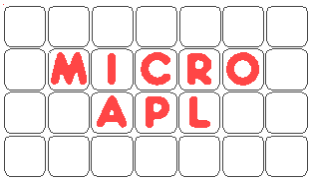
New Features in APLX Version 5



Copyright © 2009 MicroAPL Ltd. All rights reserved worldwide.

APLX, APL.68000 and MicroAPL are trademarks of MicroAPL Ltd. All other trademarks acknowledged.

APLX is a proprietary product of MicroAPL Ltd, and its use is subject to the license agreement in force. Unauthorized copying or use of APLX is illegal.

MicroAPL Ltd makes no warranties in respect of the suitability of APLX for any particular purpose, and accepts no liability for any loss arising out of the use of APLX or arising from the information contained in this manual.

MicroAPL welcomes your comments and suggestions.
Please visit our website: http://www.microapl.co.uk/apl

APLX Version 5.0.2 Upgrade notes: August 2009

Contents

1. Performance Profiling

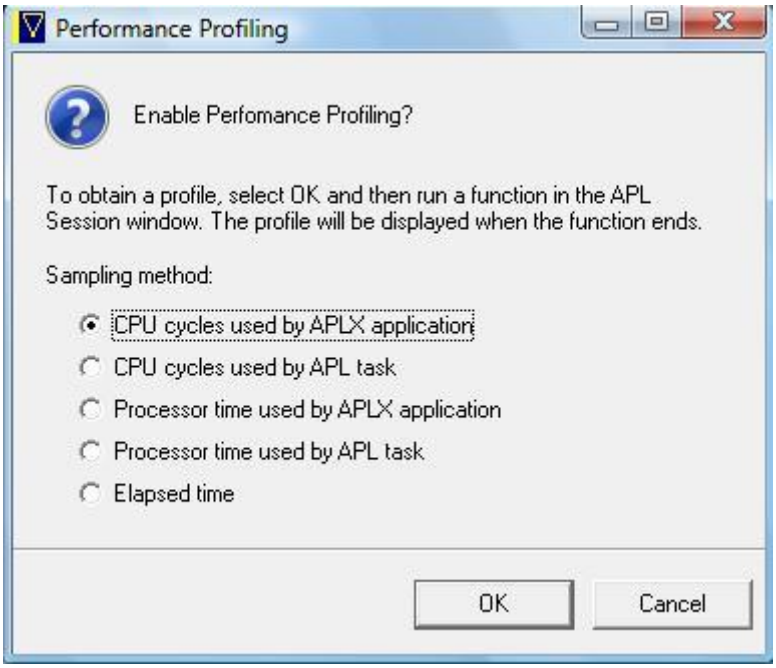
Overview

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:

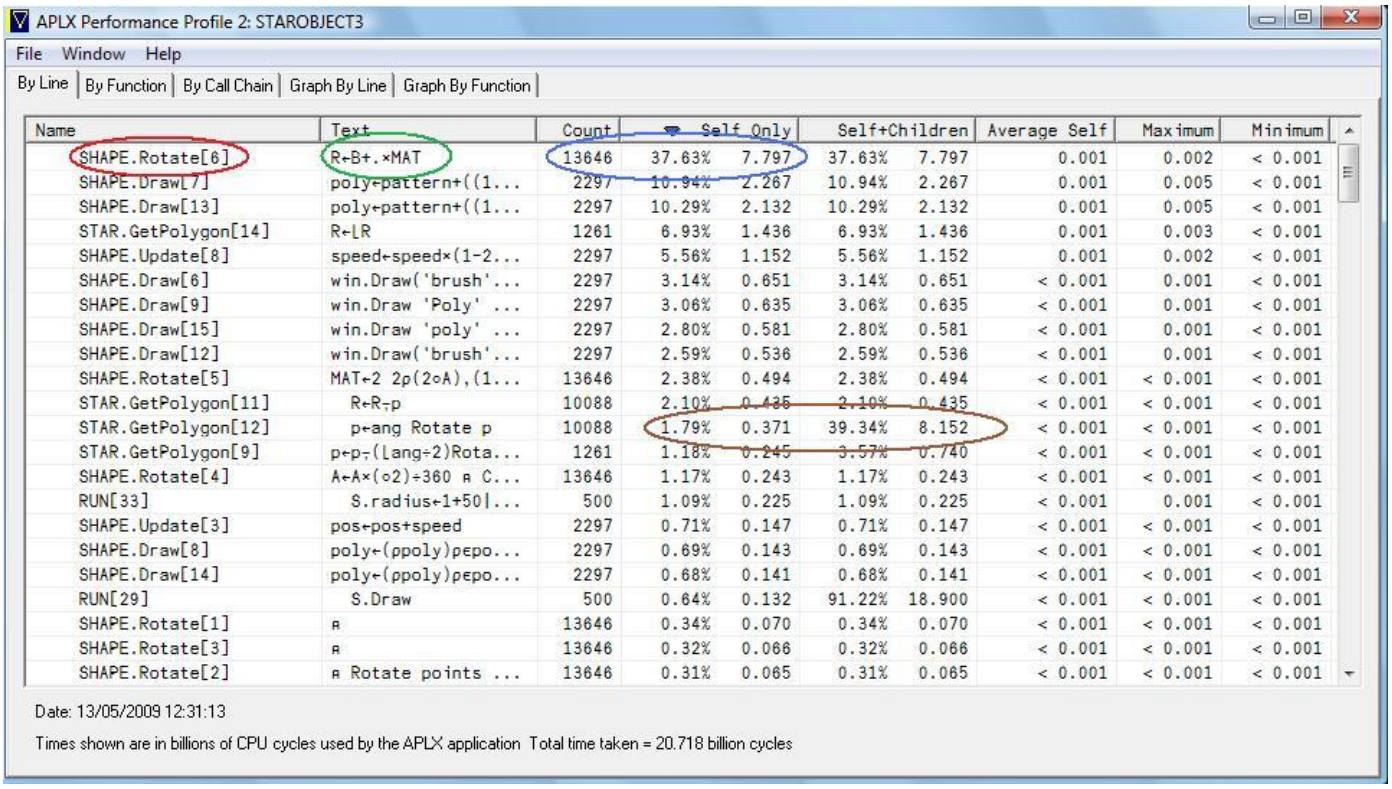


Depending on which platform you are using, one or more of the timing methods may not be available. For example, earlier versions of Windows cannot measure the number of CPU cycles used by an application. If the method specified is not available it will be disabled in the dialog. Measuring CPU cycles, 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:



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.

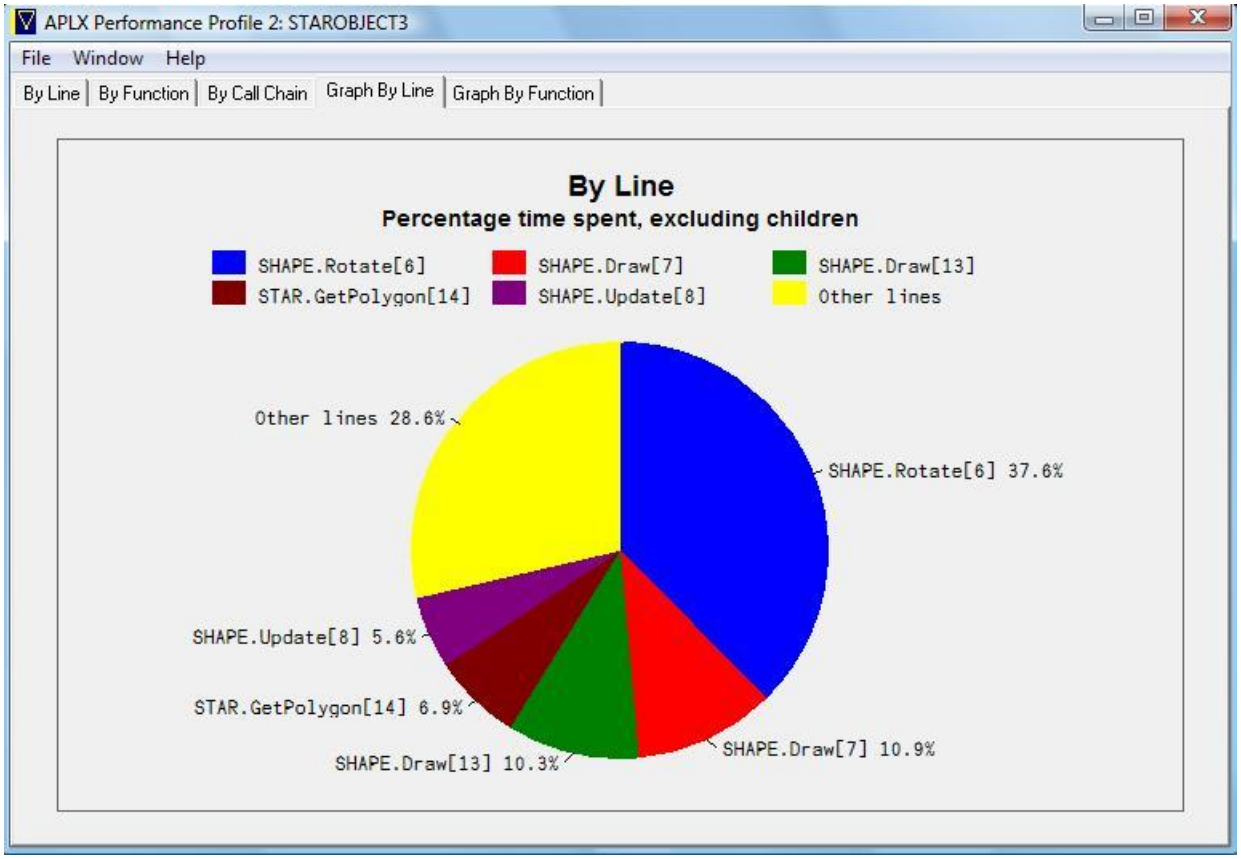
CPU usage by line

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

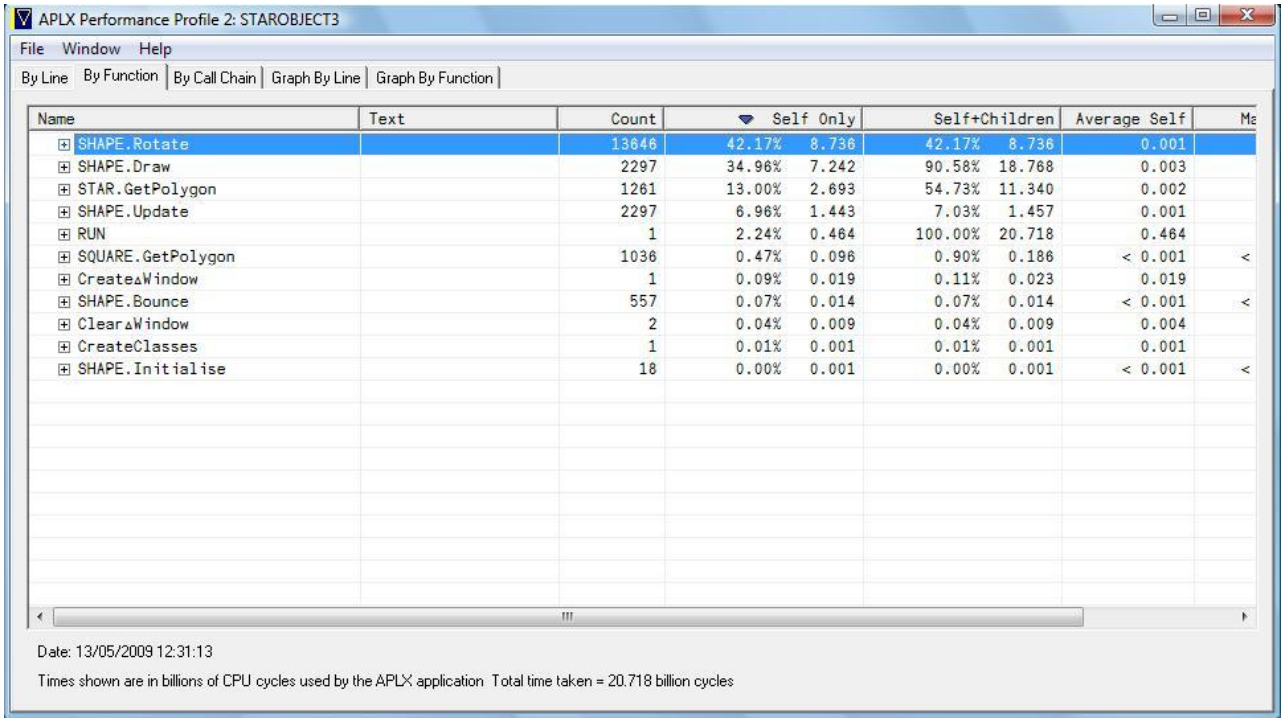
('Self+Children') displays the time taken both in the line itself and in any functions called. In our example, line 12 of `STAR.GetPolygon` has taken 39.34% of the total time, if you include the functions called by it, but only 1.79% was spent in `STAR.GetPolygon[12]` itself (highlighted in brown). In 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:

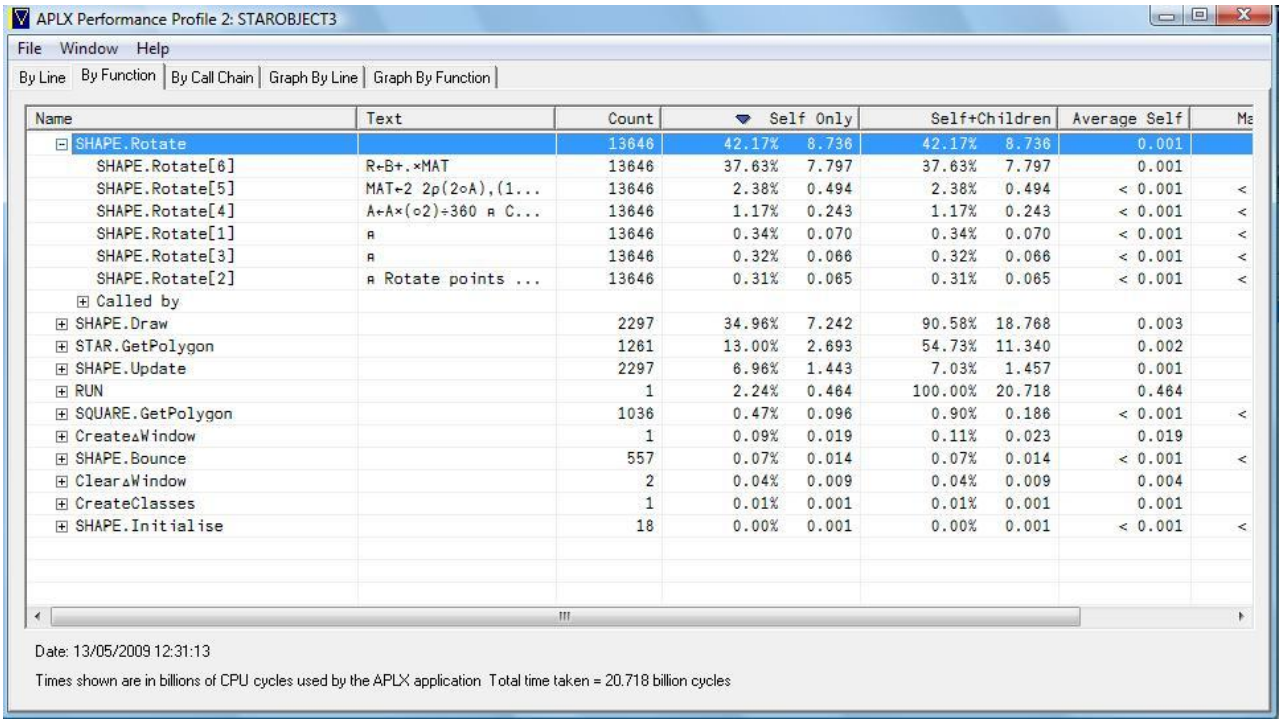


CPU usage by function

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 'By Function' tab of the results window:



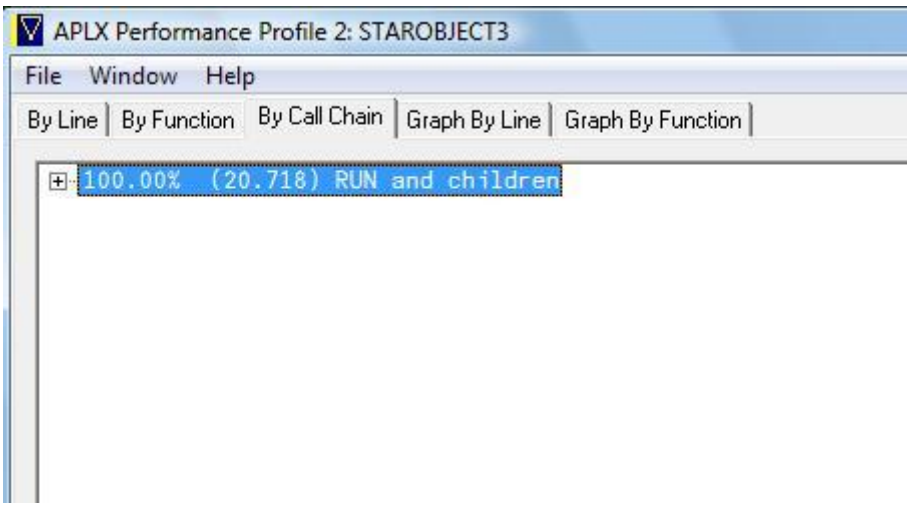
From this display you can 'drill down' within a given function, to find out where the time is spent in that function, by clicking on the twist-down by the function name:



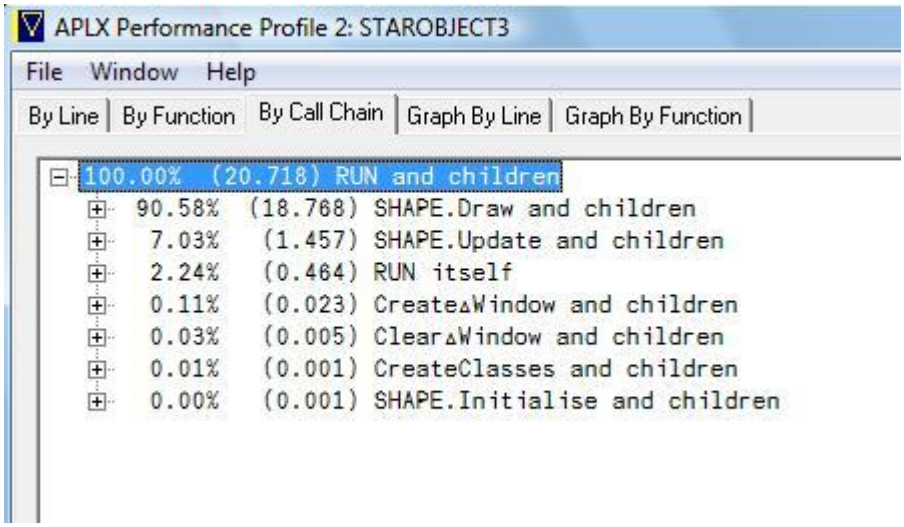
The 'Graph By Function' tab shows the overview by function as a pie chart.

CPU usage by Call Chain

The third way of looking at the data is by *call chain*. This is best shown by our example:



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:



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.

Using ⎕PROFILE for more detailed control of profiling

For more control over the profiling process you can use the ⎕PROFILE system function as follows:

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'`

2. Function, Workspace and Text Comparison

Overview

APLX Version 5 includes a powerful new facility for comparing and/or merging:

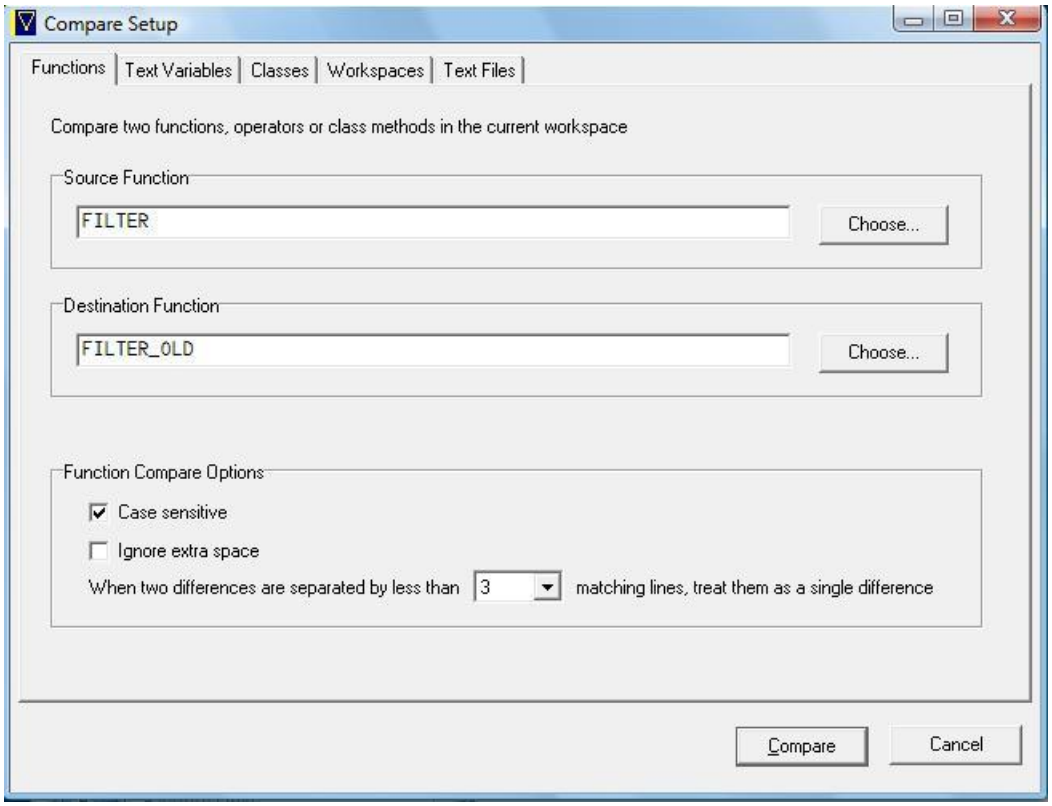
- Two functions (or class methods) in the current workspace
- Two text variables in the current workspace
- Two class definitions in the current workspace
- A workspace on disk with the current workspace
- Two text files

In each case, you see all the differences in a two-pane display, and optionally copy across changes from one of the versions (the *source*) to the other (the *destination*).

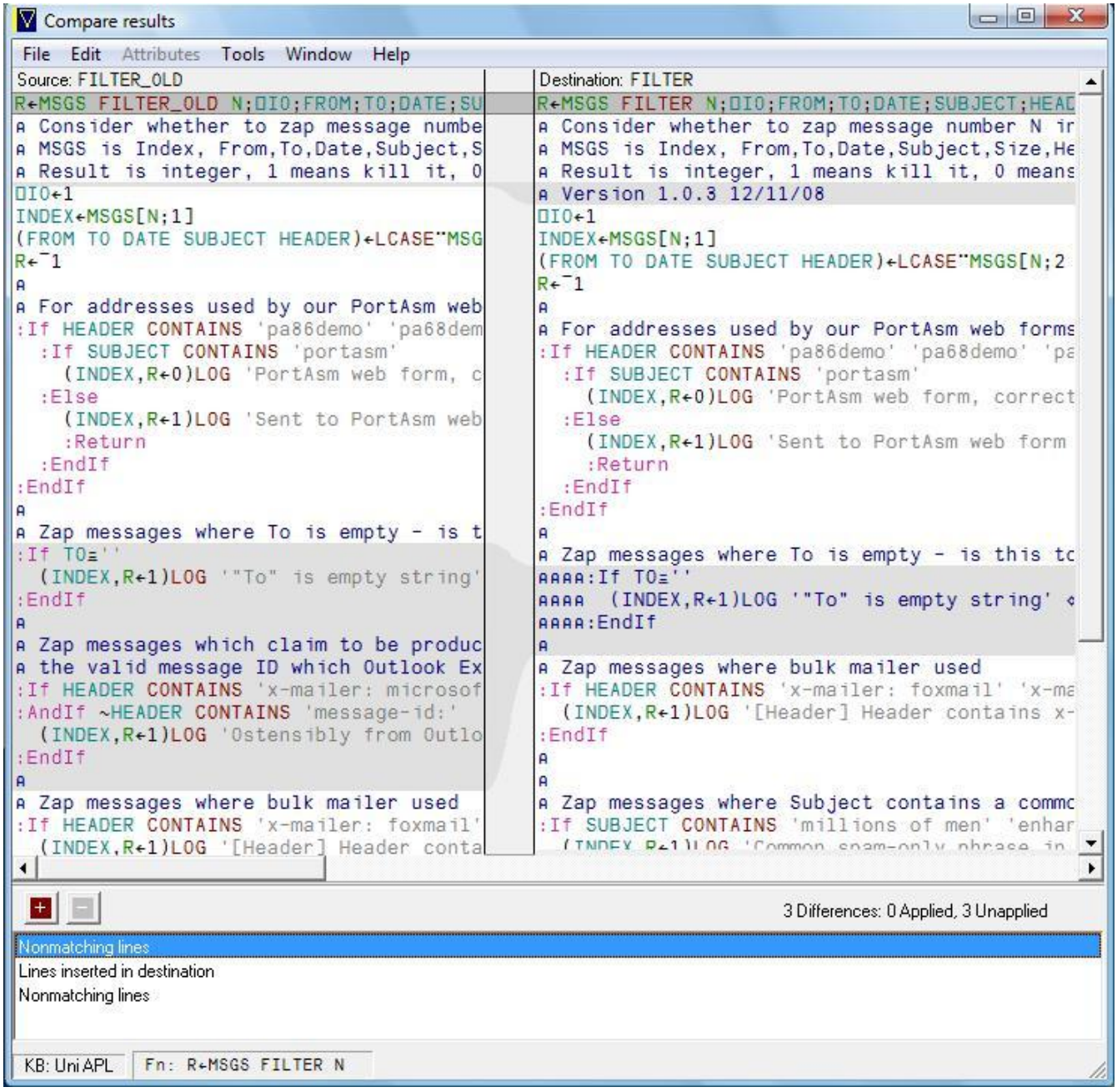
You access the Comparison feature by selecting Compare.. in the Tools menu. A dialog then appears which allows you to specify which items you want to compare.

Comparing two functions or text variables

In this use of the Compare feature, you choose two functions/class methods (or text variables) in the current workspace. Any changes you make will affect the Destination only:



When you click OK, APLX brings up the result of the comparison:



There are three types of difference:

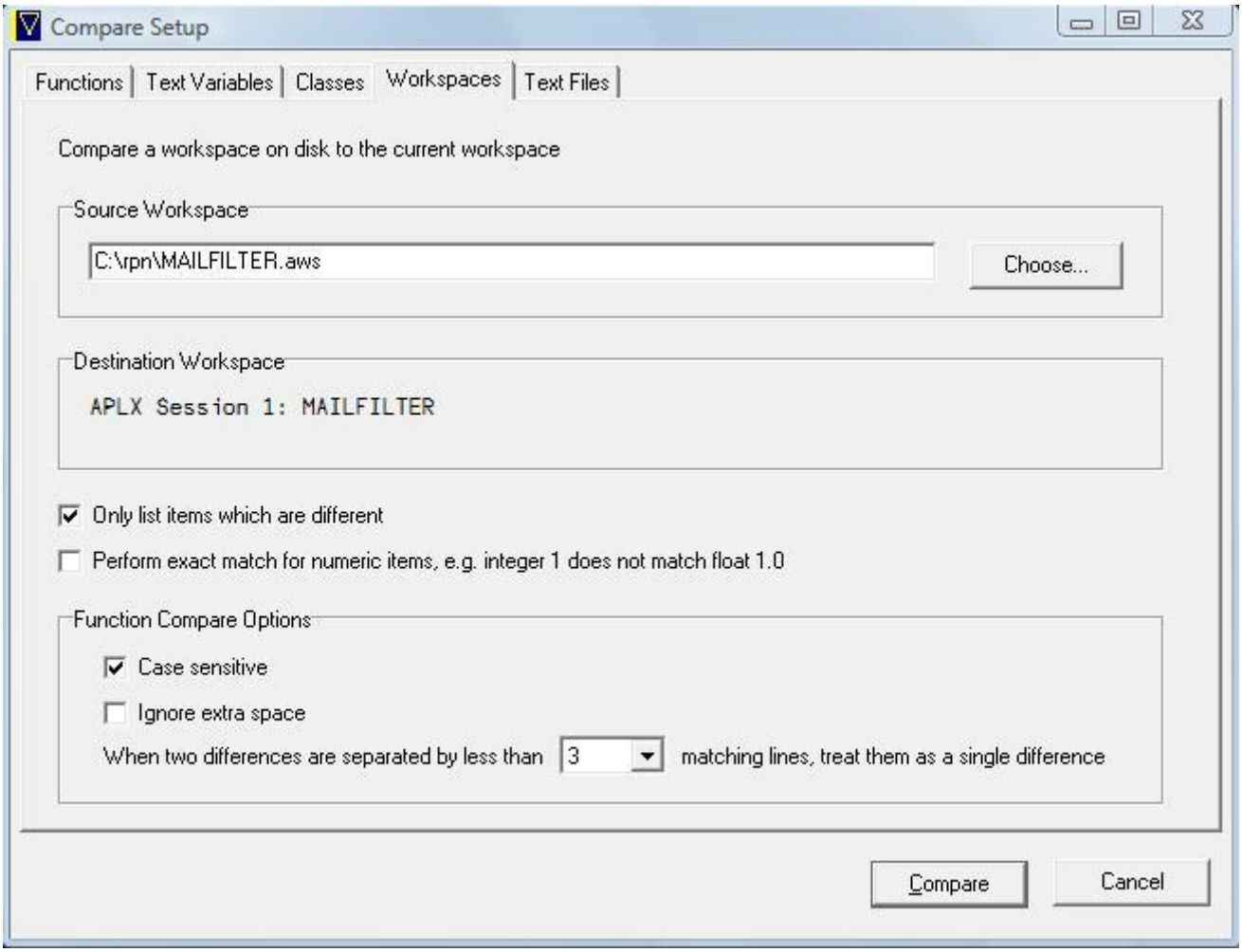
- Lines which appear in both versions, but are different in content. These are identified as 'Nonmatching lines' in the lower pane of the window. In the above example, the first line is different because the source has the function name `FILTER_OLD` and the destination `FILTER`.
- Lines which appear in the destination, but not in the source. These are marked as 'Lines inserted in destination'. An example is the second difference, where the right-hand pane has an extra line with the version number comment.
- Lines which appear in the source, but not in the destination, marked as 'Lines deleted from destination'.

In each case, you can click on the appropriate difference in the lower pane (or use the Up and Down arrow keys) to select the difference, and then press the red + button to copy the difference over to the destination. (You can undo this by clicking on the red - button).

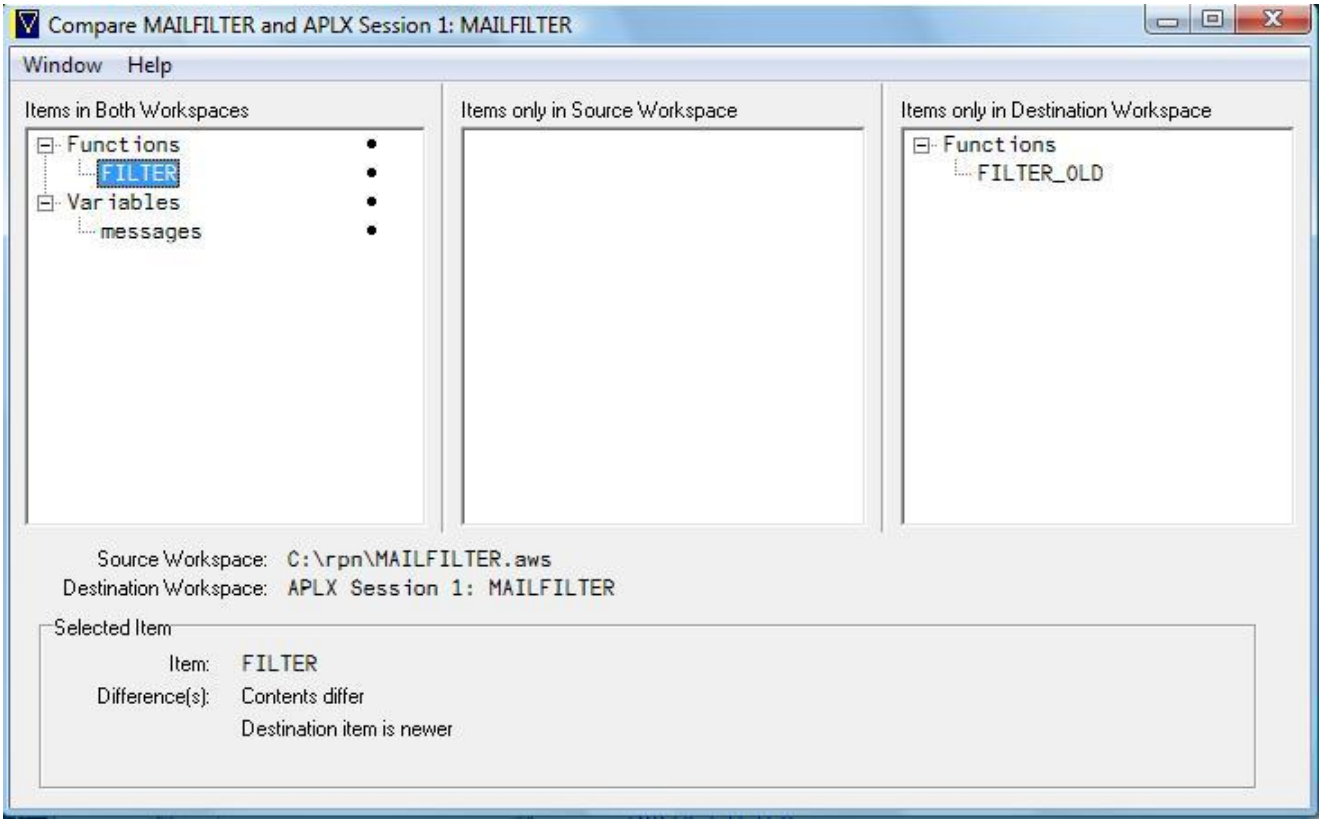
You can also make changes by hand in the destination pane.

Finally, when you close the window, you will be prompted to confirm that you want to save the differences back in the workspace.

Comparing two workspaces



In this case, the comparison is always done between a workspace on disk (the source), and the currently active workspace (the destination). A summary of the results appears first, showing which items appear in both workspaces but have been found to be different (left hand pane), which items appear only in the source (middle pane), and which items appear only in the destination:



For functions, methods and operators, and for text variables, you can double-click on the name of the item (for example `FILTER` or `messages` in the above picture) to bring up the same comparison/edit window which we have described above. This means that you can easily merge differences from the saved workspace (on disk) into the currently-active workspace

Comparing two classes in the current workspace

This is similar to comparing two functions, except that the individual members (methods and properties) are identified separately in the results window.

Comparing two text files

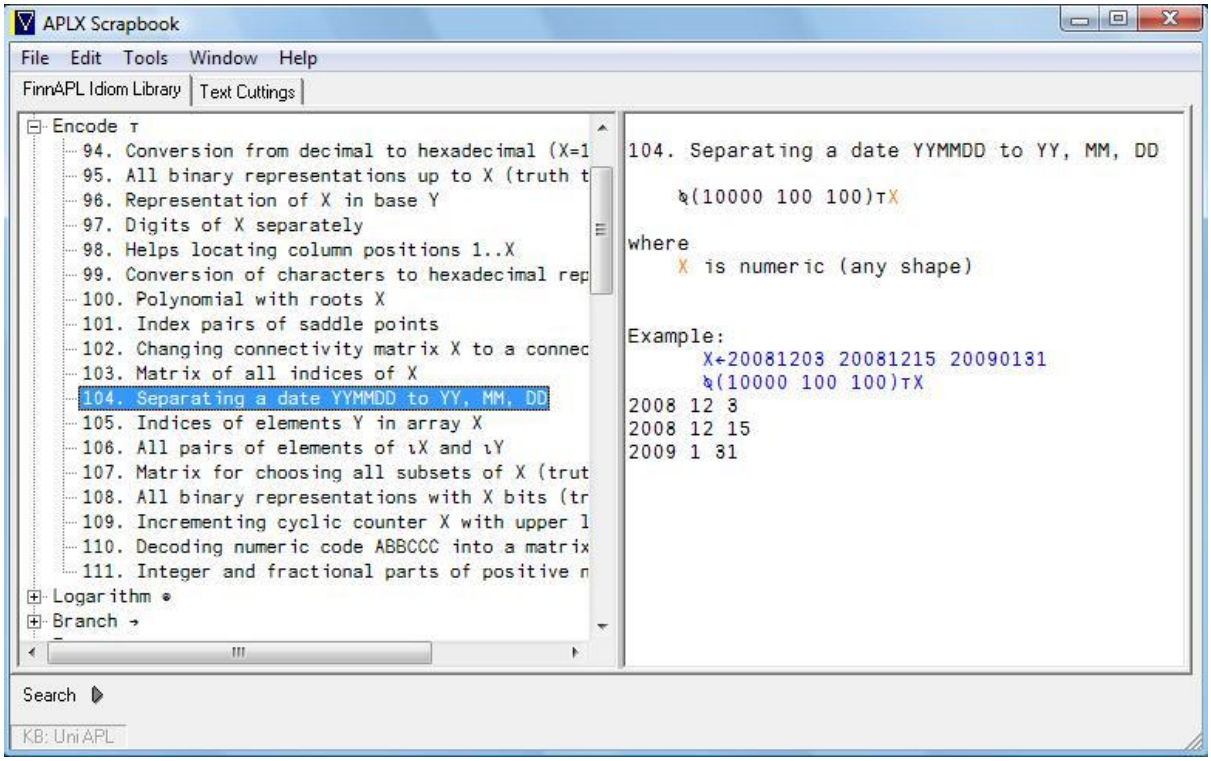
This is similar to comparing two text variables, except the source and destination are text files on disk.

3. Scrapbook: Idioms and Cuttings

The new APLX Scrapbook facility is designed to make it easy for you to re-use code snippets. You access it via the Tools menu.

FinnAPL Idioms

The Finnish APL Association FinnAPL have put together an extensive catalogue of small APL idioms, and have kindly given MicroAPL their permission to include it in APLX distributions. We have updated it and added some extra examples and explanation, and it can be accessed and searched using the APLX Scrapbook:



Note that the FinnAPL idiom library was written in a time before the APL2 extensions to the APL language existed. In some cases there is now a simpler way of achieving the same result.

Text Cuttings

The ‘Text Cuttings’ pane can be used to save APL expressions or code samples which you use frequently, or any other text which you might need again. For example, if you use a standard function documentation template, you could store it here as a cutting. Cuttings are saved automatically when you exit APLX.

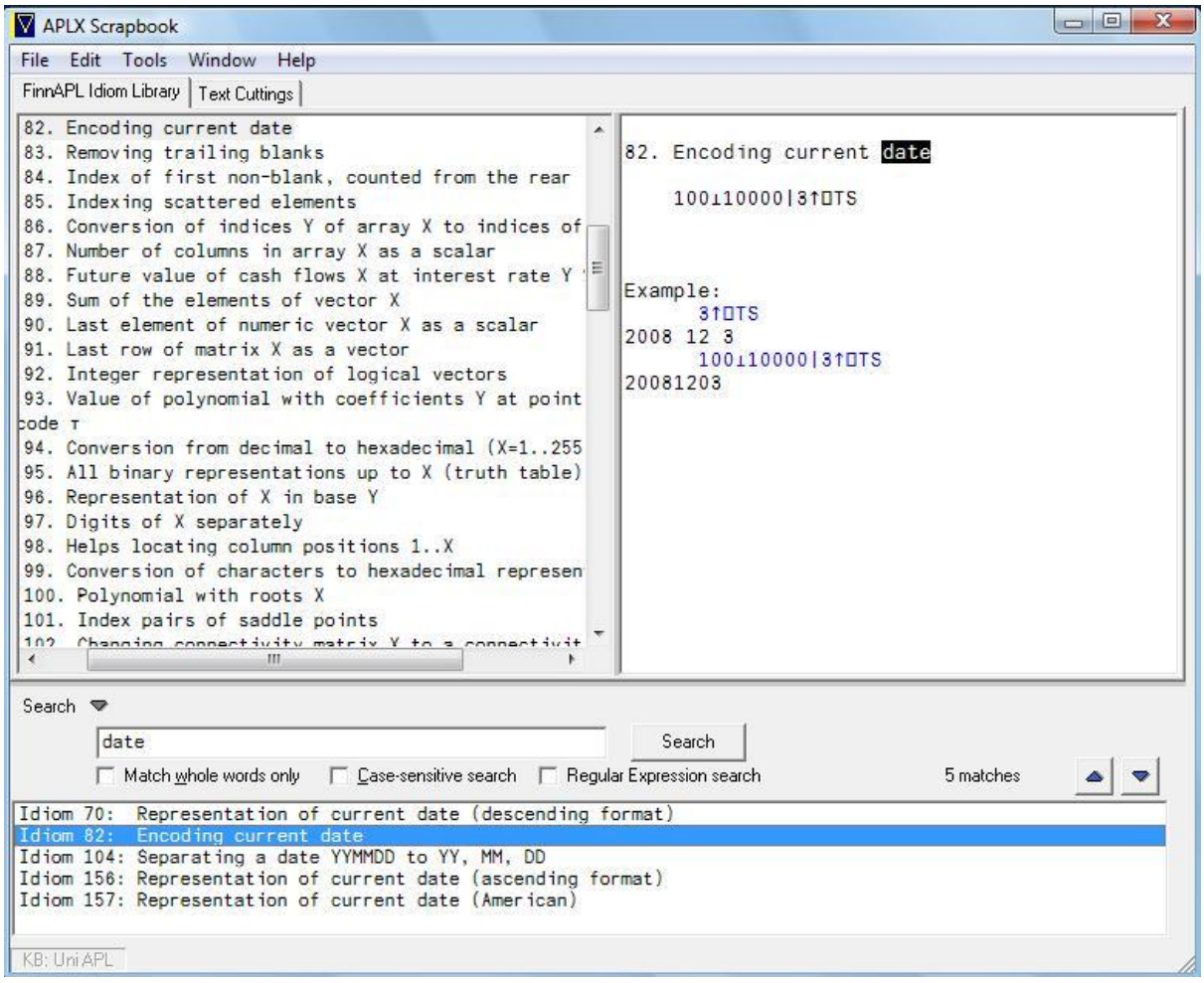
Use the New button to create a cutting.

You can change the name of a cutting at any time by clicking twice on the cutting name (not double-clicking) and then editing the name.

You can copy text from the Idioms or Cuttings pane to the clipboard (using the Edit menu or Ctl-C).

Searching Idioms and Cuttings

In both cases you can search the collection of idioms or cuttings by using the 'Search' twistdown at the bottom of the window:



In this example, we have searched the FinnAPL idiom list for the word 'date'. Five matches have been found. You can navigate through the matches by using the arrow buttons at the bottom right, or by clicking on one of the lines in the results panel at the bottom.

4. White-Space Retention

In versions of APLX prior to version 5, the interpreter automatically strips out unnecessary blank space within lines of code in functions, operators, and methods (other than before comments). It also strips out empty lines.

Version 5 includes a new option (enabled by default) which causes the interpreter to keep a copy of the exact layout seen in the editor window when you saved the function. This enables you to lay out the code exactly as you wish.

The retention of blank space works both for functions edited using the editor window, and for functions fixed using `⎕FX`. It does not work for functions edited using the line-oriented `∇` (del) editor.

The new facility is enabled by default. If you want to disable it, and go back to the old behaviour, you can do so using the 'APL' tab of the Preference dialog.

Note that if you load an APLX Version 5 workspace into a previous version of APLX and edit a function, the extra white space will be stripped out.

5. Object-Oriented programming: Mixins

What are Mixins?

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.

6. New external class interface: R

APLX Version 4 introduced object-oriented APL programming using classes and objects, and also implemented a unique external class interface which allows the APL programmer to make use of classes written in other object-oriented environments, in particular .Net, Java and Ruby.

APLX Version 5 adds a fourth external interface, to the R statistical language and set of statistical packages. Although R is not a full object-oriented language, the same object-based interface makes it very easy to use from APLX.

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. To find out what they should be set do, take a look at the shell script called 'R'. As a minimum, you need to ensure the `R_HOME` environment variable is set to the installation directory where R is installed (typically `/usr/local/lib/R/` for Linux, or `/Library/Frameworks/R.framework/Resources/` for MacOS. You can set up the environment in one of three ways:

- Define `R_HOME` (and any other environment variables required) in your user profile, so that they are always defined;
- *(Linux only)* Define the environment variables in your `startaplx` script, so they are defined before APLX is invoked;
- Use `⎕SETUP` to set the environment from within APL, before you open the interface to R, as in these examples:

Linux:
```
'r' ⎕SETUP 'R_HOME' '/usr/local/lib/R/'
```
MacOS:
```
'r' ⎕SETUP 'R_HOME' '/Library/Frameworks/R.framework/Resources/'
```

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
```

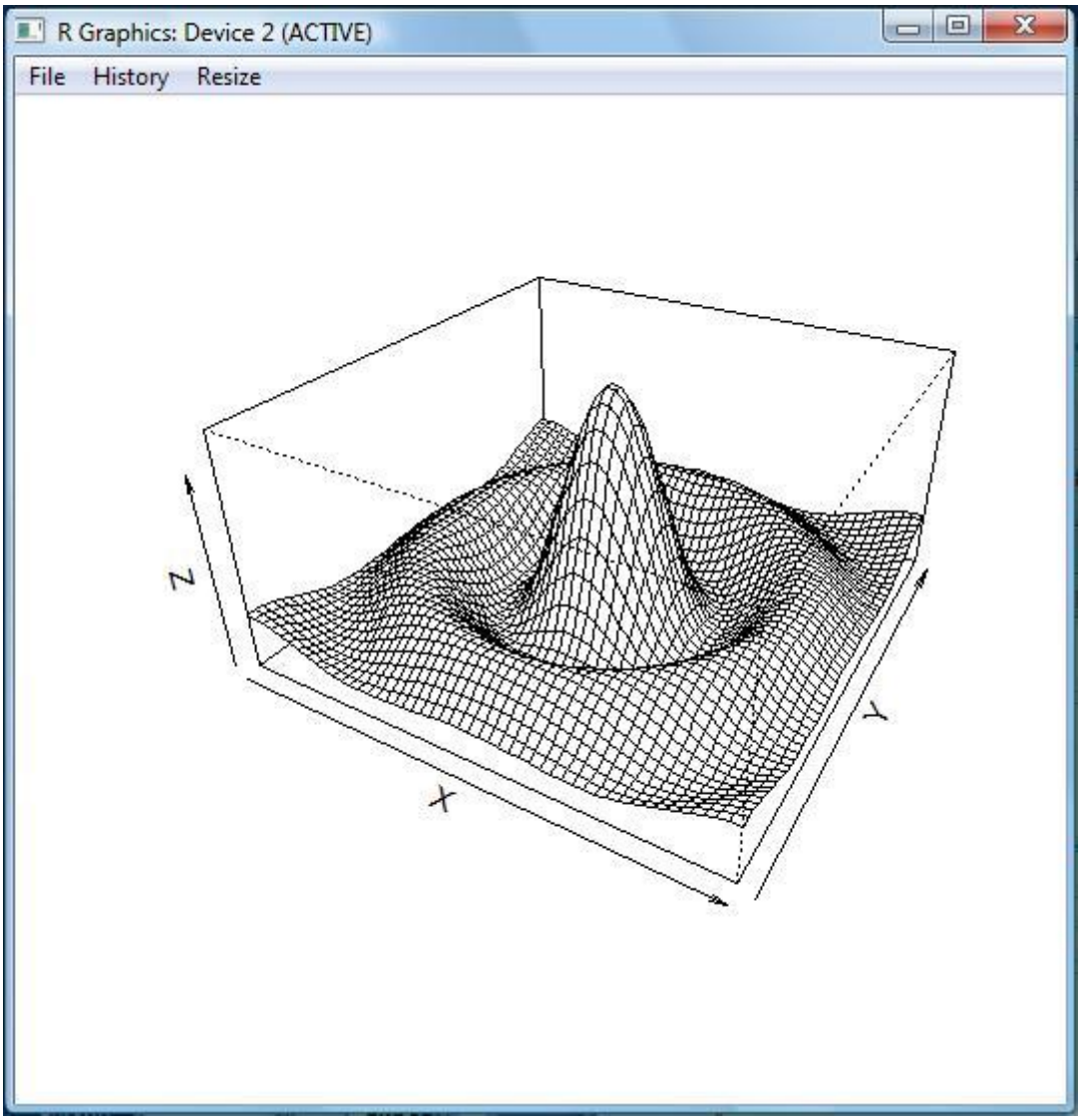
Example: 3-D plot

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
┌→──────────┐
│ ┌→──┐ ┌→┐ │
│ │Fred│ │99│ │
│ └───┘ └~┘ │
└∈──────────┘
```

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 is 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 convenient 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

     frame.plot
```

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`.

7. New Primitive Functions

APLX Version 5 introduces the following new primitive functions:

Primitive fn	***Description***
∪	Unique
∪	Union
∩	Intersection
⊣	Stop
⊣	Left
⊢	Pass
⊢	Right
≠	Not Match

∪ Unique

Unique is a monadic function 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
```

∪ Union

Union is a dyadic function which 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│ └─────┘     │
│                           └──┘             │
└∈──────────────────────────────────────────┘
```

∩ Intersection

Intersection is a dyadic function which 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
```

⊣ Stop

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

The dyadic 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

The monadic function ⊢ (pass) simply passes its argument through unchanged. The argument can be of any type, rank and shape; the result is identical.

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

The dyadic 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
```

≢ 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
1                             enclosed 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.

8. New System Functions & Variables

APLX Version 5 introduces the following new system functions and variables:

System Fn/Var	***Description***
⎕LE	Last Exception
⎕MC	Missing Character
⎕PROFILE	Performance profiling
⎕XML	Convert to/from XML

⎕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.

⎕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.

In this example, the Unicode value 937 (hex 03A9, representing the Greek capital omega character) is translated to the default 'missing character' value (question mark) because it has no equivalent in the APLX character set:

```
      ⎕UCS 937 8364 223
?€β
```

A different 'missing character' can be set using ⎕MC

```
      ⎕MC←'$'
      ⎕UCS 937 8364 223
$€β
```

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
- extracted from XML data using ⎕XML

⎕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 cases, you don't need to use ⎕PROFILE: 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, the ⎕PROFILE system function can be used.

See the separate section on Performance Profiling for details.

⎕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.

Note: The ⎕IMPORT and ⎕EXPORT functions have been enhanced in APLX version 5 to allow data to be transferred to/from XML files in a single step. The format and functionality is the same that of ⎕XML, except that the XML prologue is handled automatically (see below).

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.

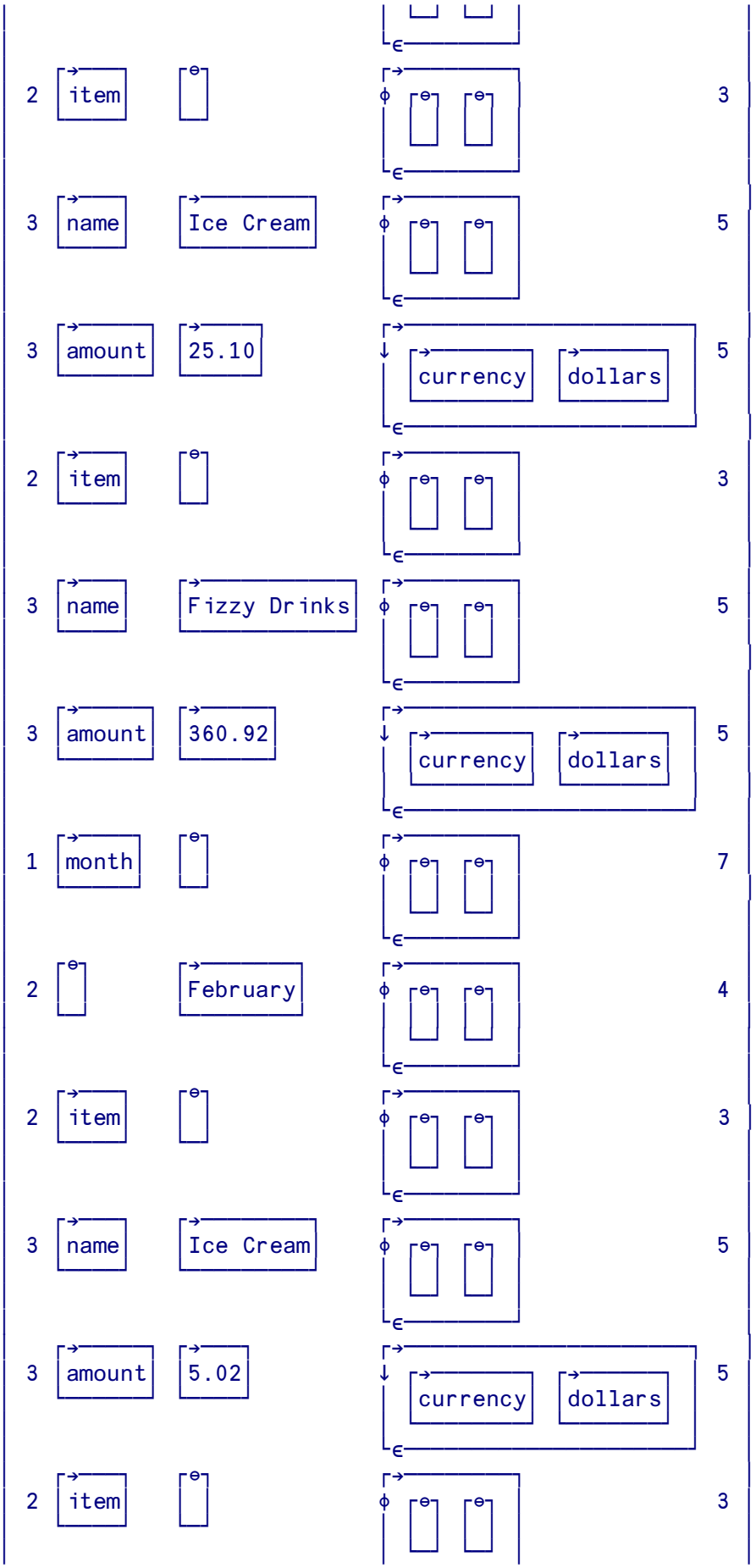
The result is an N-row, 5-column matrix containing a flattened representation of the XML data. Each element in the XML data will produce one row in the result. The columns are as follows:

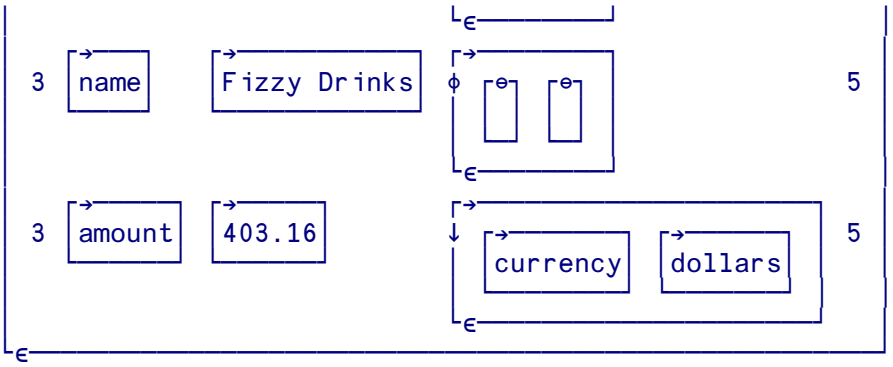
Column 1:	An integer indicating the depth of nesting of the element. A value of 0 is used for the outer-most nesting level, with deeper nesting being indicated by higher numbers.
Column 2:	The element name as specified in the start tag.
Column 3:	The element data as a character vector
Column 4:	An M-row, 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 ┌→─────┐ ┌⊖┐          ┌→──────────┐                      3 │
     │sales │ │ │          ⍬ ┌⊖┐ ┌⊖┐ │                        │
     └──────┘ └─┘          │ │ │ │ │ │                        │
                           └∈──────────┘                        │
     ┌→─────┐ ┌⊖┐          ┌→──────────┐                      7 │
  1  │month │ │ │          ⍬ ┌⊖┐ ┌⊖┐ │                        │
     └──────┘ └─┘          │ │ │ │ │ │                        │
                           └∈──────────┘                        │
     ┌⊖┐      ┌→───────┐   ┌→──────────┐                      4 │
  2  │ │      │January │   ⍬ ┌⊖┐ ┌⊖┐ │                        │
     └─┘      └────────┘   │ │ │ │ │ │                        │
```


2 item
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



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 **'strip-enclosing'**

  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 stripped.

  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 "&" 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 as unmodified text, e.g. "&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.

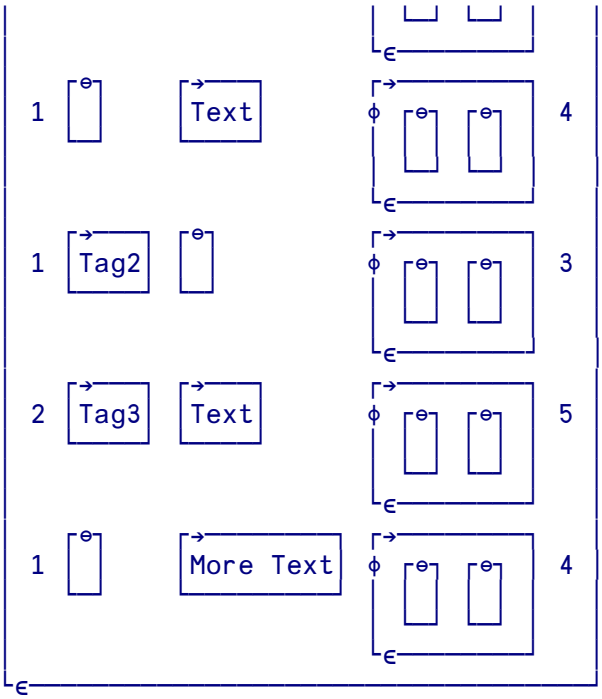
32 Element is XML Processing Instruction, e.g. *<?xml version="1.0" encoding="utf-8"?>*.
The processing instruction is returned in column 2, and column 3 is blank.

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│
│   └─────┘ └─┘       │ │ │ │ │ │  │
└──────────────────────┴─┴─┴─┴─┴─┴──┘
```



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). For 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 **'strip-enclosing'**

  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 stripped. 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

⎕XML is based on the original design concepts and implementation by Mark E. Johns, and has been designed in cooperation with Dyalog Ltd

9. New 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. The following new system methods have been added in APLX Version 5:

System Method	***Description***	***Applies to internal classes?***	***Applies to external classes?***
⎕EVAL	Evaluate expression	No	Yes *(R only)*
⎕MIXIN	Mix another class into object	Yes	No
⎕MIXINS	Return list of mixins	Yes	No
⎕UNMIX	Remove mixins from object	Yes	No

⎕EVAL Evaluate external expression

Implemented (as a system method) for the R external class only.

Syntax:

```
result ← objref.⎕EVAL string
```

The monadic ⎕EVAL system method allows an arbitrary expression to be evaluated in the external environment (currently R only). It is provided as a more convenient form of the system function of the same name. The object reference must be an R session object.

The right argument is a text vector containing the expression. 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:

```
      'r' ⎕eval 'mean(x[2,])'
5
```

⎕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.. (Within user-defined method, same as ⎕THIS.⎕MIXIN)
    mixin_ref ← {arch} ⎕MIXIN Class Arg1...      "
```

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). In the latter case, the system function is dyadic.

⎕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 separate 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
```

⎕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.

10. Enhanced System Functions

⎕IMPORT ⎕EXPORT

These now support import and export of XML files, using the file type/extension '`xml`', e.g.

```
array ⎕EXPORT 'filename' 'xml'
```

For `⎕EXPORT` the left argument (which gets written to file in XML format) 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"?>
```

For `⎕IMPORT` the explicit result is an APL array with the same specification as `⎕XML`. In otherwords

```
array ← ⎕IMPORT 'filename' 'xml'
```

...is equivalent to:

```
array ← ⎕XML ⎕IMPORT 'filename' 'utf8'
```

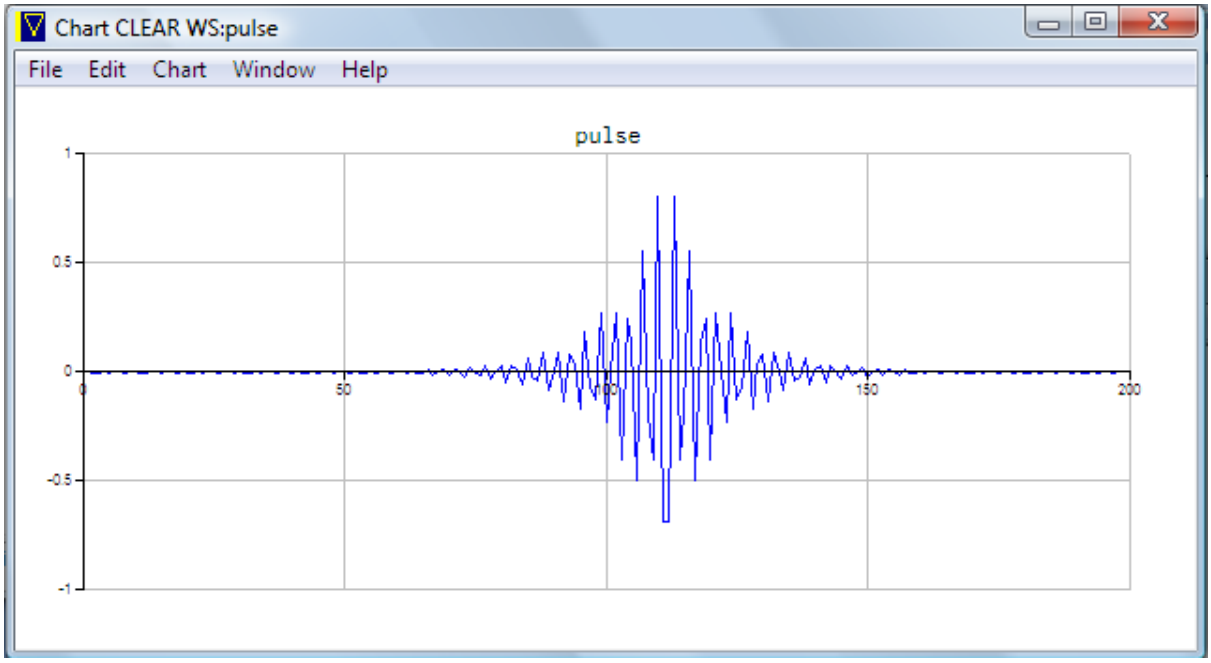
⎕CHART

`⎕CHART` has been extended to allow the a window to be re-used, for 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.

For example:

```
      ∇ 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
      ∇
```

This displays an animated window with a pulse wave moving rapidly across it.



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
```

⎕PFKEY Set up Function keys

Associating a sequence of strings with a function key

An extension to ⎕PFKEY (function-key programming) now allows you to associate a sequence of strings, rather than just a single string, 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.

11. Enhancements to System Classes

Scalable Vector Graphics (SVG) in Chart Object and Draw method

The Chart object and the Draw method now support the export of the chart or image in Scalar Vector Graphics (SVG) format.

SVG is a platform-independent high-quality graphics standard which, as its name implies, allows an image to be scaled to a different size, without losing quality. It is thus ideal for publishing your charts on a web-page or for printing in a publication.

There are two ways in which you can access the SVG representation of a Chart object:

- The new `svg` read-only property returns a text vector containing the chart's image in SVG format.

- The `Save` method can now save the image to disk in SVG format.

For images you create directly using the `Draw` method, you can access the current image produced by the drawing system (as a character vector of SVG commands) by using the `'GetSVG'` keyword. The syntax is:

```
R ← Control.Draw 'GetSVG'
```

System Object – Support for user-defined and animated cursors

The new method `Loadpointer` method can be used to define a new pointer or cursor. The pointer is given a number in the range 100 - 119 which can then be used when setting the `pointer` property of a visible control, as shown in this example:

```
⍝ Load pointer from file and give it the number 100
'⎕' ⎕wi 'Loadpointer' 100 'C:\Windows\Cursors\banana.ani'

⍝ Set 'mycontrol' to use this as its pointer
mycontrol.pointer←100
```

The pointer only needs to be loaded once and it can then be shared by multiple controls.

Loading a pointer from file

Syntax: `'⎕' ⎕WI 'Loadpointer' Number Filename`

This form allows a pointer to be loaded from a file. `Number` is an integer in the range 100 - 119, and `Filename` is a character vector

Windows: The data in the file must be in either .CUR or .ANI format.

Macintosh and Linux: Loading a cursor from file is not currently supported

Creating a pointer from data

Syntax: `'□' □WI 'Loadpointer' Number Hostspot_Y Hotspot_X Bitmap Mask`

This form allows a monochrome pointer to be specified directly. `Bitmap` is an M x N boolean matrix, where a 0 represents black and 1 represents white. `Mask` is a boolean matrix of the same shape, in which a 0 specifies that the corresponding bit in `Bitmap` is transparent. The position of the pointer hot-spot is given by the `Hotspot_Y` and `Hotspot_X` parameters, with `0 0` being the top left corner.

Example:

```
cursor←15 15⍴16↑1
cursor←cursor∨⌽cursor
'□' □WI 'Loadpointer' 100 8 8 (~cursor) cursor
```

Using a pre-loaded pointer

Syntax: `'□' □WI 'Loadpointer' Number Handle`

This form can be used if you have obtained a handle to a cursor through some other means, for example a `□NA` call to the operating system. `Handle` is an integer scalar containing the handle to use.

Image Class – Support for overlaying transparent pictures

The Image class can now be used to overlay one transparent or semi-transparent image over another. Two new methods implement this:

`Setopacity`: ImageMagick supports images which are transparent, which means that when you place one image on top of another using the `Overlay` method, the background image will still be partially visible through the foreground image. Some file formats, such as PNG, allow specific parts of the image to be transparent. The `Setopacity` method allows you to specify that the whole image is transparent. it takes a single argument, which is a number between 0.0 (completely transparent) to 1.0 (fully opaque, i.e. the background is not visible at all).

`Overlay`: This method allows you to overlay one image (the foreground) on top of another (the background). If the foreground image is transparent, the background image will partially show through it. Normally you first load the background image into the Image object, and then use the `Overlay` method to load the foreground image (from file) on top of it, at a specified position. You can also overlay an image from another Image object.

To load a foreground image from file, call the `Overlay` method with this syntax:

```
      Image.Overlay 'file_name' Top Left <Mode>
or
     'ImageName' ⎕WI 'Overlay' 'file_name' Top Left <Mode>
```

where `file_name` is the name of the file to load (one of the standard formats supported by ImageMagick), and `Top` and `Left` are the positions in the Image control where it should be loaded (these can be omitted, in which case 0 0 is assumed). If the file name is an empty vector, a dialog will be invoked to allow the user to select a file. The last parameter `Mode` is normally not needed - see below.

To load a foreground image from another Image object, call the `Overlay` method with this syntax:

```
      Image.Overlay Handle Top Left <Mode>
or
     'ImageName' ⎕WI 'Overlay' Handle Top Left <Mode>
```

where the `Handle` parameter is an integer representing the `handle` property of the source Image object. This is very useful if you want to create the foreground image dynamically, or load a non-transparent image and make it transparent (see the Setopacity method)

This function loads a background image from file, and places a foreground image (also from file) over it:

```
      ∇DEMO_Overlay;Win
[1]   '⎕' ⎕wi 'scale' 5
[2]   Win←'⎕' ⎕NEW 'form' ⋄ Win.title←'Transparency' ⋄ Win.where←1 1 420
700
[3]   Win.Pic.new 'Picture'
[4]   Win.Pic.Img.new 'Image' ⋄ Win.Pic.Img.scale←5 ⋄ Win.Pic.align←¯1
[5]   Win.Pic.Img.file←'c:\pictures\background.jpg'
[6]   Win.Pic.Img.Overlay 'c:\pictures\foreground.png' 200 120
[7]   ⍝1 ⎕WE Win
      ∇
```

This version of the function does the same, but uses a second Image object to hold the foreground image and transform it, before copying it into the background:

```
      ∇DEMO2_Overlay;Win;Img2;handle
[1]   '⎕' ⎕wi 'scale' 5
[2]   Win←'⎕' ⎕NEW 'form' ⋄ Win.title←'Transparency' ⋄ Win.where←1 1 420
700
[3]   Win.Pic.new 'Picture'
[4]   Win.Pic.Img.new 'Image' ⋄ Win.Pic.Img.scale←5 ⋄ Win.Pic.align←¯1
[5]   Win.Pic.Img.file←'c:\pictures\background.jpg'
[6]   ⍝ Create a second (invisible) image object:
[7]   Img2←'⎕' ⎕new 'Image'
[8]   Img2.file←'c:\pictures\foreground.png'
[9]   Img2.Transform 'Flip'
[10]  handle←Img2.handle
[11]  Win.Pic.Img.Overlay handle 200 120
[12]  ⍝1 ⎕WE Win
      ∇
```

The 'mode' parameter

The underlying ImageMagick call which does the copying is `MagickCompositeImage()`. If you omit the final Mode parameter, this is called with the 'compose' type set to `AtopCompositeOp`, which means that the foreground image is placed over the background image. This is the most common requirement. However, there are other possible values which you can specify as the Mode parameter. These are described in the full APLX Version 5 documentation.

GetMail and SendMail classes

These now support decoding of common non-ASCII character sets in the header.

In addition, you can now set the `port` property for use with non-standard TCP/IP ports.

12. Component File Systems

APLX Version 5 increases the maximum size of files in the two component file systems (accessed either by system functions such as `⎕FCREATE`, `⎕FTIE` etc, or by using the primitive functions ⌹⍇⍒⍋).

In Version 4 and earlier, component files were limited to 2GB in 32-bit implementations of APLX, and 1024GB in 64-bit implementations.

In Version 5, the limit has been raised to 1024GB in all versions. In addition, the maximum number of component files (and native files) you can have open at any one time is increased from 50 to 250.

Although this change should be transparent for most existing code, there are some implications for the primitives and system functions which relate to file sizes:

- The increased file sizes mean that (in 32-bit APLX) the results from `⎕FSIZE` and `2 ⌹` may overflow into floating-point numbers for files bigger than 2GB.

- When setting the maximum file size quota using `⎕FRESIZE` or ⍁, on 32-bit APLs the maximum explicit size limit you can set is 2GB. If you wish to allow the file to be larger than 2GB, set the limit to 0, which means no file-specific limit. The maximum will then be 1024GB, or as much as the operating system and available disk space allow.

- Files newly created under Version 5 have the limits set to the maximum by default.

- Files created under Version 4 or earlier will still be limited to 2GB. To expand the file beyond this, use `⎕FDUP` to make a fresh copy of the file, and then `⎕FRESIZE` to set the limit to the maximum.