then raise an APL error (typically DOMAIN ERROR). You can get further information by using `⎕LE` Last Exception to read back the .Net exception object: For example:

```
      '.net' ⎕NEW 'DateTime' 'Bastille Day'
Constructor on type 'System.DateTime' not found.
DOMAIN ERROR
      '.net' ⎕NEW 'DateTime' 'Bastille Day'
      ^
      exception←'.net' ⎕LE 1
      exception
[.net:MissingMethodException]

      exception.⎕NL 2
Data
HelpLink
InnerException
Message
Source
StackTrace
TargetSite

      exception.Message
Constructor on type 'System.DateTime' not found.

      exception.Source
mscorlib

      exception.ToString
System.MissingMethodException: Constructor on type 'System.DateTime' not found.
   at System.RuntimeType.CreateInstanceImpl(BindingFlags bindingAttr, Binder bin
      der, Object[] args, CultureInfo culture, Object[] activationAttributes)
   at System.Activator.CreateInstance(Type type, BindingFlags bindingAttr, Binde
      r binder, Object[] args, CultureInfo culture, Object[] activationAttribute
      s)
```

```
at NetBridge.NetBridge.CreateNamedInstance(String class_name, Object[] arg_li
   st)
at APLXObj_New(Void* arch_if, Void** objref, wsobj* args, _ExportedProcs* pro
   cs, SByte* errmsg)
```

User-interface programming in .Net

User-interface programming in APLX can be done by using the built-in System classes (formerly accessed through ⎕WI). This has the advantages that it is relatively easy to do, and that APLX applications using the System classes will work on other platforms (e.g. the Macintosh). But for more flexibility, you can alternatively use Microsoft's .Net classes directly.

User-interface programming using the .Net framework (`System.Windows.Forms`) is a special case of using .Net objects from within APLX. See the Microsoft documentation for full information on this topic.

From the APL programmer's point of view, there are a few specific points to note:

- The typical scenario is that you create a .Net `Form` object, set various properties of the `Form` (such as the `Text` property, which is the window title), and then create further items using control classes such as `Button` and `TextBox`. You need to use the `Add` method of the form's `Controls` property to link up the controls and the form. *(Note: If you have a copy of Microsoft Visual Studio or Visual Studio Express, you can use the form designer to design a form interactively, and then copy the dimensions and other properties created in C# or Visual Basic into your APLX code, adjusting the syntax as necessary).*
- To handle .Net events, you need to assign the name of an APL callback function (or some other expression) to one of the control's event properties. For example, a `Button` has a `Click` event, which fires when the button is clicked. As with System classes, the event is held in a queue and de-queued using the system function ⎕WE, which causes your event handler to be run.
- When one of your callback functions is running, there are three system functions which you can use to find out more about the event. These are:

⎕EVA	Returns a reference to the event argument object (class `EventArgs`)
⎕EVN	Returns the name of the event (such as `'Click'`)
⎕EVT	Returns a reference to the target object, for example the `Button` object

Example 1

The following complete example (available in the workspace 10 HELPDOTNET) shows a simple application which displays a window with an edit box, a text box to output results, and a button. When the button is clicked, it simply evaluates the APL expression typed into the edit box and displays the result in the text box. Note in particular the callbacks set up on lines 32 and 33. The first of these (the `Closed` event) is triggered when the window is closed; this terminates the function by clearing the `)SI`, and this in turn causes all the object references to be deleted because they are held in localized variables. The second call back (the `Click` event of the button) causes the `EVALUATE` function to be run.

```
      ∇DOT_Forms;X;F;EditResult;EditIn;ButtonDo
[1]   ⍝ Simple example of using Windows Forms from APLX v4
[2]   ⍝ Create main form
[3]    F←'.net' ⎕NEW 'Form'
[4]    F.Text←'Expression Evaluator'
[5]   ⍝
[6]   ⍝ Create edit box for input line
[7]    EditIn←'.net' ⎕NEW 'TextBox'
[8]    EditIn.Left←12 ⋄ EditIn.Top←21
[9]    EditIn.Width←265 ⋄ EditIn.Height←20
[10]   EditIn.Font←'.net' ⎕NEW 'Font' 'APLX Upright' 10
[11]   F.Controls.Add EditIn
[12]  ⍝
[13]  ⍝ Create multi-line box for result
[14]   EditResult←'.net' ⎕NEW 'TextBox'
[15]   EditResult.Multiline←1
[16]   EditResult.Left←12 ⋄ EditResult.Top←50
[17]   EditResult.Width←265 ⋄ EditResult.Height←180
[18]   EditResult.Font←'.net' ⎕NEW 'Font' 'APLX Upright' 10
[19]   F.Controls.Add EditResult
[20]  ⍝
[21]  ⍝ Create button
[22]   ButtonDo←'.net' ⎕NEW 'Button'
[23]   ButtonDo.Left←94 ⋄ ButtonDo.Top←240
[24]   ButtonDo.Width←100 ⋄ ButtonDo.Height←22
[25]   ButtonDo.Text←'Evaluate'
[26]   F.Controls.Add ButtonDo
[27]  ⍝
[28]  ⍝ Make the button accept Enter as equivalent to clicking
[29]   F.AcceptButton←ButtonDo
[30]  ⍝
[31]  ⍝ Add callbacks
[32]   F.Closed←'"Cleaning up.." ⋄ →'
[33]   ButtonDo.Click←'EditResult EVALUATE EditIn.Text'
[34]  ⍝
[35]  ⍝ Show the window
[36]   F.Show
[37]  ⍝
[38]  ⍝ Process events
[39]   X←⎕WE ¯1
      ∇

      ∇A EVALUATE B;RESULT;⎕IO
[1]   ⍝ Evaluate expression B and put it into the Text property of object A
[2]   ⍝ If an error occurs, change the colour of the text
[3]    ⎕IO←1
[4]    RESULT←⎕EC B
[5]    :Select ↑RESULT
[6]    :Case 0   ⍝  Error
[7]      A.ForeColor←A.ForeColor.Red
[8]      A.Text←MAKEVEC 3⊃RESULT
[9]    :Case 1   ⍝   Expression with a result which would display
[10]     A.ForeColor←A.ForeColor.DarkOliveGreen
[11]     A.Text←MAKEVEC⍕3⊃RESULT
[12]   :Else
[13]  ⍝   2 Expression with a result which would not display
[14]  ⍝   3 Expression with no explicit result
[15]  ⍝   4 Branch to a line
[16]  ⍝   5 Naked branch
[17]     A.Text←''
[18]   :End
      ∇
      ∇R←MAKEVEC B
[1]  ⍝ Given a character array, ensure it's a text vector
[2]  ⍝ If matrix or higher rank, make into CRLF-delimited vector
[3]   :If 2>⍴⍴B
[4]     R←B
[5]   :Else
[6]     R←⎕R ⎕BOX B
[7]   :EndIf
[8]   R←⎕SS R ⎕R(⎕R,⎕L)
      ∇
```

Example 2

This function shows a simple example of using the `System.IO` namespace to create a text file and write some text to it. On line [1], it sets up the search path for .Net classes. Lines [2] and [3] fetch the `FileMode` and `FileAccess` enumeration types, which are then used in line [5].

```
      ∇FileWrite;sw;MyFile;FileMode;FileAccess;DateTime
[1]   '.net' ⎕SETUP 'using' 'System' 'System.IO'
[2]   FileMode←'.net' ⎕GETCLASS 'FileMode'
[3]   FileAccess←'.net' ⎕GETCLASS 'FileAccess'
[4]   DateTime←'.net' ⎕GETCLASS 'DateTime'
[5]   MyFile←'.net' ⎕NEW 'FileStream' 'c:\temp\testdotnet.txt'
      (FileMode.OpenOrCreate) (FileAccess.ReadWrite)
[6]   sw←'.net' ⎕NEW 'StreamWriter' MyFile
[7]   sw.Write 'Written from APLX',⎕R,⎕L
[8]   sw.Write 'Created by APLX Version 4 at ',DateTime.Now.ToString
[9]   sw.Close
[10]  MyFile.Close
      ∇
```

Interfacing to Java

Specifying a call to the Java environment

You can interface to Java by supplying `'java'` as the environment string (left argument) for `⎕NEW`, `⎕GETCLASS`, or `⎕CALL`. These system functions will allow you to create an instance of a Java class, or to call a Java static mathod.

Setting Java options

You can set various options for the Java virtual machine by using the system function `⎕SETUP`. For example, you can specify a particular Java Virtual Machine should be used, and set the class library path.

See the documentation on `⎕SETUP` for details on controlling the Java environment.

Conversion of Java data types to APL data

APLX by default applies the following data conversion rules to data returned from Java:

- Any Java numeric types (byte, int, long, float, double, etc) are converted to APL integers or floats. 64-bit integers are converted to APL floats on 32-bit platforms, to APL integers on 64-bit platforms.
- Java booleans are converted to APL binary values 0 or 1
- Java Strings and Chars are converted to APL character arrays, translated from Unicode to APLX internal representation. (Any characters which do not appear in the APLX character set are converted to question marks.)
- Simple Java arrays are converted to APL arrays, with individual elements converted as above.

Anything else is left as an object in the Java environment, and a reference to the object is returned to APL.

There are some special cases to consider. The data might not be convertible at all, or it might lose precision in the conversion. To handle cases like this, APLX provides the `⎕REF` system method. This forces the data to remain as a Java object. You can then call Java methods appropriate to the data type.

An example which cannot be represented at all is where a Java `Double` contains a NaN (Not A Number). APL does not handle NaNs, so it cannot be converted to an APL floating-point value. Instead, NaNs are left as Java objects. If you try to use the data in an APL expression, you will get a DOMAIN ERROR, but you can see that it is a NaN and use Java methods on it.

Using the Java interface from multiple APL tasks

Because it is not safe to call the Java virtual machine from multiple threads, you cannot use the Java interface from more than one APL task at a time. If you try to do so, you will get an error message and a DOMAIN error:

```
      'java' ⎕NEW 'java.util.Date'
Java Virtual Machine (JVM) is already in use by another APL task
DOMAIN ERROR
      'java' ⎕NEW 'java.util.Date'
      ^
```

The lock will be cleared when the APL task which has been accessing Java executes a `)CLEAR`, `)LOAD`, or `)OFF`.

Example

```
      ∇DEMO_TimeZone;date;tzclass;tz;dateFormat;dateList
[1]   ⍝ Demonstration of using a TimeZone object in Java
[2]   ⍝
[3]   ⍝ First create a date
[4]    date←'java' ⎕NEW 'java.util.Date'
[5]   ⍝
[6]   ⍝ What is the date?
[7]    'Result of date.toString: ',date.toString
[8]    ''
[9]   ⍝ To create a TimeZone object we need to call a static
[10]  ⍝ method in the TimeZone class
[11]   tzclass←'java' ⎕GETCLASS 'java.util.TimeZone'
[12]   tz←tzclass.getTimeZone 'America/Los_Angeles'
[13]  ⍝
[14]  ⍝ Could also call the static method directly...
[15]   tz←'JAVA' ⎕CALL 'java.util.TimeZone.getTimeZone' 'America/Los_Angeles'
[16]  ⍝
[17]  ⍝ Does this time zone use daylight savings time?
[18]   'Result of tz.useDaylightTime: ',tz.useDaylightTime
[19]   ''
[20]  ⍝ What is the time zone called with and without daylight savings time?
[21]   'Result of tz.getDisplayName: ',tz.getDisplayName 1(tz.LONG)
[22]   'Result of tz.getDisplayName: ',tz.getDisplayName 0(tz.LONG)
[23]   ''
[24]  ⍝ What is the current date/time in our local time zone?
[25]  ⍝ We create a SimpleDateFormat object to format the date
[26]   dateFormat←'java' ⎕NEW 'java.text.SimpleDateFormat' 'EEE, d MMM yyyy HH:mm
      :ss, zzzz'
[27]   'Today''s local date/time is: ',dateFormat.format date
[28]  ⍝
[29]  ⍝ What's the same date/time in Los Angeles?
[30]   dateFormat.setTimeZone tz
[31]   'In Los Angeles, that''s: ',dateFormat.format date
[32]   ''
[33]  ⍝
[34]  ⍝ Let's make 12 dates with different months
[35]   dateList←date.⎕CLONE 12
[36]   dateList.setMonth((⍳12)-⎕IO)
[37]  ⍝
[38]  ⍝ Format each of these dates for Los Angeles time
[39]   'Here are some more dates with the month changed:'
[40]   12 1⍴dateFormat.format¨dateList
[41]   ∇
```

This produces the following output:

```
Result of date.toString: Tue Nov 20 14:37:36 GMT 2007

Result of tz.useDaylightTime:  1

Result of tz.getDisplayName: Pacific Daylight Time
Result of tz.getDisplayName: Pacific Standard Time

Today's local date/time is: Tue, 20 Nov 2007 14:37:36, Greenwich Mean Time
In Los Angeles, that's: Tue, 20 Nov 2007 06:37:36, Pacific Standard Time


Here are some more dates with the month changed:
 Sat, 20 Jan 2007 06:37:36, Pacific Standard Time
 Tue, 20 Feb 2007 06:37:36, Pacific Standard Time
 Tue, 20 Mar 2007 07:37:36, Pacific Daylight Time
 Fri, 20 Apr 2007 06:37:36, Pacific Daylight Time
 Sun, 20 May 2007 06:37:36, Pacific Daylight Time
 Wed, 20 Jun 2007 06:37:36, Pacific Daylight Time
 Fri, 20 Jul 2007 06:37:36, Pacific Daylight Time
 Mon, 20 Aug 2007 06:37:36, Pacific Daylight Time
 Thu, 20 Sep 2007 06:37:36, Pacific Daylight Time
 Sat, 20 Oct 2007 06:37:36, Pacific Daylight Time
 Tue, 20 Nov 2007 06:37:36, Pacific Standard Time
 Thu, 20 Dec 2007 06:37:36, Pacific Standard Time
```

Interfacing to Ruby

Specifying a call to Ruby

You can interface to Ruby by supplying `'ruby'` as the environment string (left argument) for `⎕NEW`, `⎕GETCLASS`, or `⎕CALL`. These system functions will allow you to create an instance of a Ruby class, or to call a Ruby static mathod.

Loading modules and setting the Ruby DLL search path

Ruby classes are placed in *Modules*. When you use `⎕NEW` (or `⎕GETCLASS` or `⎕CALL`) to refer to a Ruby class, the system searches through the modules which have been loaded into the Ruby interpreter to resolve the class name. If the class is not found, you will get an error:

```
      dt←'ruby' ⎕new 'DateTime'
#<NameError: uninitialized constant DateTime>
DOMAIN ERROR
      dt←'ruby' ⎕new 'DateTime'
         ^
```

The above example has failed because `DateTime` is a class defined in the `Date` module, which has not been loaded. (In a Ruby script, you would get the same error if you had not loaded the `Date` module by using the Ruby `'require'` statement). You can tell the Ruby interpreter to load a module by using the system function `⎕SETUP` and the `'require'` keyword. For example:

```
      'ruby' ⎕SETUP 'require' 'Date'
      dt←'ruby' ⎕new 'DateTime'
      dt
[ruby:DateTime]
```

The effect of using the `'require'` keyword is to add a given Ruby module (or script) to the list in which Ruby will search for class definitions. The parameter is a character vector containing the module name. This can be specified either as a full path name, or as just a file name, in which case Ruby will search in its current search path for the script:

```
      'ruby' ⎕SETUP 'require' 'c:\ruby\myapp.rb'
```

You can use the `'addpath'` keyword to add one or more directories to Ruby's current search path for modules:

```
      'ruby' ⎕SETUP 'addpath' 'c:\rubyapps' 'c:\rubylibs\version2'
```

See the documentation on `⎕SETUP` for more options for controlling the Ruby environment.

Conversion of Ruby data types to APL data

APLX by default applies the following data conversion rules to data returned from Ruby:

- Ruby integer ("Fixnum") values are converted to APL integers.
- Ruby Float and Bignum values are converted to APL floating-point numbers.
- Ruby Booleans (`true` or `false`) are converted to APL binary values 1 or 0
- Ruby Strings are converted to APL character arrays, translated to APLX internal representation.
- Ruby `nil` values are converted to APL Null objects
- Simple Ruby arrays are converted to APL arrays, with individual elements converted as above.

Anything else is left as an object in the Ruby environment, and a reference to the object is returned to APL.

There are some special cases to consider. The data might not be convertible at all, or it might lose precision in the conversion. For example, a Ruby `Bignum` might have a higher precision than an APL double-precision floating point can represent. To handle cases like this, APLX provides the `⎕REF` system method. This forces the data to remain as a Ruby object. You can then call Ruby methods appropriate to the `Bignum` or other data type, to manipulate the data without losing precision.

An example which cannot be represented at all is where a Ruby `Double` contains a NaN (Not A Number). APL does not handle NaNs, so it cannot be converted to an APL floating-point value. Instead, NaNs are left as Objects. If you try to use the data in an APL expression, you will get a DOMAIN ERROR, but you can see that it is a NaN and use Ruby methods on it.

Supplying Boolean arguments to Ruby methods

Ruby methods which expect `true` or `false` as arguments have to be handled in a special way, because Ruby does not allow 1 and 0 as equivalents to Booleans. To work around this, pass a one element matrix (`1 1⍴1` or `1 1⍴0`) as the argument; the interface will recognize this as a special case and convert the data to a Ruby `true` or `false` value.

Using the Ruby interface from multiple APL tasks

Because it is not safe to call the Ruby interpreter from multiple threads, you cannot use the Ruby interface from more than one APL task at a time. If you try to do so, you will get an error message and a FILE LOCKED error:

```
      dt←'ruby' ⎕new 'Date'
This interface cannot be used by more than one APL task at a time
FILE LOCKED
      dt←'ruby' ⎕new 'Date'
         ^
```

The lock will be cleared when the APL task which has been accessing Ruby executes a `)CLEAR`, `)LOAD`, or `)OFF`.

Evaluating Ruby expressions

Because Ruby is an interpreted language, it is possible to use ⎕EVAL to run lines of Ruby code, and for setting up variables in the Ruby environment:

```
      'ruby' ⎕EVAL 's=String.new "Hello there"'
Hello there
      'ruby' ⎕EVAL 's.length'
11
      'ruby' ⎕EVAL 'Math.sqrt(9)'
3
```

Ruby naming conventions

Ruby allows a method names to include various characters (such as = and ?) which are not valid in APL names. Indeed by convention in Ruby, methods which return a Boolean often end with a question mark. For example, the method `leap?` of the Ruby `DateTime` class returns a Boolean indicating whether the date falls in a leap year, and the method `responds_to?` is a standard way of finding out whether a Ruby object supports a given method call (message).

The problem with this is that the APL parser has a different view of what constitutes a valid name. So if you write:

```
    is_leap_year←RUBYDATE.leap?
```

then the APL interpreter will think the rightmost token of the line is the APL ? primitive, separate from the compound identifier `RUBYDATE.leap`. It will therefore give an error.

To work around this problem, the $ character can be used as an escape character in external names. It has the effect of treating the next character as part of the name. So the valid way of calling the `is_leap?` method is:

```
    is_leap_year←RUBYDATE.leap$?
```

Example 1

This example shows the use of the Ruby `Hash` class, which maintains a list of Key - Value pairs.

```
      h←'ruby' ⎕NEW 'Hash'
      h.length
0
      h.store 'France' 'Paris'
Paris
      h.store 'UK' 'London'
London
      h.store 'Italy' 'Rome'
Rome
      h.store 'Germany' 'Berlin'
Berlin
      h.length
4
      h.to_a
  UK London   France Paris   Italy Rome   Germany Berlin
```

```
      h.sort             ⍝ Sort by key values, return array
  France Paris   Germany Berlin   Italy Rome   UK London
      h.key$? 'France'  ⍝ Does key 'France' exist? (Note use of $ escape
character)
1
      h.key$? 'USA'     ⍝ Does key 'USA' exist?
0
      h.fetch 'France'
Paris
      h.fetch 'USA'
#<IndexError: key not found>
DOMAIN ERROR
      h.fetch 'USA'
      ^
```

Example 2

This example shows the use of the Ruby `Complex` class, for manipulating complex numbers:

```
      'ruby' ⎕setup 'require' 'complex'
      compclass←'ruby' ⎕GETCLASS 'Complex'
      a←⎕NEW compclass 3 4
      a.⎕DS
3+4i
      b←⎕NEW compclass 2 ¯1
      b.⎕DS
2-1i
      b.conjugate
[ruby:Complex]
      b.conjugate.⎕DS
2+1i
      b.polar
2.236067977 ¯0.463647609
      c←a.$* b    ⍝ Complex multiplication. Note use of $ escape char
      c.⎕DS
10+5i
```

Interfacing to the R statistical language

What is R?

R is an open-source language and set of packages aimed principally at statistical analysis. It includes a huge library of pre-written statistical and mathematical routines, which can be accessed immediately and very conveniently from APLX. It also includes mathematically-oriented graphing facilities.

R is available from http://www.r-project.org, which describes R as follows:

> *R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.*
>
> *R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.*
>
> *R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.*

Installing R

R can be downloaded either in source code form, or as a pre-compiled binary for most popular platforms, from a number of wesbites (see http://www.r-project.org). In each case you need the R shared library (called `libR.so` in Linux, `R.dll` under Windows, and `libR.dylib` under MacOS); this is usually available in the pre-compiled binaries. If installing from source, be sure to specify the option `--enable-R-shlib` when running the configure script.

Installing under Windows

This is most easily done using the installer provided with the pre-built binaries. The only additional step which you might need to take is to add the R binary directory to your search path, so that APLX can find the DLL `R.dll`.

Installing under Linux and MacOS

Follow the instructions provided with the R download. You also need to set up environment variables for R; this is usually done in the R script.

Calling R from APLX

Most of the interface between APLX and R is done using a single external class, named `'r'`, which represents the R session that you are running. (Note that this is different from most of the other external class interfaces, where objects of many different classes can be created separately from APLX). You create a single instance of this class using `⎕NEW`. R functions (either built-in or loaded from packages) then appear as methods of this object, and R variables as properties of the object.

For example:

```
      ⍝ Open the R interface and try a few simple things
      r←'r' ⎕new 'r'
      r.sqrt 2
1.414213562
      r.sqrt (⊂⍳5)
1 1.414213562 1.732050808 2 2.236067977
      r.sqrt ¯1
[r:NAN]                  ⍝ Returns a special R object NAN
      r.mean (⊂⍳10)
5.5
```

When calling R functions, the APLX right argument is always a vector where each element corresponds to one argument of the R function. The calls to the `sqrt` and `mean` functions above illustrate this; to pass an array as the argument, it needs to be enclosed.

Creating variables in the R environment

Assigning to a symbol as though it were a property of the R session class creates a variable in the R world:

```
      r.x←2 3⍴⍳6         ⍝ x is an R variable
      r.x
1 2 3
4 5 6
      r.x.⎕ref
[r:matrix]
```

Evaluating R expressions

Because R is an interpreted language, it is possible to use the System Function `⎕EVAL` to run lines of R code, for setting up variables in the R environment, for defining R functions, and so on.

```
       'r' ⎕eval '4:9'
4 5 6 7 8 9
```

However, a more convenient syntax is provided (for the 'r' class only) in which `⎕EVAL` is a monadic system *method*.

The right argument is a text vector containing any expression which is a valid line of R code. The result is the explicit result (if any) of evaluating the expression in the external environment. For example:

```
      r←'r' ⎕new 'r'
      r.x←2 3⍴⍳6         ⍝ x is an R variable
      r.x
1 2 3
4 5 6

      r.⎕eval 'x[2,]'
4 5 6
      r.⎕eval 'mean(x[2,])'
5
```

Note that the last line could be executed using the alternative syntax where `⎕EVAL` is a system *function*:

```
      'r' ⎕eval 'mean(x[2,])'
5
```

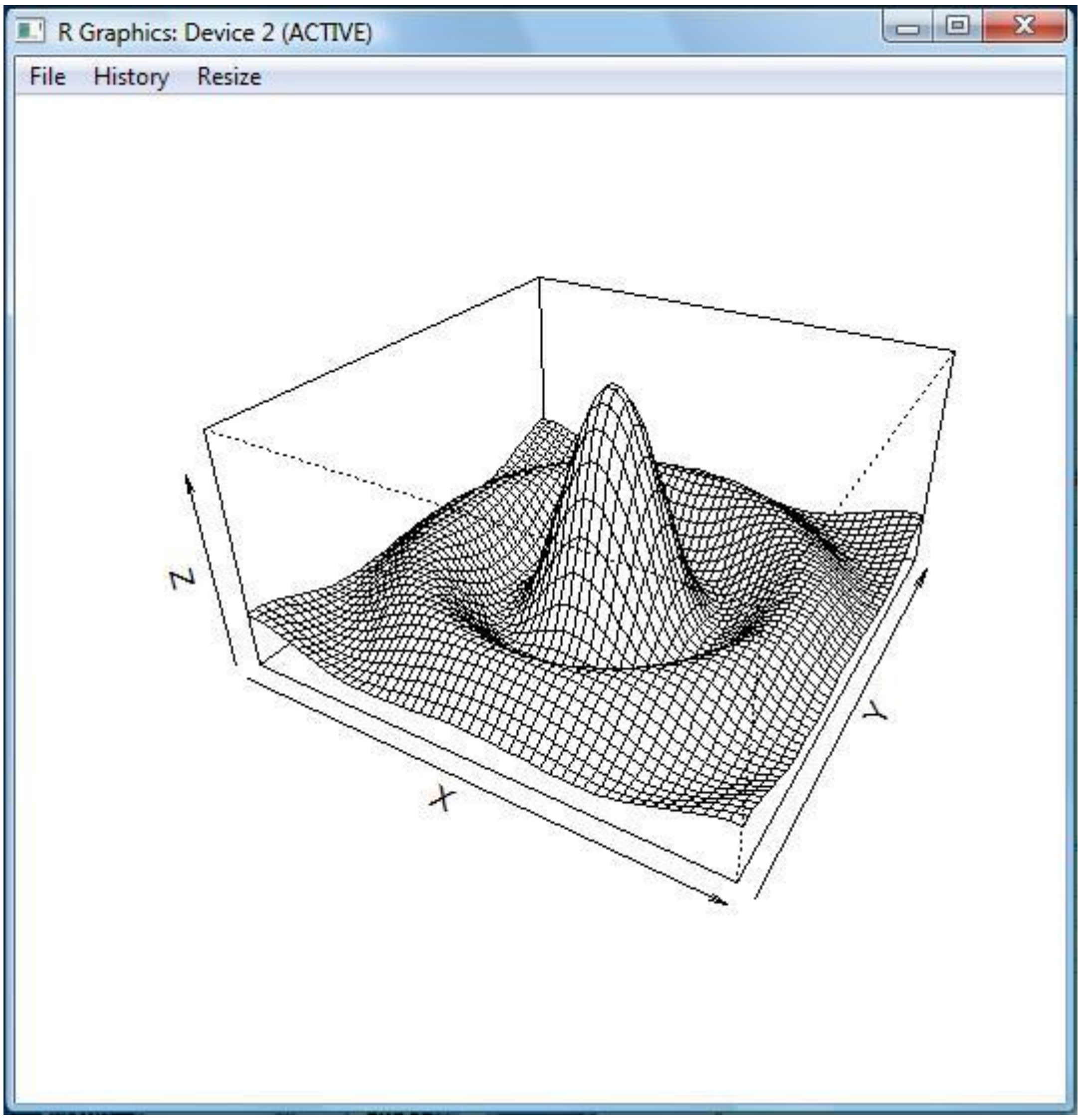
Example: 3-D plot

In this short but complete example (based on an article by Skomorokhov and Kutinsky from Quote Quad 123 No 4), we create some data in the R environment, define an R function, and run the R outer product to create some test data. We then call the R `persp` function to create a 3-D plot:

```
      r←'r' ⎕new 'r'
      x←r.⎕eval 'seq(-10,10,length=50)'
      y←x

      ⍝ Define an R function and return a reference to it:
      fn←r.⎕eval 'foo<-function(x,y){r<-sqrt(x^2+y^2);10*sin(r)/r}'
      fn
[r:function]
      r.z←r.outer(x y fn)
      r.x←x
      r.y←y
      ⊣r.⎕eval
'persp(x,y,z,theta=30,phi=30,expand=0.5,xlab="X",ylab="Y",zlab="Z")'
```

This causes R to open a window and display a 3-d perspective chart:



Listing R variables and functions

The ⎕NL system method can be used to get the names of R variables and/or functions. The function list includes built-in functions and functions from all the loaded R packages, so may be several thousand items long:

```
      ⍝ List R variables:
      vars←r.⎕nl 2
      ⍴vars
129 21
      ⍝ List R functions:
      fns←r.⎕nl 3
      ⍴fns
2058 34                    ⍝ There are lots of them!
```

⎕DESC can be used to get the full R function list together with details of the parameters (Caution: the result is very large):

```
      fns2←r.⎕desc 3
```

```
    fns2[1445+⍳5;]
pwilcox (q, m, n, lower.tail = TRUE, log.p = FALSE)
q (save = "default", status = 0, runLast = TRUE)
qbeta (p, shape1, shape2, ncp = 0, lower.tail = TRUE, log.p = FALSE)
qbinom (p, size, prob, lower.tail = TRUE, log.p = FALSE)
qbirthday (prob = 0.5, classes = 365, coincident = 2)
```

R naming conventions

R function names can have characters such as a `<` and `-` in them, which are not legal as symbol names in APLX. To call these in APLX as direct method calls, you need to escape the illegal character with a `$` character. (This is not of course necessary when using `⎕EVAL`, where the string is passed as-is to R).

For example, to call `attr<-` from APLX, you would call `r.attr$<$-`.

Conversion of R data types to APL data

Simple numeric arrays and arrays of strings passed from APLX to R are converted directly to the R equivalent array, and are converted back automatically ('unboxed') when referenced or returned from an R function call, unless you use `⎕REF` to force an object reference to be returned:

```
      r.y←2.2 3.3 4.4

      r.y
2.2 3.3 4.4
      r.y.⎕ref
[r:numeric]
      (r.y.⎕ref).⎕ds    ⍝ Use R to format the R array
[1] 2.2 3.3 4.4
      r.⎕eval 'mean(y)'
3.3
```

Complex, NA and NAN data types

The APLX R interface defines three special object classes for NA ('Not Available'), NaN ('Not A Number') and complex-number data, which R routines may return, or which you may want to pass as arguments into R functions.

For example, the following R expression returns a complex number:

```
      c←r.⎕eval '3+4i'
      c
[r:complex]
      c.format
3+4i
```

Instances of these object classes can be created by using `⎕NEW`:

```
      NA←'r' ⎕new 'NA'
      NA
[r:NA]
      NAN←'r' ⎕new 'NAN'
      r.z←55.6 77.4 NAN 81 NA
      r.z
55.6 77.4 [r:NAN] 81 [r:NA]
      r.sqrt (⊂r.z)
7.456540753 8.797726979 [r:NAN] 9 [r:NA]
```

The `complex` class allows you to create either a single complex number, by using a constructor with two numbers for real/imaginary parts:

```
      c←'r' ⎕new 'complex' 2 3
      c
[r:complex]
      c.format
2+3i
```

or to build an R complex array by passing an array of length-2 vectors of the real and imaginary parts of each complex number:

```
      m←'r' ⎕new 'complex' (3 2⍴(1 2) (3 4) (5 6) (7 8) (9 10) (11 12))
      m
[r:matrix]
      m.format
 1+ 2i  3+ 4i
 5+ 6i  7+ 8i
 9+10i 11+12i
```

You can access or specify the real and imaginary parts directly using the pseudo-properties `real` and `imag` of the `complex` object:

```
      m.real
1  3
5  7
9 11
      m.imag←3 2⍴.1×⍳6
      m.format
 1+0.1i  3+0.2i
 5+0.3i  7+0.4i
 9+0.5i 11+0.6i
      m.imag
0.1 0.2
0.3 0.4
0.5 0.6
```

NAs and NaNs are also supported in Complex arrays:

```
      v←'r' ⎕new 'complex' ((3.2 3.4) NA (1.1 8.2))
      v.format
3.2+3.4i        NA 1.1+8.2i

      v.real
3.2 [r:NA] 1.1
      v.imag
3.4 [r:NA] 8.2
      (r.sqrt v).format
1.983563+0.857043i                   NA 2.164885+1.893865i
```

Advanced R data types

Other R types, such as factors and lists, are left 'boxed up' as references to the underlying R object (unless you use ⎕VAL to force an unbox, if this is possible):

```
      lst←r.⎕eval 'list(name="Fred",age=99)
      lst
[r:list]
      lst.⎕val
 Fred  99
      ⎕display lst.⎕val
```

An object which is still boxed up can be passed as an argument to an R function:

```
      r.length lst
2
      r.names lst
 name age
```

As a convenience you can also write this last example as:

```
      lst.length
2
     lst.names
 name age
```

This works because APLX treats the expression

```
        obj.function arg1,arg2,...
```

...as equivalent to:

```
        r.function   obj,arg1,arg2,...
```

Examining an object with ⎕DS

The system method ⎕DS can be used to examine an R object. It's equivalent to calling the `print` method when working in an interactive R session.

```
      lst←r.⎕eval 'list(name="Fred",age=99)
      lst
[r:list]
      lst.⎕ds
$name
[1] "Fred"

$age
[1] 99
```

Functions on the left side of an R assignment

In R, a function name can sometimes be given on the left side of an R assignment as the fourth line of the following example written in the R language shows:

```
      > lst<-list(name="Fred",age=99)
      > names(lst)
      [1] "name" "age"
      > names(lst)<-c("firstname", "age")
      > names(lst)
      [1] "firstname" "age"
```

What actually happens 'under the hood' is that R treats an assignment like:

```
      function(obj) <- value
```

...as being a call to a function called "function<-" with the function result assigned to the object, i.e.

```
      obj  <-  "function<-" (obj, value)
```

If you wanted to call this function in APLX you could do so, using the $ character to escape the function name:

```
      lst←lst.names$<$- (⊂'firstname' 'age')
      lst.names
 firstname age
```

However, APLX also supports a much more convenience syntax:

```
        lst.names←'firstname' 'age'
```

Indexing lists by name

In the R language a list can be indexed either by number or by name, e.g.

```
      > lst[[2]]
      $age
      [1] 99

      > lst$age
      [1] 99
```

This is achieved by special R indexing functions called [[and $ which can also be called from APLX (once again using a $ to escape the function name):

```
      lst.$[$[ 2
99
      lst.$$ 'age'
99
```

It is also possible to change the value of a list item, which you would do in R by writing "`lst$age<-95`". Under the hood, R is using a function called `$<-` which we can call from APLX:

```
      lst←lst.$$$<$- 'age' 95
```

Attributes

R objects can have *attributes* attached to them. By convention, any reference to `∆XXX` is interpreted as an implicit call to `attr(obj, XXX)`:

```
      ⍝ Get a copy of the R 'Iris' variable, a sample 'data.frame'
      iris←r.iris
      iris
[r:frame]
      (iris.attributes).names
 names row.names class
      iris.∆names
 Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

You can also change the value of attributes or add your own. Any assignment to `∆XXX` is interpreted as an implicit call to `attr<-(obj, XXX)`:

```
      f.∆mycustomatt ← 'Some attribute'
      f.∆mycustomatt
Some attribute
      ⍝ Longer-winded way of doing the same thing, but creating a new object:
      f2←r.attr$<$- f 'mycustomattr' 'Some other attribute'
      r.attr f2 'mycustomattr'
Some other attribute
```

Here is an example of creating an R data.frame object from some APL data:

```
      data←?3 5⍴100      ⍝ Random APL data for demo
      data
95  6  77 78 83
13  2  69 87 63
74 73 100 89 24

      frame←r.data.frame (⊂data)
      frame.attributes.⎕ds
$names
[1] "X1" "X2" "X3" "X4" "X5"

$row.names
[1] 1 2 3

$class
[1] "data.frame"


      frame.∆names←'Fish' 'Chips' 'Ham' 'Eggs' 'Tea'

      frame.⎕ds
  Fish Chips Ham Eggs Tea
1   95     6  77   78  83
2   13     2  69   87  63
3   74    73 100   89  24
```

```
      frame.summary.⎕ds
      Fish           Chips           Ham             Eggs            Tea
 Min.   :13.00   Min.   : 2.0   Min.   : 69.0   Min.   :78.00   Min.   :24.00
 1st Qu.:43.50   1st Qu.: 4.0   1st Qu.: 73.0   1st Qu.:82.50   1st Qu.:43.50
 Median :74.00   Median : 6.0   Median : 77.0   Median :87.00   Median :63.00
 Mean   :60.67   Mean   :27.0   Mean   : 82.0   Mean   :84.67   Mean   :56.67
 3rd Qu.:84.50   3rd Qu.:39.5   3rd Qu.: 88.5   3rd Qu.:88.00   3rd Qu.:73.00
 Max.   :95.00   Max.   :73.0   Max.   :100.0   Max.   :89.00   Max.   :83.00
```

Using the R interface from multiple APL tasks

Because it is not safe to call the R interpreter from multiple threads, you cannot use the R interface from more than one APL task at a time. If you try to do so, you will get an error message and a FILE LOCKED error:

```
      r←'r' ⎕new 'r'
This interface cannot be used by more than one APL task at a time
FILE LOCKED
      r←'r' ⎕new 'r'
         ^
```

The lock will be cleared when the APL task which has been accessing R executes a `)CLEAR`, `)LOAD`, or `)OFF`.

Custom interfaces

Specifying a call a custom object-oriented interface

If the environment string (left argument) for `⎕NEW`, `⎕GETCLASS`, or `⎕CALL` is not one of the standard options described in previous sections, APLX will attempt to load a shared library to implement the interface. The name of the shared library it tries to load will be `aplxobj_XXX.dll` (under Windows), or `aplxobj_XXX.bundle` (under MacOS), where XXX is the environment string supplied as the left argument.

This makes it possible to extend APLX to interface to:

- Other modern object-oriented languages which support *reflection*, such as Python.
- Custom libraries or application interfaces which can be called from a shared library and where an object-oriented interface makes sense. For example, a class library written in C++ could be packaged up is this way.

If you would like more information on writing custom interfaces, or if you would to commission MicroAPL to write an interface on your behalf, please contact MicroAPL.

Auxiliary Processors

The Auxiliary Processor (AP) mechanism

It is sometimes necessary to write parts of an APL-based application in a low-level procedural language such as C, for example to speed up a critical routine, to re-use an existing library of subroutines, or to access some part of the system like a filing system for which no APL- or object-based interface is provided. Such a piece of code is known as an *Auxiliary Processor*. It is accessed through the *Shared Variable* interface described here.

(See also `⎕NA` which may provide a more convenient and flexible alternative.)

Dynamic Binding

Auxiliary Processors in APLX are held as separate disk files and are loaded by APLX when required using a mechanism known as *Dynamic Binding*. This is a mechanism used by many modern programs which call standard library routines. Instead of the library routines being permanently linked to the program when it is created, the operating-system loader delays performing the linking until the program is executed.

As an example, suppose you write a program in C which calls the C library function `printf()`. On systems which don't support dynamic binding, you would compile the main program and then link it to `printf()` to create an executable file. Using dynamic binding, the routine `printf()` is not automatically linked; instead when you run the program the loader binds it to the current version of the `printf()` library routine. A similar approach is used in Windows for calling the operating system or other libraries. The routines which are called are held in Dynamic Link Libraries (DLLs).

APLX Auxiliary Processors are treated in a similar way. They are dynamically bound to APL during execution. The only difference is that binding doesn't occur when APL is first executed; it is further delayed until APLX is explicitly instructed to go and load the routine from disk and bind it. The processor is loaded into user memory outside the APL workspace, and so has no impact on the available workspace size.

To load an Auxiliary Processor written in another language, it must be written to be a shared library, and must be compiled to the appropriate format (for example, under AIX it must be in IBM's XCOFF format, and in Windows it must be a valid DLL). Most modern compilers will be able to produce shared libraries in this way. Examples in this chapter are written in C.

Loading and Unloading Auxiliary Processors

Binding of Auxiliary Processors is carried out through APL Shared Variable interface. To load a module from disk and bind it, you can use the APL shared variable offer function `⎕SVO` to share a variable with Auxiliary Processor 3 (AP3), and then assign to it the name of the module to load:

```
      3 ⎕SVO 'PROC'             Share a variable called PROC
      PROC←'myprocessor'        Load the AP called myprocessor
```

If the specified processor name includes a full path description, an attempt is made to load from the specified directory. If the name does not include a path element, APLX uses a library path to locate the module. The exact behavior varies according to the host system:

- **AIX:** The library search path for external modules can be set in the APLX resource database by specifying the resource 'ap_libpath'.
- **Linux:** The search path can be set in the environment variable LD_LIBRARY_PATH; if it is not found there, Linux looks in the list of libraries specified in /etc/ld.so.cache, and finally in /usr/lib, followed by /lib.
- **Windows:** Windows looks in the directories specified in the PATH environment variable. If you omit the file extension in the processor name, Windows will assume the default file extension ".dll"
- **MacOS:** Under MacOS, APLX supports two different file formats for auxiliary processors. These are Code Fragment Manager 'shared libraries', and the 'bundles' introduced in MacOS X. Shared libraries are files of type 'shlb' containing a 'cfrg' resource which identifies the shared library. APLX looks for these first in the 'processors' folder of the APLX installation, and then in the 'bin' folder of the APLX installation (i.e. the folder where the APLX executable resides). You cannot use a full path for a shared library. Alternatively, if you use a MacOS X 'bundle', it *must* be referred to by a full path.

As a convenience, instead of the two-step share procedure using AP3 outlined above, you can directly load an auxiliary processor using any processor number N over 100. The name of the processor to load will be ap*N*. For example, to load a processor called "ap124":

```
      124 ⎕SVO 'CTL'           Share a variable called CTL and load ap124
```

We recommend that processors loaded directly using this method should be placed in the 'processors' directory of the APLX installation, although you can also place them in the host-specific search path described above. The name of the file containing the Auxiliary Processor would be 'ap124' under AIX or Linux, or 'ap124.dll' under Windows. Under MacOS, it could be either a shared library called 'ap124' (of type 'shlb' with a 'cfrg' resource identifying the library as 'ap124'), or alternatively a MacOS X bundle of name 'ap124.bundle'.

To unload a bound processor and free up the memory it uses, simply erase the shared variable (or retract the share using `⎕SVR`).

Note that all auxiliary processors are automatically unloaded when the user executes a)CLEAR or an)OFF.

APLX Auxiliary Processor Calling Conventions

APLX accesses the auxiliary processor through a shared variable; you can assign data to the processor and get back results using normal APL variable assignment and referencing syntax:

```
        PROC←DATA
        R←PROC
```

The auxiliary processor has only one entry point which is called from APLX. The entry point must be called `processor`, and must be an exported routine name. It is called in four different circumstances:

- As soon as the processor is loaded using `⎕SVO`
- When data is assigned to it (shared variable specified)
- When data is read from it (shared variable referenced)
- Just before it is unloaded

The syntax of the main routine of the processor, expressed in C notation, is:

```
int processor (
      int           op,                 // Operation code AP_xxx
      WS_Entry      *data,      // Pointer to APL data specified, or where
                                // to put result
      APLINT        wsfree,     // Free workspace (in bytes) on AP referenced
      void          *scratch,   // Pointer to 256-byte scratch area
                                // for use by the AP
      void          *wsbase,    // Pointer to base of workspace
      WS_Entry      *last_var, // On reference, pointer to header of last var
                                // specified, or NULL if it was temp
      APLINT        tie,        // An internal tie number uniquely
                                // identifying shared variable
      ExportedProcs *exports    // A structure containing addresses of
                                // routines which the AP can call
);
```

Note: `APLINT` is a signed 32-bit integer for 32-bit versions of APLX, and a signed 64-bit integer for APLX64.

The op parameter is the reason for the call, and is one of:

```
      AP_LOADED
      AP_SPECIFIED
      AP_REFERENCED
      AP_UNLOADED
```

The include file **aplx.h** contains the appropriate definitions.

Note for users upgrading from APL.68000 Level II: This function prototype has changed somewhat, as there are now some extra parameters. APs written using the `SHAREC` mechanism of *APL Level II for MacOS* will need to be modified and re-compiled. APs written for *APL Level II for the RS/6000* can still be used unchanged, as the old mechanism is still supported under AIX.

Auxiliary Processor data

Be aware that you can have the same auxiliary processor bound to more than one shared variable at one time. It is important to understand that the *same* copy of the processor is bound in each case, and that the processor is not finally unloaded from memory until all of the shared variables which use it have been erased. If the processor is designed to be used in more than one simultaneous instance, the programmer must be careful to associate any static data retained by the processor with the tie number by which it is being accessed, or to use the 256-byte scratch area to hold data associated with the instance. You can, of course, allocate more memory for use by the AP, as long as you free it again when the AP is unloaded.

Return value

The explicit result of the routine is an APL error code. 0 means no error. Error codes are defined in the **aplx.h** include file.

If the result returned is `APL_NOMESSAGE` (`¯4`), then the system assumes that the processor has already displayed the error message. The system will therefore display only the line in error and the caret.

Internal Structure of an Auxiliary Processor

The internal structure of the auxiliary processor can be anything which makes sense to the programmer. The processor can call other library routines (which are dynamically bound when the processor is loaded), allocate and deallocate memory, fork other processes, etc. The processor can also take over signal handling, but it should restore APL's signal handlers before returning.

Typically, the processor runs a routine when it is loaded to allocate any resources it needs. The read and write data routines do the real work, and the unload routine is used to deallocate resources just before the processor is unloaded from memory.

In C, the main routine might be a simple switch statement:

```
#include "aplx.h"      /* Include the APLX header equates */

int processor (int op, WS_Entry *data, APLINT ws_free, void *scratch,
                    void *wsbase, WS_Entry *last_var,
                    APLINT tie, ExportedProcs *exports)
{
     int result;

     switch (op) {

     case AP_LOADED:
         result = load_routine (tie);
         break;
     case AP_UNLOADED:
         result = unload_routine (tie);
         break;
     case AP_SPECIFIED:
         result = specify_routine (data, tie);
```

```
            break;
        case AP_REFERENCED:
            result = reference_routine (data, wsfree, last_var, tie);
            break;
        default:
            return APL_SYSERR;    /* Can't ever happen */
        }
        return result;            /* Explicit result is APL error number */
}
```

The question of when to do the real work - on the specify routine or the reference routine - will depend on the application.

For example, a processor which calculates the mean and standard deviation of a set of numbers might do the computation once when the data is specified, and store the results in static data to be picked up on the reference routine. This is the simplest construction.

However, a routine which performs a transformation on a matrix has a choice: (a) store the matrix data on the specify routine and do the transformation when the processor is referenced, or (b) perform the transformation when the data is specified and store the result.

In the latter case, where the original matrix is stored in an APL variable, a pointer to it is passed to the reference routine in the argument last_var, making it possible to omit the data storage in (a) above. If the original data specified was not a variable but constant data, it no longer exists when the reference routine is called, and a NULL pointer is passed.

Format of an APLX workspace entry

When data is written to the processor the pointer to the workspace entry header is passed as one of the parameters to the called routine. Similarly, when data is read a pointer to the result area where the processor must build the result header is passed. We recommend that you use the `WS_Entry` structure defined on **aplx.h** to access the fields. (Note that all data is held in the natural byte-order for the machine in question. Thus, for little-endian processors like the x86, the actual bytes in memory within each 2- 4- or 8-byte field are held backwards). The structure of the header is as follows:

The first four bytes are used internally by APL and are not important to the person writing the processor, except that the field must be accounted for when returning a result. The next four bytes (or 8 bytes in APLX64) are a length field (`we_length`), which is the total length of the variable in bytes. (APL automatically rounds this up to be a multiple of 4 or 8). The next byte of the header is a one-byte field (`we_type`) describing the variable type:

DT_CHR	0x05	Binary data
DT_INT	0x07	Integer data
DT_FLT	0x09	Float data (IEEE 64-bit format)
DT_CHR	0x0B	Character data
DT_OBJ	0x0E	Object- or class-reference
DT_NST	0x0F	Nested or mixed data

Equates for the data types `DT_xxx` and for the `WS_Entry` data structure are defined in **aplx.h**.

The next byte is an unused padding byte, followed by a two-byte field (`we_rank`) containing the rank of the data times 4, with zero indicating that the data is a scalar. Following the rank field are multiple four (or eight) byte rho entries. After the rho entries, the data begins.

Data Formats

Binary data is stored as individual bits in left justified 32-bit or 64-bit fields, depending on the version of APLX. In a 32-bit APL, the binary vector `1 0 1 1 0 1 1 1 1 0 1` would be stored as follows in a 32 bit block (shown in Hexadecimal): `B7A00000`. The entire workspace entry would look like:

```
00000000     00000014     05000004     0000000B     B7A00000
(Reserved)   (Length)     (Type-Rank)  (Rho)        (Data)
```

Integer data is stored in 32-bit (or 64-bit) signed blocks. The 32-bit workspace entry for the integer vector `10 ¯2` would look like:

```
00000000     00000018     07000004     00000002     0000000A     FFFFFFFE
(Reserved)   Length)     (Type-Rank)   (Rho)        (Data)
```

Float data is stored in the 64 bit IEEE floating Point Standard:

```
     Bit     63        -    Sign of Mantissa, (1=negative)
     Bits    62-52     -    Exponent biased by decimal 1024
     Bits    51- 0     -    Mantissa, binary, normalized, with the MSB
                            not stored and assumed 1
```

The 32-bit workspace entry for the floating scalar `256.5` would be:

```
00000000     00000014     09000000     40700800     00000000
(Reserved)   (Length)     (Type-Rank)  (Data)
```

Character data is stored as 1 character per byte, with possibly unused pad characters to keep the character count a multiple of 4. The 32-bit workspace entry for the character vector `HIT` would look like:

```
00000000     00000014     0B000004     00000003     68697400
(Reserved)   (Length)     (Type-Rank)  (Rho)        (Data)
```

Object- or class-references (data type DT_OBJ) are a special case. The data which follows is an integer representing the index into the table of objects which APLX keeps in the workspace, or 0 for a NULL object. Auxiliary processors cannot make any use of this data, so you should report an error if an object reference is passed to an AP. Equally, do not pass back back data of type DT_OBJ (other than NULL objects) to APLX from an AP.

Mixed or nested data is stored in a potentially recursive format. Following the header fields (Reserved, length, type-rank, rho) are the appropriate number of 4 (or 8) byte pointers (giving an offset from the start of the workspace entry) to individual sub-arrays, which are held within the overall data portion.

```
        VAR←2⍴10 'ABCD'
```

```
00000000     0000003C     0F000004     00000002     00000018     00000028
(Reserved)   (Length)    (Type-rank)   (Rho)        (Pointers)

00000000     00000010     07000000     0000000A
(Reserved)   (Length)     (Type-rank)  (Data)

00000000     00000014      0B000004     00000004     61626364
(Reserved)  (Length)      (Type-rank)  (Rho)        (Data)
```

Programmers should note that if data is inherently capable of being represented as simple data it must be. Thus a simple shape 2 2 matrix must be held as a simple array.

Prototype

Empty nested arrays must have a prototype entry appended. The overall length must account for the length of the prototype. The prototype entry follows the main entry and must contain only 0s or space characters.

```
        VAR←⊂2 2⍴⍳4
        X←0↑VAR

00000000     00000028     0F000004     00000000     00000000
(Reserved)   (Length)     (Type-rank)  (Rho)        (Reserved)

00000018     05000008     00000002     00000002     00000000
(Length)     (Type-rank)  (Rho)        (Rho)        (Data)
```

(All of the above examples are shown in big-endian order, for 32-bit systems.)

Using APLX Exported Routines

A number of useful routines in the APLX interpreter are available to be called by the auxiliary processor. A structure containing pointers to these routines is passed as a parameter to the AP, in the `ExportedProcs` structure defined in **aplx.h**

Exported routines include :

```
char fontin (char font_char);   /* Translate character from font representation to ⎕AV*/

char fontout (char apl_char);   /* Translate character from ⎕AV to font representation*/

int check_event_attention (void); /* Call APL attention check. Returns non-zero
                                     if attention hit */
```

Note that if the processor retains control for a long period (more than a second) without returning to APL, it should call `check_event_attention` periodically. This routine checks whether the Interrupt key has been hit, and also handles user interaction with the windowing interface. If attention checking is not performed for a long period, the user will see this as a window which does not respond to resize events, menu selections, etc.

An Example Auxiliary Processor

The following (supplied as `sample_ap.c` with APLX) is a simple but complete Auxiliary Processor which calculates the exclusive OR of the bytes in a character vector. Note the use of the `ascii_line_send()` routine (via the pointer in the `ExportedProcs` structure) to send a messages to the APL console.

```
//--------------------------------------------------------------------------
//   Sample AP for APLX
//--------------------------------------------------------------------------

// *** MAKE SURE YOU DEFINE LITTLE_ENDIAN IF COMPILING FOR WINDOWS OR OTHER
// *** x86 PLATFORM.  THIS MUST BE DONE BEFORE INCLUDING aplx.h
//
// *** MAKE SURE YOU DEFINE APL64 IF COMPILING FOR APLX64
// *** ON A 64-BIT PLATFORM.  THIS MUST BE DONE BEFORE INCLUDING aplx.h
//
// For AIX, compile with:
//
//  cc -o ap103 -G -Iusr/lpp/aplx/processors -qcpluscmt sample_ap.c
//
//  (The -G option tells the linker to create a shared library.)
//
//  This should create an output file 'ap103'.
//
// For Linux, compile with:
//
//  gcc -o ap103 -shared -Iusr/local/aplx/processors sample_ap.c
//
//  This should create an output file 'ap103'.
//
//  For Windows, you need to add a DLL entry point:
//
//   int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
//   {
//          return 1;
//   }
//
//   You also need to export the 'processor' function by declaring it
//   as type:  __declspec(dllexport)
//
//   Compile the AP as a DLL, and ensure it is called ap103.dll
//
//  For MacOS, build as a shared library or bundle, for example using the
//  Metrowerks Code Warrior or Apple Project Builder.
//
// Once you have built the AP, place it in the 'processors' directory of
// the APLX installation, and test it in APLX with:
//
//      103 ⎕SVO 'DATA'
// 2                              <-- If it doesn't return 2, AP not found
//      DATA ← ⎕A                 Give it a character vector
//      ⎕AF DATA                  See what it returns
// 27
//


// Leave this undefined for MacOS (PowerPC) and AIX.
// Define it for Windows, x86 Linux and Mac Intel.
// #define LITTLE_ENDIAN    1
// Leave this undefined for 32-bit versions of APLX.
// Define it for APLX64
// #define APL64            1

#include "aplx.h"

int processor (
        int           op,         /* Operation code AP_xxx */
        WS_Entry      *data,      /* Pointer to APL data specified,
                                     or where to put result */
        APLINT        wsfree,     /* Free WS (in bytes) on AP referenced */
        void          *scratch,   /* Pointer to 256-byte scratch area for
                                     use by the AP */
        void          *wsbase,    /* Pointer to base of workspace */
        WS_Entry      *last_var,  /* On reference, pointer to header of last
                                     var specified, or NULL if it was temp. */
        APLINT        tie,        /* An internal tie number uniquely
                                     identifying shared var */
        ExportedProcs *exports)   /* Structure containting exported routines
                                     you can call */
{
    int        result;            /* Error code to return to APL */
    char       xor = '\0';        /* The XOR result */
    APLINT     i;                 /* Counter */

    switch (op) {

        case AP_LOADED:
            result = APL_GOOD;
            break;

        case AP_UNLOADED:
            result = APL_GOOD;    /* Nothing to do */
            break;

        case AP_SPECIFIED:
            if (data->we_type_rank == CHR_VEC) {
                for (i = 0; i < data->we_vec_length; i++)
                    xor ^= data->we_vec_cdata[i];
                ((char *)scratch)[0] = xor;   /* Save result in scratch area */
                result = APL_GOOD;
            } else {
                result = APL_DOMERR;
            }
            break;

        case AP_REFERENCED:
            if (wsfree < 256)
                return APL_WSFULL;
            data->we_reserved = 0;            /* Plug 0 in reserved field */
                       /* total length of result: */
            data->we_len = offsetof(we_vec_idata, WS_Entry);
            data->we_type_rank = CHR_SCL;     /* type and rank  of result */
            data->we_scl_cdata = ((char *)scratch)[0];/* return saved result */
            result = APL_GOOD;
            break;
        }
        return result;
    }
```

Section 10: Performance Profiling

Performance Profiling

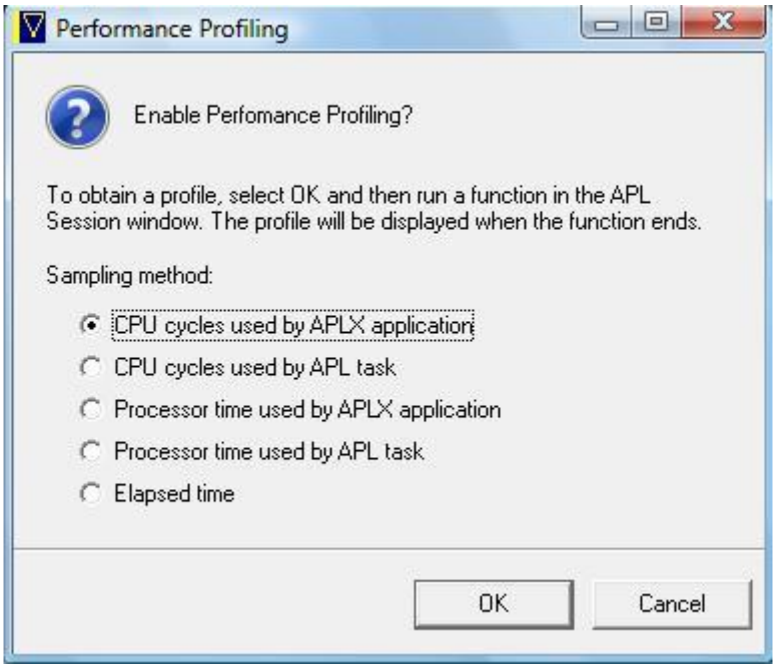
Overview

Performance profiling can be used to find out which parts of your APL code take the most time to execute, or are executed most often, and so helps you to determine which functions to concentrate on when optimising performance. You can view the performance data in a number of different ways, and easily 'drill down' to get more detail on exactly where execution time is spent. You can either use the very easy menu-based profiling described below, or for more detailed control use `⎕PROFILE`.

Profiling using the Tools menu

For simple profiling you can enable profiling through the APLX Tools menu. You then run the code to be profiled. When the code completes and APLX returns to desktop calculator mode, the profile is automatically shown in a Profile window.

When you select 'Performance Profiling' from the Tools menu, APLX brings up a dialog which offers you a choice of different methods for measuring the execution time:



Enable Perfomance Profiling?

To obtain a profile, select OK and then run a function in the APL Session window. The profile will be displayed when the function ends.

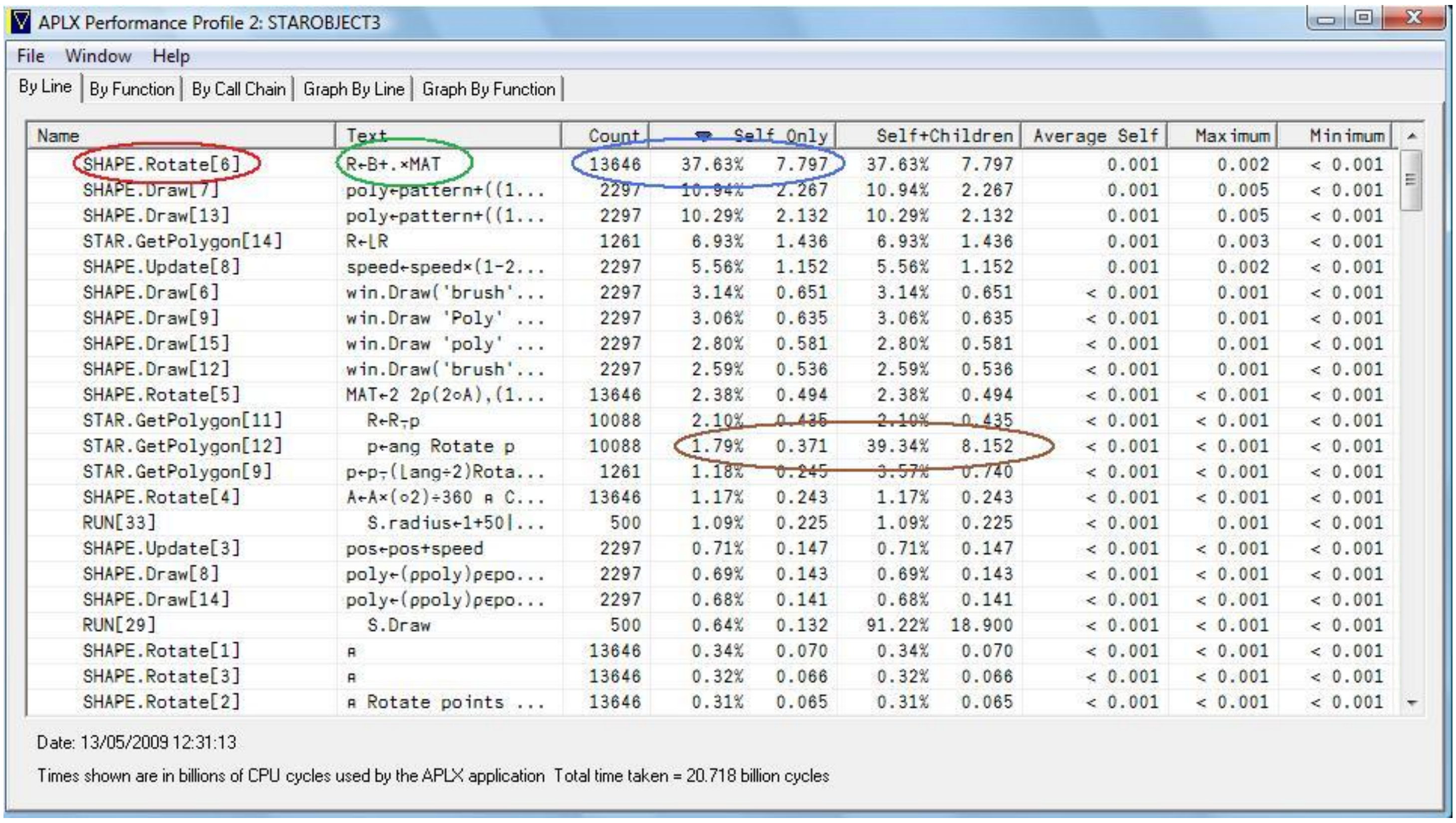
Sampling method:

- [✓ CPU cycles used by APLX application]
- CPU cycles used by APL task
- Processor time used by APLX application
- Processor time used by APL task
- Elapsed time

Depending on which platform you are using, one or more of the timing methods may not be available. For example, earlier versions of Windows cannot measure the number of CPU cycles used by an application. If the method specified is not available it will be disabled in the dialog. Measuring CPU cycles, if available, usually gives the most accurate results. (See the description of `⎕PROFILE` for more details on the different measures.)

Once you click OK on this dialog, profiling is enabled. You then run your APL code (a function, which in turn will typically call many other functions, and can ask for input from the user as part of its operation). As soon as the function finally completes or is interrupted, and APL returns to desk calculator mode, profiling is automatically disabled and the results window will open.

In this example (based on the `HELPOBJECTS` workspace supplied with APLX in library 10), we have executed a function called `RUN` which has created various graphics objects in a rotating pattern, and displayed them. On completion, the following window opens:



Name	Text	Count	Self Only		Self+Children		Average Self	Maximum	Minimum
SHAPE.Rotate[6]	R←B+.×MAT	13646	37.63%	7.797	37.63%	7.797	0.001	0.002	< 0.001
SHAPE.Draw[7]	poly←pattern+((1...	2297	10.94%	2.267	10.94%	2.267	0.001	0.005	< 0.001
SHAPE.Draw[13]	poly←pattern+((1...	2297	10.29%	2.132	10.29%	2.132	0.001	0.005	< 0.001
STAR.GetPolygon[14]	R←⌊R	1261	6.93%	1.436	6.93%	1.436	0.001	0.003	< 0.001
SHAPE.Update[8]	speed←speed×(1-2...	2297	5.56%	1.152	5.56%	1.152	0.001	0.002	< 0.001
SHAPE.Draw[6]	win.Draw('brush'...	2297	3.14%	0.651	3.14%	0.651	< 0.001	0.001	< 0.001
SHAPE.Draw[9]	win.Draw 'Poly' ...	2297	3.06%	0.635	3.06%	0.635	< 0.001	0.001	< 0.001
SHAPE.Draw[15]	win.Draw 'poly' ...	2297	2.80%	0.581	2.80%	0.581	< 0.001	0.001	< 0.001
SHAPE.Draw[12]	win.Draw('brush'...	2297	2.59%	0.536	2.59%	0.536	< 0.001	0.001	< 0.001
SHAPE.Rotate[5]	MAT←2 2⍴(2○A),(1...	13646	2.38%	0.494	2.38%	0.494	< 0.001	< 0.001	< 0.001
STAR.GetPolygon[11]	R←R⍪p	10088	2.10%	0.435	2.10%	0.435	< 0.001	< 0.001	< 0.001
STAR.GetPolygon[12]	p←ang Rotate p	10088	1.79%	0.371	39.34%	8.152	< 0.001	< 0.001	< 0.001
STAR.GetPolygon[9]	p←p⍪(⌊ang÷2)Rota...	1261	1.18%	0.245	3.57%	0.740	< 0.001	< 0.001	< 0.001
SHAPE.Rotate[4]	A←A×(○2)÷360 ⍝ C...	13646	1.17%	0.243	1.17%	0.243	< 0.001	< 0.001	< 0.001
RUN[33]	S.radius←1+50∣...	500	1.09%	0.225	1.09%	0.225	< 0.001	0.001	< 0.001
SHAPE.Update[3]	pos←pos+speed	2297	0.71%	0.147	0.71%	0.147	< 0.001	< 0.001	< 0.001
SHAPE.Draw[8]	poly←(⍴poly)⍴epo...	2297	0.69%	0.143	0.69%	0.143	< 0.001	< 0.001	< 0.001
SHAPE.Draw[14]	poly←(⍴poly)⍴epo...	2297	0.68%	0.141	0.68%	0.141	< 0.001	< 0.001	< 0.001
RUN[29]	S.Draw	500	0.64%	0.132	91.22%	18.900	< 0.001	< 0.001	< 0.001
SHAPE.Rotate[1]	⍝	13646	0.34%	0.070	0.34%	0.070	< 0.001	< 0.001	< 0.001
SHAPE.Rotate[3]	⍝	13646	0.32%	0.066	0.32%	0.066	< 0.001	< 0.001	< 0.001
SHAPE.Rotate[2]	⍝ Rotate points ...	13646	0.31%	0.065	0.31%	0.065	< 0.001	< 0.001	< 0.001

Date: 13/05/2009 12:31:13

Times shown are in billions of CPU cycles used by the APLX application Total time taken = 20.718 billion cycles

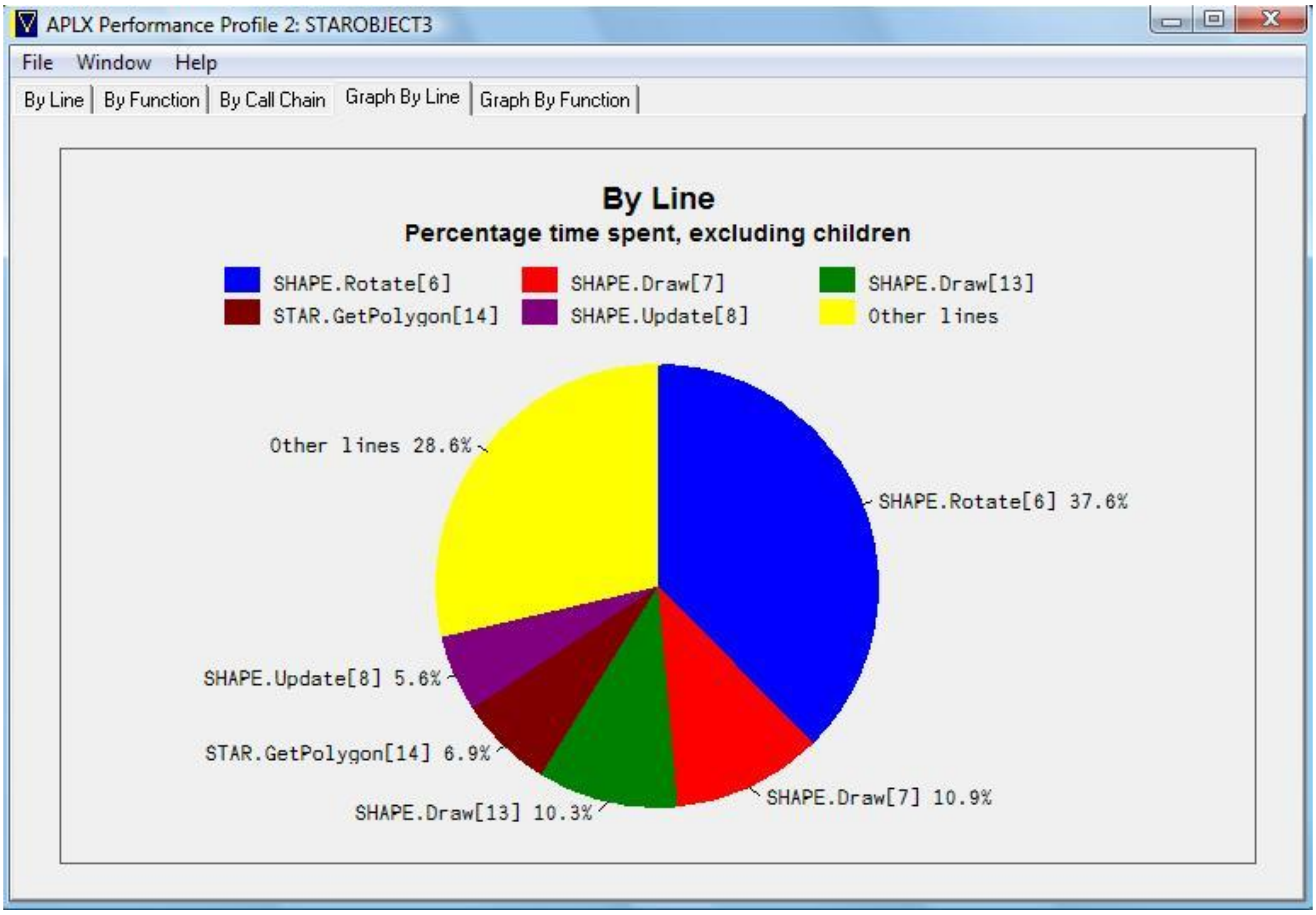
As you can see, the results are displayed on five tabs. The first tab ("By Line") tells you which individual function lines have taken the most CPU (or elapsed) time, as shown above. They are initially sorted with the one which has taken the most time first, but you can change the sort criteria by clicking on the header of a column.

CPU usage by line

In our example, the function line which has been most CPU-intensive is line 6 of `SHAPE.Rotate`, i.e. the `Rotate` method of the `SHAPE` class. This is highlighted in red in the above picture. As a convenience, the corresponding line of code is shown in the second column; you will see that it is an inner product `B+.×MAT` (highlighted in green). This line has been called 13,646 times, and accounts for 37.63% of the total execution time, or in absolute terms 7.797 billion CPU cycles (highlighted in blue in the picture). The next two most CPU-intensive lines were lines 7 and 13 of the `SHAPE.Draw` method, which took a further 10.94% and 10.29% of the total execution time respectively. In other words, nearly 60% of the total execution time was spent in just three lines of the application. Clearly, therefore, if you were wanting to optimise the code, you would see if you can reduce the number of times these critical lines are executed, or write them in a more efficient way.

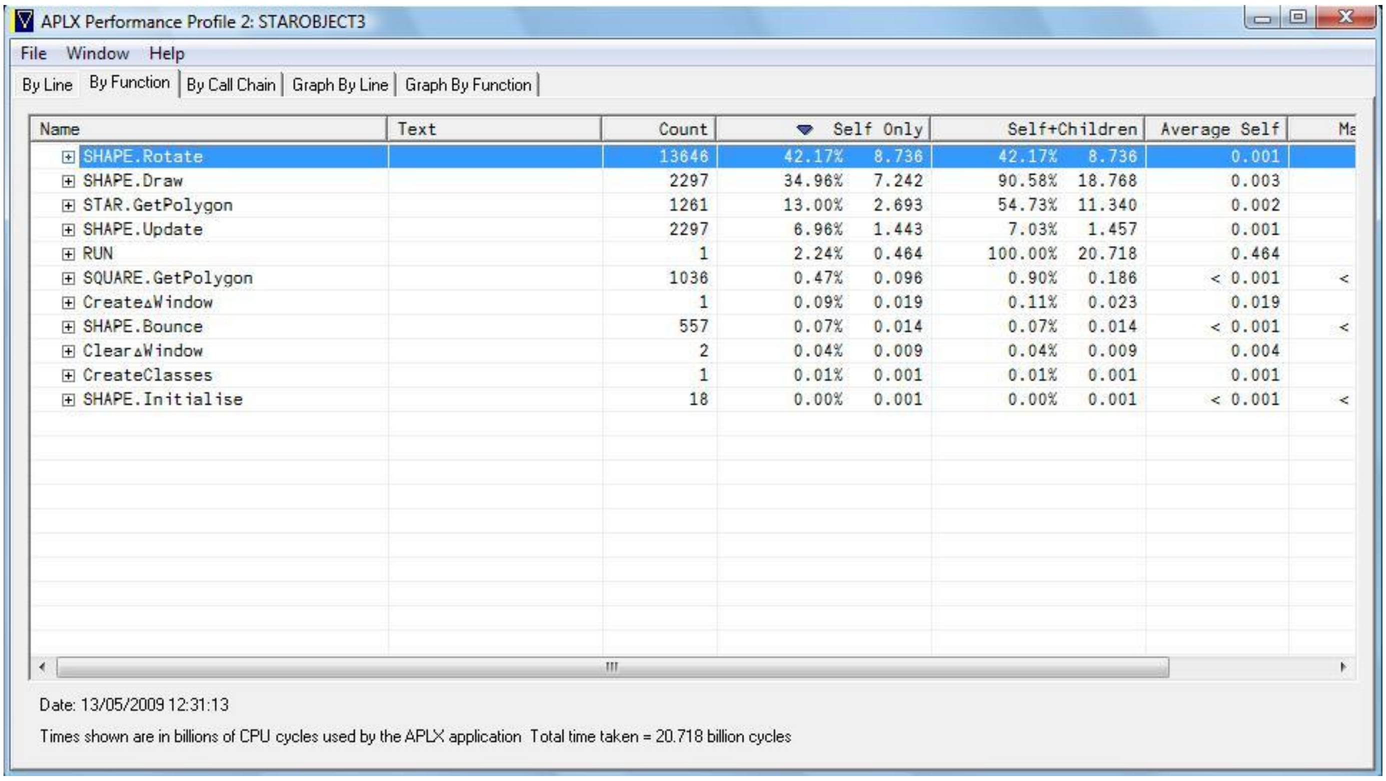
The critical information is usually in the column shown as 'Self Only'. This relates to the CPU usage for the line itself, excluding any functions called by the line. The next column ('Self+Children') displays the time taken both in the line itself and in any functions called. In our example, line 12 of `STAR.GetPolygon` has taken 39.34% of the total time, if you include the functions called by it, but only 1.79% was spent in `STAR.GetPolygon[12]` itself (highlighted in brown). In fact, most of that time was actually spent in the inner product highlighted at the top of the display.

By selecting the 'Graph By Line' tab, you can get an immediate graphical representation of which lines took the most CPU time:



CPU usage by function

As well as looking at individual lines, you can also get an overview of which functions (and methods) took the most time, by selecting the second tab of the results window ('By Function'):


APLX Performance Profile 2: STAROBJECT3

File Window Help

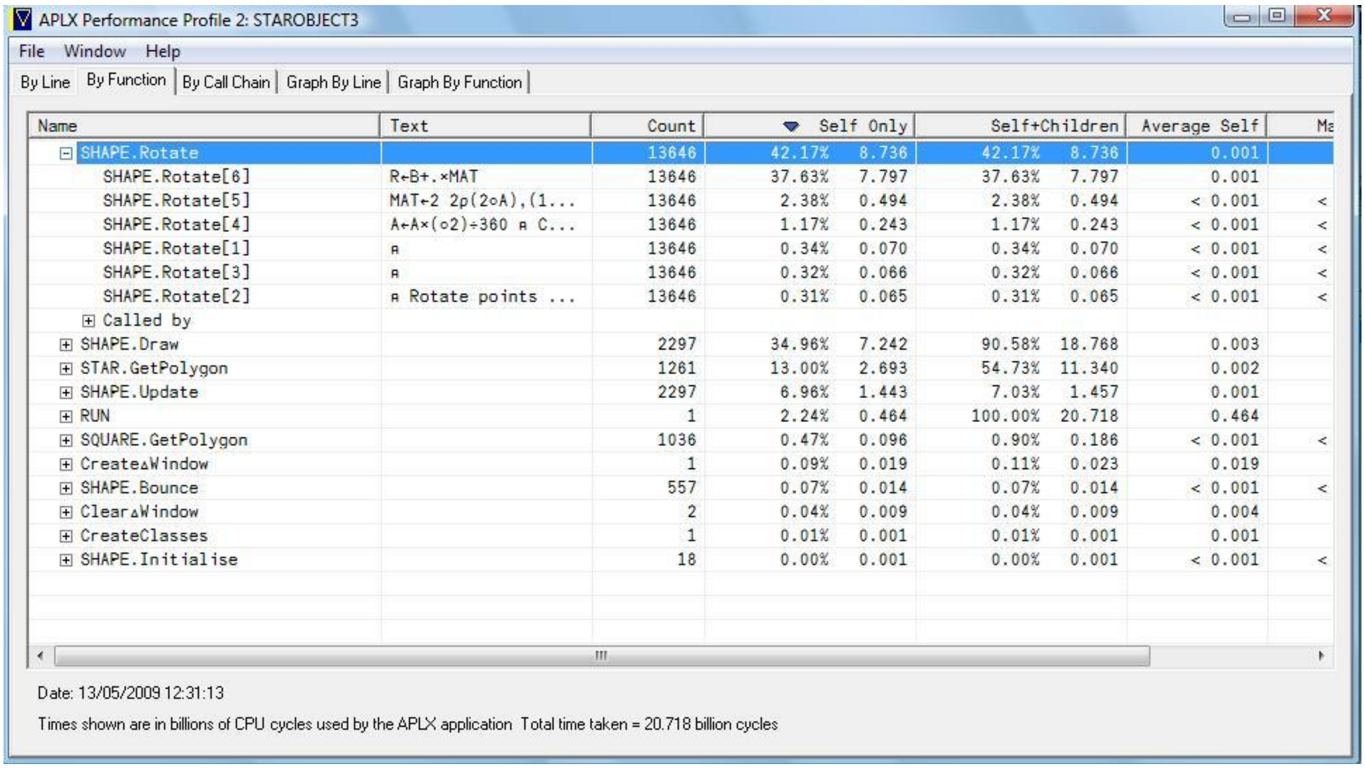
By Line | By Function | By Call Chain | Graph By Line | Graph By Function

Name	Text	Count	Self Only		Self+Children		Average Self	Ma
⊞ SHAPE.Rotate		13646	42.17%	8.736	42.17%	8.736	0.001	
⊞ SHAPE.Draw		2297	34.96%	7.242	90.58%	18.768	0.003	
⊞ STAR.GetPolygon		1261	13.00%	2.693	54.73%	11.340	0.002	
⊞ SHAPE.Update		2297	6.96%	1.443	7.03%	1.457	0.001	
⊞ RUN		1	2.24%	0.464	100.00%	20.718	0.464	
⊞ SQUARE.GetPolygon		1036	0.47%	0.096	0.90%	0.186	< 0.001	<
⊞ Create∆Window		1	0.09%	0.019	0.11%	0.023	0.019	
⊞ SHAPE.Bounce		557	0.07%	0.014	0.07%	0.014	< 0.001	<
⊞ Clear∆Window		2	0.04%	0.009	0.04%	0.009	0.004	
⊞ CreateClasses		1	0.01%	0.001	0.01%	0.001	0.001	
⊞ SHAPE.Initialise		18	0.00%	0.001	0.00%	0.001	< 0.001	<

Date: 13/05/2009 12:31:13

Times shown are in billions of CPU cycles used by the APLX application Total time taken = 20.718 billion cycles

From this display you can 'drill down' within a given function, to find out where the time is spent in that function, by clicking on the twist-down by the function name:


APLX Performance Profile 2: STAROBJECT3

File Window Help

By Line | By Function | By Call Chain | Graph By Line | Graph By Function

Name	Text	Count	Self Only		Self+Children		Average Self	Ma
⊟ SHAPE.Rotate		13646	42.17%	8.736	42.17%	8.736	0.001	
SHAPE.Rotate[6]	R←B+.×MAT	13646	37.63%	7.797	37.63%	7.797	0.001	
SHAPE.Rotate[5]	MAT←2 2⍴(2○A),(1...	13646	2.38%	0.494	2.38%	0.494	< 0.001	<
SHAPE.Rotate[4]	A←A×(○2)÷360 ⍝ C...	13646	1.17%	0.243	1.17%	0.243	< 0.001	<
SHAPE.Rotate[1]	⍝	13646	0.34%	0.070	0.34%	0.070	< 0.001	<
SHAPE.Rotate[3]	⍝	13646	0.32%	0.066	0.32%	0.066	< 0.001	<
SHAPE.Rotate[2]	⍝ Rotate points ...	13646	0.31%	0.065	0.31%	0.065	< 0.001	<
⊞ Called by								
⊞ SHAPE.Draw		2297	34.96%	7.242	90.58%	18.768	0.003	
⊞ STAR.GetPolygon		1261	13.00%	2.693	54.73%	11.340	0.002	
⊞ SHAPE.Update		2297	6.96%	1.443	7.03%	1.457	0.001	
⊞ RUN		1	2.24%	0.464	100.00%	20.718	0.464	
⊞ SQUARE.GetPolygon		1036	0.47%	0.096	0.90%	0.186	< 0.001	<
⊞ Create∆Window		1	0.09%	0.019	0.11%	0.023	0.019	
⊞ SHAPE.Bounce		557	0.07%	0.014	0.07%	0.014	< 0.001	<
⊞ Clear∆Window		2	0.04%	0.009	0.04%	0.009	0.004	
⊞ CreateClasses		1	0.01%	0.001	0.01%	0.001	0.001	
⊞ SHAPE.Initialise		18	0.00%	0.001	0.00%	0.001	< 0.001	<

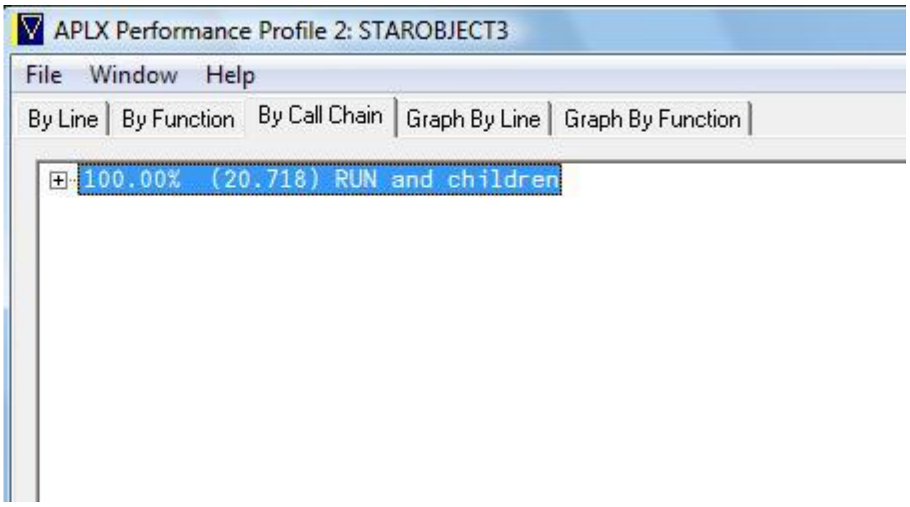
Date: 13/05/2009 12:31:13

Times shown are in billions of CPU cycles used by the APLX application Total time taken = 20.718 billion cycles

The 'Graph By Function' tab shows the overview by function as a pie chart.

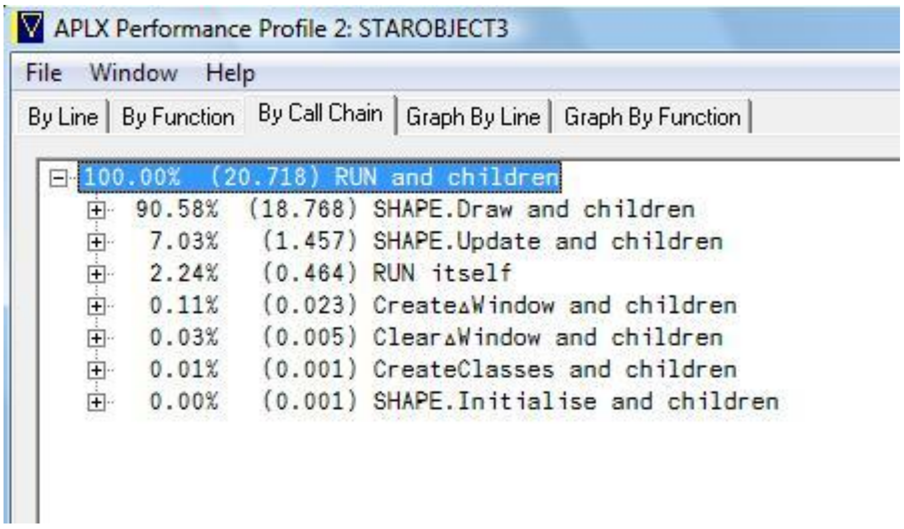
CPU usage by Call Chain

The third way of looking at the data is by *call chain*. This is best shown by our example:



```
100.00%  (20.718) RUN and children
```

This shows that, unsurprisingly, 100% of the time was spent in our top-level function `RUN` and in all the functions called by it. We can now use the twist-downs to drill down into the the hierarchies of calls to see how that time was divided up:



```
100.00%  (20.718) RUN and children
    90.58%  (18.768) SHAPE.Draw and children
     7.03%   (1.457) SHAPE.Update and children
     2.24%   (0.464) RUN itself
     0.11%   (0.023) Create∆Window and children
     0.03%   (0.005) Clear∆Window and children
     0.01%   (0.001) CreateClasses and children
     0.00%   (0.001) SHAPE.Initialise and children
```

Saving the profiling data as a web page

As well as looking at the data in the APLX window, you can select 'Save As Web Page' from the File menu. This saves a complete report (either as a summary of the most important items only, or of the whole application), as a web page which you can load in any standard browser.

Appendix: APLX Character Set and Unicode Mapping

APLX Character set

⎕AV characters and mapping to Unicode

The Unicode values used when translating APLX characters to Unicode follow the standard described by Adrian Smith in Vector 19.3 (January 2003). The full set of characters (not all of which may be displayable on a given platform), and the values used when mapping from APLX characters to Unicode, are as follows:

(Non-printing control characters are not shown).

Char.	⎕AV position (hex)	Unicode value (hex)	Description
⍐	07	2350	File hold
	08	0008	Backspace
⍗	09	2357	File drop
	0A	000a	Line feed
⍇	0B	2347	File read
⍈	0C	2348	File write
	0D	000d	Carriage-return, Newline
⍱	0E	2371	Nor
⍲	0F	2372	Nand
⍒	10	2352	Grade down
⍋	11	234b	Grade up
⌽	12	233d	Reverse
⍉	13	2349	Transpose
⊖	14	2296	Circle bar
⍟	15	235f	Log
⌶	16	2336	I-Beam
⍫	17	236b	Del-Tilde
⍎	18	234e	Execute
⍕	19	2355	Format
⍀	1A	2340	Slope-Bar
⌿	1B	233f	Slash-Bar
⍝	1C	235d	Lamp
⍞	1D	235e	Quote-Quad
!	1E	0021	Exclamation
⌹	1F	2339	Domino
	20	0020	Space
¨	21	00a8	Diaeresis, Each
)	22	0029	Right parenthesis
<	23	003c	Less than
≤	24	2264	Not greater than
=	25	003d	Equal
>	26	003e	Greater
]	27	005d	Right bracket
∨	28	2228	Or
^	29	005e	And
≠	2A	2260	Not equal
÷	2B	00f7	Divide
,	2C	002c	Comma
+	2D	002b	Plus
.	2E	002e	Period

/	2F	002f	Slash
0	30	0030	0
1	31	0031	1
2	32	0032	3
3	33	0033	3
4	34	0034	4
5	35	0035	5
6	36	0036	6
7	37	0037	7
8	38	0038	8
9	39	0039	9
(	3A	0028	Left parenthesis
[	3B	005b	Left bracket
;	3C	003b	Semi-colon
×	3D	00d7	Multiply
:	3E	003a	Colon
\	3F	005c	Slope
¯	40	00af	High minus
⍺	41	237a	Alpha
⊥	42	22a5	Decode
∩	43	2229	Cap
⌊	44	230a	Floor
∊	45	220a	Epsilon
_	46	005f	Underbar
∇	47	2207	Del
{	48	007b	Left brace
⍳	49	2373	Iota
∘	4A	2218	Jot
'	4B	0027	Single quote
⎕	4C	2395	Quad
|	4D	007c	Stile, Remainder
⊤	4E	22a4	Encode
○	4F	25cb	Circle
*	50	002a	Star
?	51	003f	Query
⍴	52	2374	Rho
⌈	53	2308	Ceiling
~	54	007e	Not
↓	55	2193	Drop
∪	56	222a	Cup
⍵	57	2375	Omega
⊃	58	2283	Right shoe
↑	59	2191	Take
⊂	5A	2282	Left shoe
←	5B	2190	Left arrrow
⊢	5C	22a2	Right tack
→	5D	2192	Right arrow
≥	5E	2265	Not less than
-	5F	002d	Minus
⋄	60	22c4	Diamond
A	61	0041	A
B	62	0042	B
C	63	0043	C
D	64	0044	D
E	65	0045	E
F	66	0046	F
G	67	0047	G
H	68	0048	H
I	69	0049	I
J	6A	004A	J
K	6B	004B	K
L	6C	004C	L

M	6D	004D	M
N	6E	004E	N
O	6F	004F	O
P	70	0050	P
Q	71	0051	Q
R	72	0052	R
S	73	0053	S
T	74	0054	T
U	75	0055	U
V	76	0056	V
W	77	0057	W
X	78	0058	X
Y	79	0059	Y
Z	7A	005A	Z
∆	7B	2206	Delta
⊣	7C	22a3	Left tack
⍪	7D	236a	Comma bar
$	7E	0024	Dollar
}	7F	007d	Right brace
┌	90	250c	Line draw top left
┐	91	2510	Line draw top right
└	92	2514	Line draw bottom left
┘	93	2518	Line draw bottom right
─	94	2500	Line draw horizontal
│	95	2502	Line draw vertical
┼	96	253c	Line draw cross
├	97	251c	Line draw join right
┤	98	2524	Line draw join left
┴	99	2534	Line draw join up
┬	9A	252c	Line draw join down
	9B	001B	Escape
	9C	001C	[Spare]
Í	9D	00cD	I acute
	9E	001E	[Spare]
	9F	001F	[Spare]
"	A0	0022	Double-quote
#	A1	0023	Hash
%	A2	0025	Percent
&	A3	0026	Ampersand
@	A4	0040	At
£	A5	00A3	Pound
`	A6	0060	Backquote
≡	A7	2261	Match
≢	A8	2262	Not match
⍷	A9	2377	Epsilon underbar
⍸	AA	2378	Iota underbar
⌻	AB	233b	Quad-Jot
⍂	AC	2342	Quote-Slope
⍤	AD	2364	Jot diaeresis
⍥	AE	2365	Circle diaeresis
⌷	AF	2337	Squad
Ä	B0	00c4	A diaeresis
Å	B1	00c5	A ring
Ç	B2	00c7	C cedilla
É	B3	00c9	E acute
Ñ	B4	00d1	N tilde
Ö	B5	00d6	O diaeresis
Ü	B6	00dc	U diaeresis
á	B7	00e1	a acute
à	B8	00e0	a grave
â	B9	00e2	a circumflex

ä	BA	00e4	a diaeresis
ã	BB	00e3	a tilde
å	BC	00e5	a ring
ç	BD	00e7	c cedilla
é	BE	00e9	e acute
è	BF	00e8	e grave
ê	C0	00ea	e circumflex
ë	C1	00eb	e diaeresis
í	C2	00ed	i acute
ì	C3	00ec	i grave
î	C4	00ee	i circumflex
ï	C5	00ef	i diaeresis
ñ	C6	00f1	n tilde
ó	C7	00f3	o acute
ò	C8	00f2	o grave
ô	C9	00f4	o circumflex
ö	CA	00f6	o diaeresis
õ	CB	00f5	o tilde
ú	CC	00fa	u acute
ù	CD	00f9	u grave
û	CE	00fb	u circumflex
ü	CF	00fc	u diaeresis
À	D0	00c0	A grave
Ã	D1	00c3	A tilde
Õ	D2	00d5	O tilde
Œ	D3	0152	OE
œ	D4	0153	oe
Æ	D5	00c6	AE
æ	D6	00e6	ae
⍬	D7	236c	Zilde
Ø	D8	00d8	O /
ø	D9	00f8	o /
¿	DA	00bf	Inverted ?
¡	DB	00a1	Inverted !
β	DC	00df	Beta
ÿ	DD	00ff	y diaeresis
	DE	0000	[Spare]
	DF	0000	[Spare]
	E0	0000	[Spare]
a	E1	0061	a
b	E2	0062	b
c	E3	0063	c
d	E4	0064	d
e	E5	0065	e
f	E6	0066	f
g	E7	0067	g
h	E8	0068	h
i	E9	0069	i
j	EA	006A	j
k	EB	006B	k
l	EC	006C	l
m	ED	006D	m
n	EE	006E	n
o	EF	006F	o
p	F0	0070	p
q	F1	0071	q
r	F2	0072	r
s	F3	0073	s
t	F4	0074	t
u	F5	0075	u
v	F6	0076	v

w	F7	0077	w
x	F8	0078	x
y	F9	0079	y
z	FA	007A	z
⍙	FB	2359	Delta underbar
È	FC	00c8	E grave
€	FD	20ac	Euro
	FE	0000	[Spare]
	FF	007f	Rubout

Alternative Unicode mappings

There may be several different Unicode characters which look similar to, or provide a similar function to, an APLX character. In order to maximise the probability of being able to represent the text when converting from Unicode to internal representation, APLX accepts as input a number of alternative Unicode values for certain characters. These are as follows:

Unicode character (hex)	Unicode meaning	Mapped to
00a6	Broken vertical bar	| Remainder
2223	'Divides' in 'Mathematical operators'	| Remainder
2227	Logical And in 'Mathematical operators'	∧ And
223c	Tilde in 'Mathematical operators'	∼ Not
22c6	Star in 'Mathematical operators'	⋆ Star
2013	En dash	− Minus
2212	Minus in 'Mathematical operators'	− Minus
2044	'Fraction slash'	/ Slash
2215	'Division slash'	/ Slash
25AF	Square in 'Geometric shapes'	⎕ Quad
25ca	Diamond in 'Geometric shapes'	⋄ Diamond
203e	Overline	¯ High minus
2018	Left quote	' Single quote
2019	Right quote	' Single quote
201c	Left double quote	" Double quote
201d	Right double quote	" Double quote
03b1	Greek alpha	⍺ Alpha
03b2	Greek beta	β Beta-S
03b5	Greek epsilon	∊ Epsilon
03b9	Greek iota	⍳ Iota
03c1	Greek rho	⍴ Rho
03c9	Greek omega	⍵ Omega
2028	Line-separator character	Carriage-return
2029	Paragraph-separator character	Carriage-return