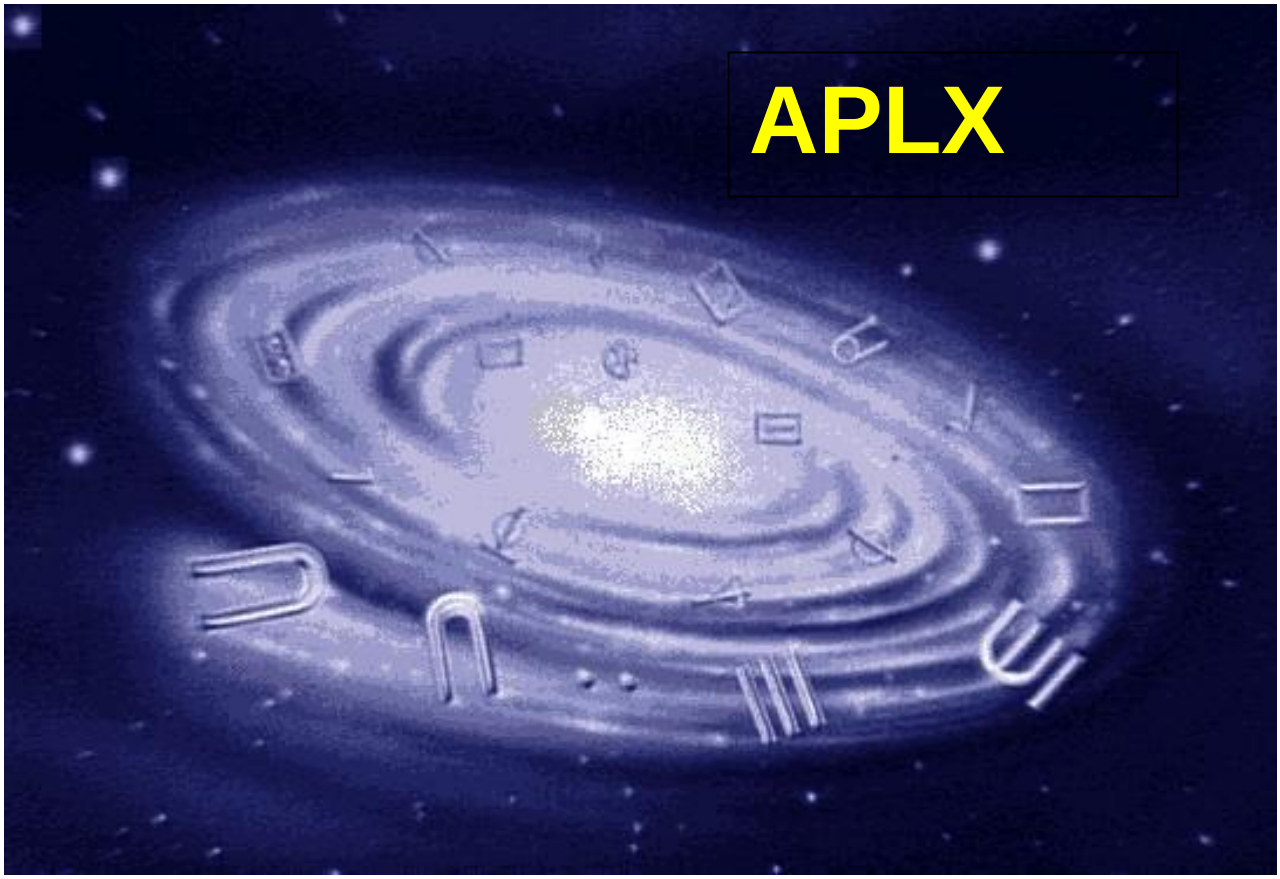


Learning APL with APLX

Version 5.0

MICRO APL

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

Version 5.0 July 2009

Contents

Introduction to APL

Origins of APL

APL began life as a notation for expressing mathematical procedures. Its originator, Dr Kenneth Iverson, published his notation in 1962 in a book called 'A Programming Language' from which title the name APL derives.

Subsequently Iverson's notation was implemented on various computers and did indeed become 'a programming language'.

The specification of the language was substantially enhanced during the 1980's, notably by Dr James Brown in IBM's APL2, giving rise to the term 'second generation APL' to cover those versions of APL that included the enhanced specifications.

The development of the APL language has continued in the 21st century. Whilst retaining compatibility with APL2, APLX Version 4 adds object-oriented language extensions, as well as a large number of other facilities, to the language.

Some features of the language

Data handling

The ability to handle generalized array data without complicated programming is one of APL's strongest points.

The language lets you define and give a name to a single item of data, a list of items, or a table of items, or indeed arrays of more dimensions. This means it's just as easy to write an instruction that adds two tables as one that adds two numbers.

The extra work caused by such an instruction is handled by APL internally. It goes through the named arrays of data, successively selecting and adding corresponding items, and it stops automatically when it finds that the items are exhausted.

Consequently there's usually no need for counts, loops or any of the mechanisms traditionally used to control such operations in other programming languages: the structure of the data effectively does this for you.

Power

APL has a powerful repertoire of *functions* (i.e. operations it can perform). These include a full range of sophisticated mathematical functions and a range of data manipulative functions which can do anything from a simple sort to a matrix inversion.

These functions can be combined so that on a single line you have a series of functions interacting dynamically, one function's results forming the next function's data.

APL also has *operators* which modify the way in which functions are applied. This gives you a range of general purpose building-blocks which can easily be combined in a very flexible and consistent manner.

This flexibility combined with APL's ability to handle complicated data makes it a uniquely powerful language.

APL Symbols

The syntax of most programming languages is restricted to the ASCII character set, which was originally designed for use in commercial data processing with very limited hardware, and which lacks even a basic multiply symbol. As a result, most programming languages resort to compromises such as using the asterisk symbol for multiply, and compound symbols such as <= to mean 'less-than-or-equal' or ** to mean 'power'. APL is not limited to the ASCII characters; instead, it uses a range of symbols to represent its built-in functions. This permits the APL user to carry out a wide range of programming tasks in a very concise manner.

There are no reserved words in APL. The symbolic nature of APL gives it an international appeal which transcends different national languages.

For many years, the use of special symbols by APL, and the special keyboards and display devices associated with them, was seen by many people as a big disadvantage of the language. Nowadays, with Unicode fonts and easy-to-use input methods, the special symbols are no longer such an issue.

Modularity

Besides giving you functions that are built in to the language, APL lets you define your own. In fact, what you might normally think of as a 'program' consists, in APL, of a collection of user-written functions each of which does a part of the total task.

These functions can 'call' each other and exchange data, but each is quite separate and self-contained. This has two very useful consequences.

First, debugging is much easier since each function can be tested separately and errors are easy to isolate.

Second, it imposes a natural top-down structure on your programs.

APLX takes the traditional modularity of APL further by borrowing the concepts of 'object-oriented' programming from other languages. This allows you to define classes (collections of data and related program logic), and to create and use instances of those classes (objects), in a natural APL style. You can even use classes written in other languages such as C#, Visual Basic, Java, or Ruby.

Convenience

APL is convenient to use for many reasons. In the first place, it's concise. A very few lines of APL can do a lot of work. Then it's an essentially interactive language. As you write each line or function you can try it out and get immediate feedback. It's also a fully *dynamic* language, which

means that you do not have to specify in advance what is valid for a given data item or function; you can just use it immediately.

In addition, it has a very useful concept called the *workspace*. This is basically a collection of the data items, functions, and classes which you set up in the course of doing a particular job.

The workspace is in computer memory while you work, making everything you want immediately accessible. It can be saved (i.e. copied on to a disc) in its entirety when you stop, and loaded back into memory next time you want to use it.

Ease of learning

APL is easy to learn in the sense that it's easy to get started. You'll be able to do simple things almost immediately.

Once started you can explore the language, discovering the functions and techniques that are of interest to you. APL supports this way of learning: it's totally consistent and does not have many arbitrary rules. And, as already said, it supplies immediate results and feedback.

What's more, because APL is different from conventional languages, it actually helps to be a computer novice!

Some professional programmers find it difficult to accept that APL handles complicated data without loops and counts. They're unused to the APL concept of a 'workspace' in which data and code rub shoulders. They fret about 'type-safety' and the fact that APL lets you do what you want. The functions written by the user behave neither like programs nor subroutines - what are they?

If you haven't programmed before, none of these questions will bother you. You'll accept the way APL does things as natural and convenient. For this reason, APL has traditionally been used by people who are not primarily computer programmers, but who need to write quite sophisticated programs in the course of their work or research - actuaries, engineers, statisticians, biologists, financial analysts, market researchers, and so on.

Productivity

Unlike many programming languages, APL wasn't designed to match the ways in which a computer works internally. It was intended to help people define procedures for solving problems.

This makes it efficient for people to use, though it may give the computer a little more work to do in the process.

This is reflected in development times. You'll find you can produce a working prototype very rapidly. Thanks to the modular approach encouraged by the language, and the ease with which modules can be tested, corrected and combined, it takes little extra time to turn this prototype into the final, fully tested version.

Entering APL Symbols

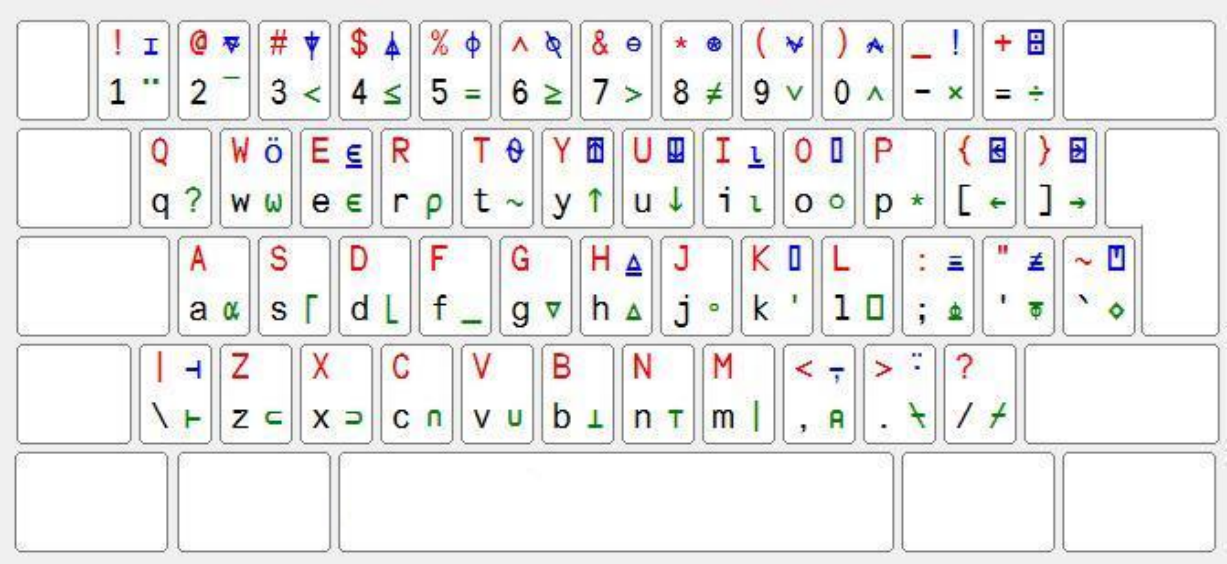
APL uses a variety of symbols to represent its built-in functions. Many of these are symbols that do not exist on a standard keyboard. Some of these will be familiar (for example, ÷ for divide, and ≥ for greater-than-or-equal). Others are specific to APL (for example ⌈ for round-up or greater-of, and ≡ for finding out if two arrays are exactly the same in shape and contents). In order to use APL, you need to be able to enter these symbols. Most versions of APLX provide you with two ways of doing this.

The QuickSym™ pop-up

If you are completely new to APL, you will probably find it easiest, at least to start with, to use the QuickSym™ feature of APLX. By pressing a single key (usually the 'Menu' key in Windows, or a function key - by default F1 - under MacOS), you can bring up a panel which displays the APL symbols. This is described in the next section.

Using an extended keyboard layout

The traditional way of entering special APL symbols is to use one of the extended keyboard layouts which APLX provides. You will probably want to use the 'unified' layout which is selected by default. This is configured as follows (the exact layout will vary according to the country in which you live and the specific model of keyboard):



As you can see, each key is shown with up to four symbols. As with a conventional keyboard, a given character key generates a lower case character and, when depressed in conjunction with the Shift key, the corresponding upper case character. These are shown in black and red on the above diagram, in the lower-left and upper-left positions on each key.

In order to enter the other symbols, you use the AltGr key, which acts just like the Shift key in that it modifies the effect of another key. (In non-APL applications, AltGr is used to type various different characters, primarily ones that are unusual for the locale of the keyboard layout, such as foreign currency symbols and accented letters.) Using the AltGr key with another key produces the symbol displayed in green, on the lower right of each key in the diagram above. The remaining symbols (shown in blue, on the top right of each key) are entered by pressing the AltGr and Shift keys simultaneously. (Note: On a Macintosh keyboard, use the Alt or Option key instead of AltGr).

For example, the key in the upper row of the keyboard, inscribed with the digit 4, produces the following symbols:

- If pressed by itself, the digit `4`
- If pressed with Shift, the dollar sign `$`
- If pressed with AltGr, the APL less-than-or-equals symbol `≤`
- If pressed with Shift and AltGr, the APL grade-up (sort) symbol `⍋`

If you have problems with the keyboard (e.g. because your keyboard does not have an AltGr key), see: http://www.microapl.co.uk/apl/aplx_support.html#keyboard

Choosing and customizing the keyboard layout

APLX supports three keyboard layouts. These are:

1) The **Unified** APL layout, as shown above. This is similar to an ordinary ASCII keyboard layout for unshifted and shifted keys. Special APL symbols are obtained by using AltGr or Shift-AltGr combinations. If you are learning APL we recommend that you use this layout.
2) The **Standard** or **Traditional** APL layout. This is based on the traditional APL keyboard where alphabetic keys unshifted give upper-case letters, shifted give APL symbols, and with the Alt or AltGr key give lower-case letters. It is recommended only for people who are used to programming in APL using a special APL keyboard.
3) The **Default non-APL** layout. In this mode, keyboard mapping is the same as in non-APL applications. For example, in the US it would usually be the ordinary US QWERTY keyboard. In France it will usually be the French AZERTY keyboard.

You can select which layout you want to use by selecting the Keyboard Layout item in the Tools menu of APLX's Session window. In addition, you can at any time swap between the Default non-APL layout and your preferred APL layout (Standard or Unified) by pressing Ctrl-N (or Cmd-N on the Macintosh). This is very useful for entering comments and text in APL functions, especially if you are using a language other than English

The current keyboard layout can be shown at any time by selecting the Keyboard Layout item in the Help menu. Under Windows, you can also display this by pressing Ctrl-F1.

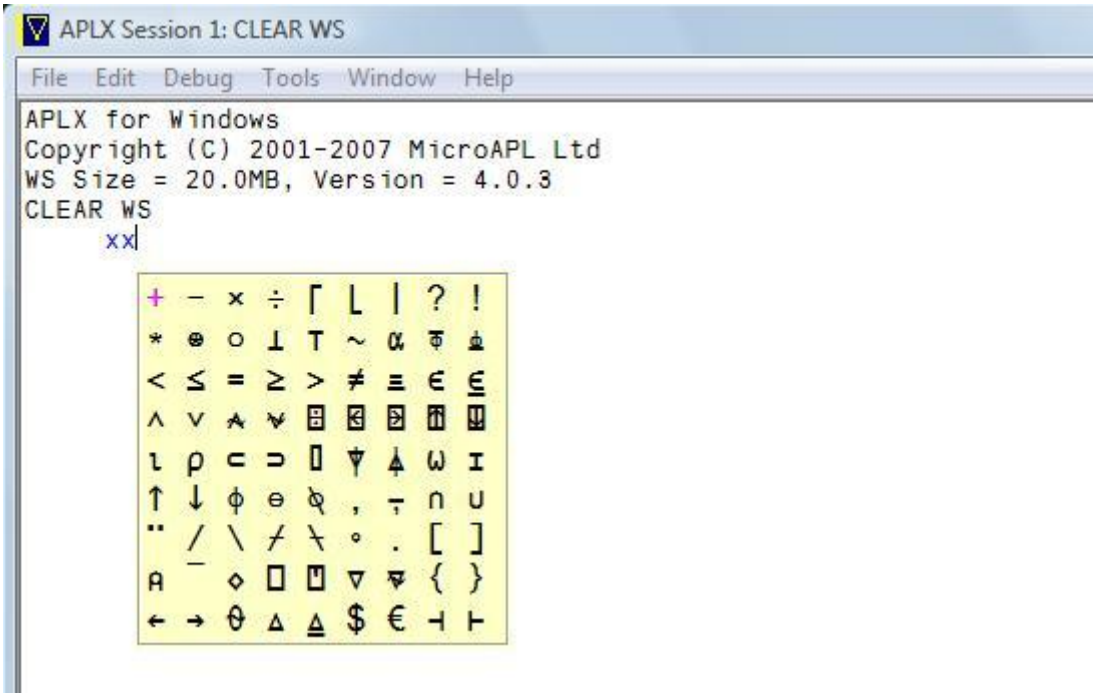
Customizing the keyboard layout

You can customize the keyboard layout using the Keyboard tab of the Preferences dialog. This allows you to move specific characters around on the keyboard.

The QuickSym pop-up

Bringing up the QuickSym window

The QuickSym™ feature of APLX allows you to enter special APL symbols without using a special keyboard layout. By pressing a single key (the 'Menu' key in Windows, which is usually to the right of the space bar next to the Ctrl key, or a function key - by default F1 - under MacOS), you can bring up a panel which displays the APL symbols:



You can now enter one or more APL symbols by using the mouse to click on the corresponding image in the QuickSym pop-up window. To assist in selecting the character, hovering the mouse over a symbol brings up a brief description of what it does (if it has one-argument and two-argument forms which have different functions, these will be shown on separate lines).

Using the keyboard with the QuickSym window

The QuickSym window is designed so that it is there when you need it, but gets out of the way when you don't. Once you have displayed the QuickSym window, the keyboard behaves as follows:

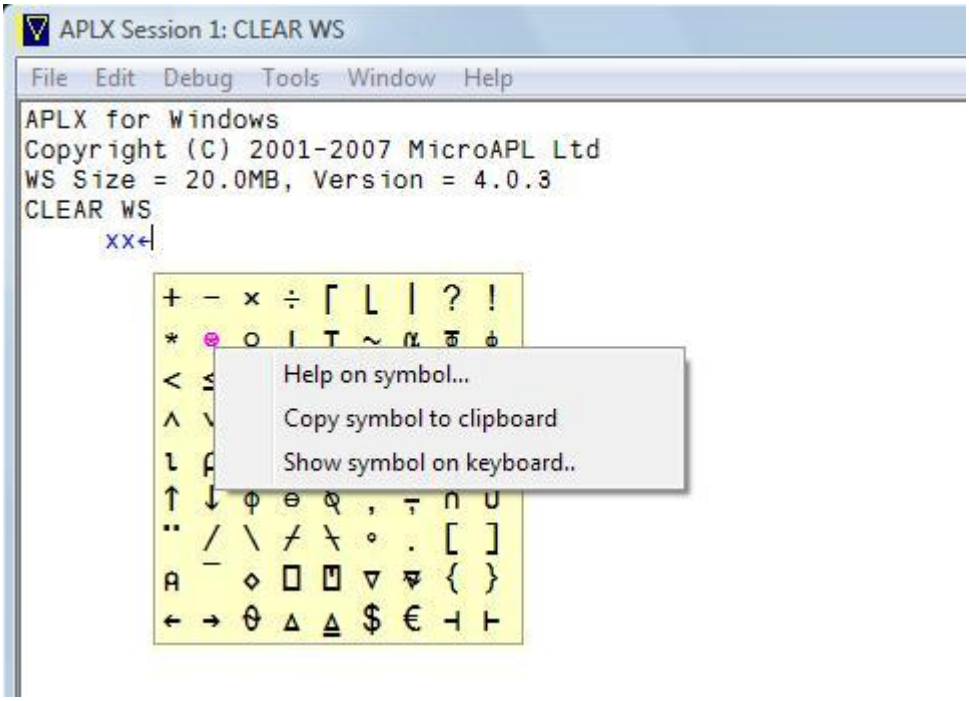
- Any key which produces a printing character behaves as normal. This means that you can freely mix typing at the keyboard, and using the mouse to enter symbols from the QuickSym window. For example, with the line shown in the picture, the user has typed xx and then brought up the QuickSym window. By using the mouse to select the assignment arrow ← (in the bottom left corner), and then typing 32, the line will read xx←32
- Pressing Enter (or Return) closes the QuickSym window. The keystroke is passed to the underlying window. Thus, in the above example, if the user now presses Enter, the

QuickSym window will disappear, and the line of APL xx←32 will be executed.

- Backspace and Delete act in the normal way, and do not close the QuickSym window. This means you can easily correct mistakes in the line you are entering.
- The cursor movement keys (Cursor Left, Cursor Right, Cursor Up, Cursor Down, Page Up, Page Down, and if appropriate Home and End) allow you to navigate around the QuickSym window by using the keyboard rather than the mouse. The currently-selected character will be highlighted in the display. Pressing Ctrl-Enter (Cmd-Enter on the Macintosh) inserts the currently-selected character in the line you are entering, without closing the QuickSym window.
- Pressing the Help key (F1 on Windows, Ctrl-H on Linux) brings up the APLX Help page for the selected character.
- The Escape key, or pressing the Menu or function key again, closes the QuickSym pop-up, without having any other effect.
- Any other key closes the QuickSym window, and is ignored.

The contextual menu

Right-clicking on a character in the QuickSym Window (or click-and-hold on the Macintosh) brings up a pop-up menu:

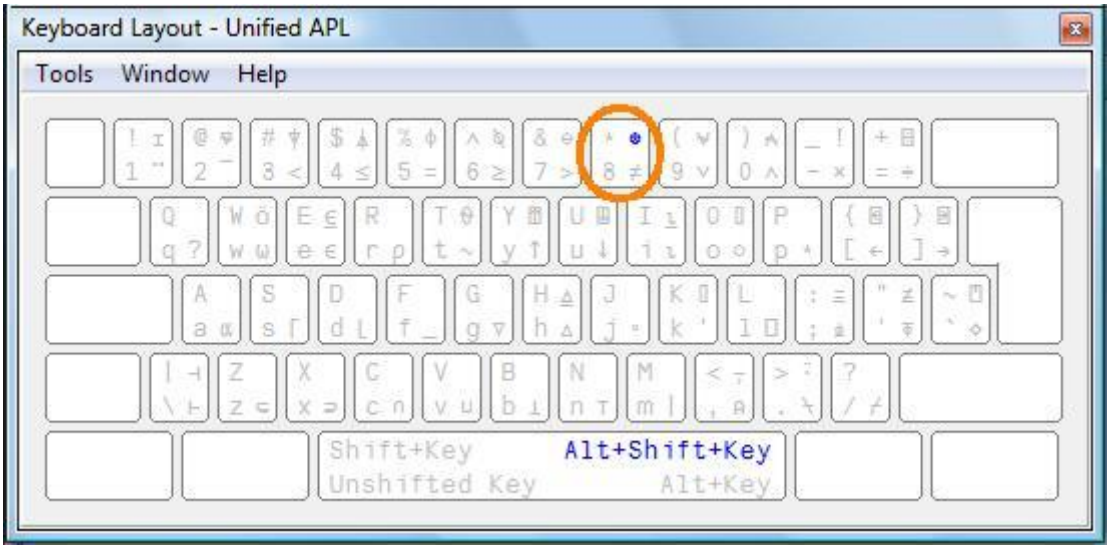


This offers the following options:

- 'Help on symbol' brings up the APLX Help page for the selected symbol (this is the same as pressing F1 under Windows, or the Help key under MacOS, when you have selected a symbol with the cursor keys). If the symbol has more than one Help page, there will normally be a link on the page which appears. If the symbol does not have a Help page, this

option will be disabled.

- 'Copy symbol to clipboard' puts the selected character into the clipboard. This is useful if you just want to copy an APL character to another application.
- 'Show symbol on keyboard' brings up the keyboard layout window, with the key on which the character appears highlighted. This is very useful if you are familiarising yourself with the APL keyboard layout, or are used to a different APL layout:



Simple Arithmetic

This is the first practical session. It consists mainly of examples for you to enter and comments on what happens when you do.

You'll be explicitly asked to type in the examples to begin with, but after a few pages it will be assumed that to see an APL statement is to type it, and the explicit instructions will be dropped.

To download a time-limited evaluation copy of APLX so you can try this, visit our Download page

The initial APLX display

The exact method of launching the APLX application will vary from system to system, and you should check the relevant implementation notes for the details. When you enter APLX you will see the first APL display, the one which includes the copyright statement, the version number, the workspace size and the message that the workspace currently in memory is called 'CLEAR WS'. The message should look like:

```
APLX for Windows
Copyright (C) 2001-2008 MicroAPL Ltd
WS Size = 20.0MB, Version = 4.1.6
CLEAR WS
```

(Exact details about the version number, WS size, etc, may vary)

APL starts in calculator mode which means that each statement you type is executed as soon as you press ENTER.

Some arithmetic functions

Type in this statement:

```
      5+12 <enter>
17
```

Pressing ENTER caused the statement to be executed.

As you can see, what you type is quite easy to distinguish from APL's response. Your input is inset by 6 spaces and if you are using a GUI version of APLX (Windows, Macintosh or Linux) it will also be in a different colour.

Add is one of the 50 or so functions built-into APL. (A 'function' is simply an operation, in this case, an arithmetic operation). Other familiar arithmetic functions which you'll use in this session are subtract (-) multiply (×) and divide (÷).

Try a couple of divisions:

```
      18÷3 <enter>
6
      108÷11 <enter>
```

```
9.818181818
```

Up to 10 digits are normally displayed in a result, though you can change this if you want.

Now type a multiplication:

```
      4×7 <enter>
28
```

And another with some decimal points:

```
      3.893×7.6 <enter>
29.5868
```

Subtraction too works as you would expect.

```
      100-95 <enter>
5
```

The normal minus always means subtract, as in the example you've just typed. The special 'high' minus sign indicates a negative number, as in this example:

```
      8-16 <enter>
¯8
```

Error Messages

Have you had any error messages yet? You may have if you've made a typing mistake, or if you've tried some of your own examples.

This example will produce an error message:

```
      23+ <enter>
SYNTAX ERROR
      23+
      ^
```

The text of the message depends on the error. In this case you've broken the syntax rules: 'number function number' makes sense to the APL interpreter. 'number function' does not.

The error marker (^) shows where the interpreter had got to in evaluating the statement when it found the error. This may help you identify the error, but obviously an error may not show up until something incompatible with it is found elsewhere. In such a case the error marker would be pointing to the correct but incompatible part of the statement.

There's a list of error messages and their likely causes in the APLX Language Manual. But for the time being don't worry too much about them. If an error message results from a typing mistake, simply retype the statement correctly. If it results from your own example and you can't see the cause, just try something else.

If you want to edit the last statement (rather than retype it) simply press Ctrl and Up-Arrow (or Command and Up-Arrow on the Macintosh, or on some systems Ctrl and R, or the 'last line recall'

key) and the statement will appear again with the cursor at its end.

You can try this now. Your screen at the moment should look like this:

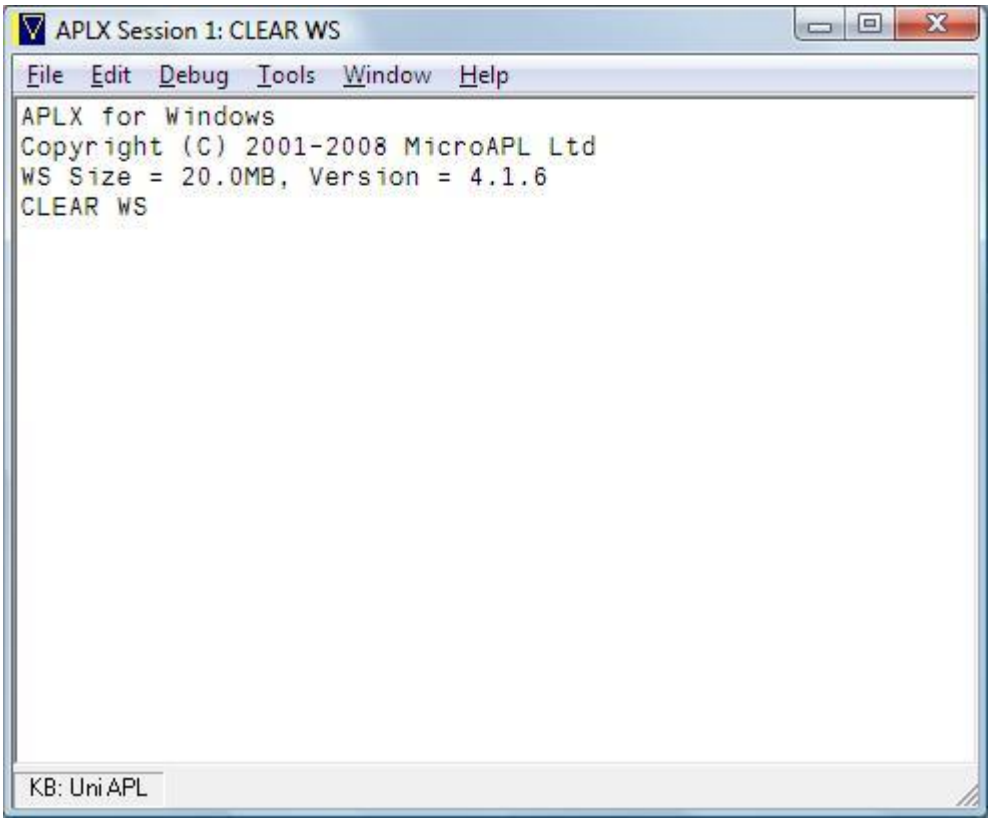
```
      23+ <enter>
SYNTAX ERROR
      23+
      ^
```

Now recall the line (Ctrl and Up-Arrow, or Command and Up-Arrow on the Macintosh) and complete the statement so that you have:

```
      23+8 <enter>
31
```

Using the Session Window

If you are using a GUI version of APLX (Windows, Macintosh or Linux), APL commands are entered in the Session window:



Normally, any text you type is added to the end of the session. If however there is already some text in the window which is exactly what you want to type in, or close to what you want, then there is no need to re-enter the text. You can select existing text on the screen, edit it, and then submit the line to APL.

To do this, you can move the mouse anywhere on the session window and click it. The text cursor (flashing vertical bar) will move to the position at which you clicked. You can now use the normal editing features of APLX such as the Delete key or enter further text, and you can also use the Edit

menu for more sophisticated editing such as cutting and pasting text. You can continue this process as much as you like - you are actually editing text *on the screen only* at this stage.

When you press ENTER then the current line - i.e. the line on which the cursor is flashing - will be re-displayed at the end of the session window, and submitted to APL. The line you changed will be returned to its former state.

Arithmetic on lists of numbers

Now we get on to something more interesting. Try this statement, making sure you include the spaces between the numbers:

```
      3+2 4 11 7 5
5 7 14 10 8
```

The number on the left of the sign has been added in turn to each number on the right of the sign.

Obviously, with lists of numbers, spaces **do** count. There's a great deal of difference between:

```
      1+2 3 4
```

and

```
      1+234
```

as you'll have seen when you typed them both in.

The list can be on either side of the sign. Here it's on the left and the single number's on the right:

```
      6 3 8 1+3
9 6 11 4
```

Now try some of the other arithmetic operations on lists. Here the divide function is used to divide each number in a list by 15:

```
      2.5 33.7 12 8÷15
0.1666666667 2.246666667 0.8 0.5333333333
```

And here's an example using the multiply function:

```
      9.8 11.2 17 1.2×1.175
11.515 13.16 19.975 1.41
```

In the last example you could be doing something useful such as multiplying a list of prices by 1.175 to give the prices including VAT (Value Added Tax) at 17.5%.

Matching up list lengths

So far the examples have involved a single number on one side of the arithmetic sign and a list of numbers on the other. But you can do arithmetic on two lists of numbers:

```
      12 3 29 4×1 3 5 2
12 9 145 8
```

The first number in the list on the left was multiplied by the first number in the list on the right, the second by the second and so on. But notice that the list on the left contained the same number of items as the list on the right.

Try this example to find out what happens if the lists don't conform in size:

```
      3 5+4 1 5
```

As you see, you get an error message like this:

```
LENGTH ERROR
      3 5+4 1 5
      ^
```

Since there are two numbers in one list and three in the other, the system doesn't know which number to add to which.

If a function operates on two lists they must both have the same number of elements. It's quite in order, though, to do an operation involving a single number and a list.

Order of execution

So far there's been no doubt about the order in which parts of a statement are executed because the examples haven't contained more than one arithmetic function. Now here's a statement which contains both multiply and subtract. Type it in, but decide what the answer will be before you press ENTER:

```
      3×3-1
```

Possibly you think the multiplication will be done first (either because it appears first in the line, or because you expect multiplication to be done before addition or subtraction). In that case you think the answer is 8.

Press ENTER and see.

```
      3×3-1
6
```

In fact APL always works from right to left. So it first did `3-1` giving `2`. Then it did the multiplication. (There's no hierarchy of operations in APL because there are too many functions in the language for one to be practicable - imagine trying to remember the order of precedence of fifty different operations!)

Here's another example of execution order. Again see if you can predict the answer before pressing ENTER:

```
      2 3 1+8÷2 2 2
```

The system first evaluates `8÷2 2 2` giving `4 4 4`. Then it does `2 3 1+4 4 4` giving `6 7 5`. Press ENTER and confirm this:

```
      2 3 1+8÷2 2 2
6 7 5
```

Parentheses

If you want to force a particular execution order, you can do so by using parentheses. Please retype the last example, using parentheses to force execution of the addition first:

```
      (2 3 1+8)÷2 2 2
5 5.5 4.5
```

Negative numbers and subtract

You saw earlier that the minus sign means 'subtract' and the high minus sign indicates a negative number. Here subtract is used:

```
      1985 - 1066
919
```

Here is an example with negative numbers in two lists which are being added together:

```
      3 ¯1 ¯7 + ¯4 ¯1 2
¯1 ¯2 ¯5
```

The minus-sign used to indicate negative numbers is known as the 'high-minus', and helps to make clear the difference between a negative number and the subtraction operation. The 'high-minus' is usually found above the number '2' on the keyboard.

These two examples illustrate this:

```
      2-3+5
¯6
      2 ¯3+5
7 2
```

In the first example, the sum of `3` and `5` was subtracted from `2`. In the second example, `¯3` was interpreted as a negative number. So `2 ¯3` was treated as a two-element list and each element was added to `5`.

Next we're going to consider a feature which effectively doubles the number of functions at your disposal. Then we're going to round things off with two new functions.

Dual-purpose functions

The examples you've seen so far have taken the form:

number(s) function number(s)

For example:

```
5+4   or   1 3 4+3 1 6
```

Each number, or list of numbers, operated on by a function is said to be an argument, so all our examples have shown a function operating on two arguments.

You may disagree, thinking perhaps of examples like this: `3x3+1`

In fact each function in that example does have two arguments. The subtract function has the arguments `3` and `1`. The Multiply function has `3` and the result of `3+1`.

There is, however, an alternative to having an argument on either side of the function. You can use any of the arithmetic functions in this format instead:

function number(s)

For example:

```
+12  or  ×4 6 3  or  ÷13 4 7
```

But when used in this way the functions have a different effect. Before reading any further please experiment with `+ - × ÷`, putting a number, or list of numbers, on the right-hand side only. See if you can decide what (if anything!) each function appears to do.

****

What did you conclude? You probably weren't unduly impressed by `+` since it appears to do nothing. We'll find a use for it though in the next chapter. You'll have discovered that minus effectively reversed the sign of everything to its right:

```
      - 3 ¯6 ¯8 4 12 ¯9
 ¯3 6 8 ¯4 ¯12 9
```

The action of `÷` may have puzzled you:

```
÷1 2 4 10 100
1 0.5 0.25 0.1 0.01
```

In fact `÷` used in this way yields the reciprocal of each number on its right (i.e. the result of dividing 1 by that number).

To put it another way, you can think of `÷3 6 2` as being equivalent to `1÷3 6 2`.

You may have concluded that `×` simply produced lists of `1`s:

```
      x2 66 8 2 13 109
1 1 1 1 1 1
```

This example gives a clue to its real purpose:

```
      x8 0 ¯3 ¯7 0 4
1 0 ¯1 ¯1 0 1
```

It returns a value of `1` for each positive number, `¯1` for each negative number and `0` for `0`s.

Now it doesn't really matter whether you remember what each of these signs does when used in this 'one-sided' format - you can always look at the APLX Language Manual if the need arises. What does matter is that you appreciate that many APL functions are capable of two interpretations. This flexibility effectively doubles APL's repertoire.

A small point before we move on.

When you typed the following expression earlier in this chapter you got an error message:

```
      23+
SYNTAX ERROR
      23+
      ^
```

This is because though the pattern 'function number(s)' is recognised by the interpreter, 'number(s) function' is not.

Ceiling and Floor (⌈ and ⌊)

To round things off we'll look at two functions that find maxima and minima. These are ⌈ (usually called **Ceiling** or **Max**) and ⌊ (usually called **Floor** or **Min**), and they are usually found above the 'S' and 'D' keys respectively.

We've had one or two examples like this:

```
      145÷11
13.18181818
```

Examples, that is, that produced a lot of decimal places. Now the eight places following the point in this example may be essential, but if they aren't, it's easy to get rid of them.

The ⌈ function **rounds up** to the nearest whole number.

The ⌊ function **rounds down** to the nearest whole number.

```
      ⌈13.18181818
14
      ⌊13.18181818
13
```

They both work on lists as well as on single numbers:

```
      ⌈120.11 12.32 65.01 13.52
121 13 66 14
      ⌊99.99 12.82 15.39 48.90
99 12 15 48
```

Since these functions simply round up or down to the nearest whole number, in the first example `120.11` is rounded up to `121` and, in the second, `99.99` is rounded down to `99`.

If you want greater accuracy in rounding, a simple piece of arithmetic will achieve it:

```
      ⌈120.11 12.32 65.01 13.52 - 0.5
120 12 65 14
```

Now `0.5` is subtracted from each number in the list before rounding up is carried out, thus ensuring a truer result. Rounding down can be corrected in a similar way by adding `0.5` to the numbers to be rounded:

```
      ⌊99.99 12.82 15.39 48.90 + 0.5
100 13 15 49
```

Possibly you've noticed that all the examples for ⌈ and ⌊ have taken the one-sided form:

function number(s)

Used with two arguments, these functions have a different meaning. Try out some statements of the form:

number(s) function number(s)

and decide what the two-argument versions of ⌈ and ⌊ do.

******

As you no doubt discovered, ⌈ selects the bigger of two numbers:

```
      2 ⌈ 6
6
```

while ⌊ selects the smaller:

```
      2 ⌊ 6
2
```

If ⌈ is used on lists of numbers, the first number in list 1 is compared with the first number in list 2, the second with the second and so on. The 'winner' in each comparison is displayed as the result:

```
      6 8 1 ⌈ 3 5 9
6 8 9
```

The equivalent procedure occurs with ⌊:

```
      6 8 1 ⌊ 3 5 9
3 5 1
```

Summary

We won't summarise everything: you can refer to the APLX Language Manual for definitions of the functions we've covered so far (`+ - × ÷ ⌈ ⌊`).

There are, however, three points made in this chapter that are essential to your understanding of APL:

1) APL always works from right to left, passing intermediate results along the line to the left as it works them out.

2) A single number and a list can be involved in an operation, but if two lists are involved, they must be the same size (i.e. have the same number of elements).

3) Many functions can be used with either one or two arguments. The number of arguments determines what the function does.

Practice

See what you can do with the functions covered so far. If you run out of ideas, why not look up '`/`' in the APLX Language Manual? It greatly increases the scope of the arithmetic (and other) functions. It will, however, be covered in a later session so don't feel you have to master it now. When you've tried things out to your satisfaction, do the problems on the next page.

When you want to end the session, type:

```
)OFF
```

Problems

Q1. Enter statements to:

a) Multiply each of the three numbers, `3 6 2` by `8` and then add `4` to the results of the multiplication.

b) Add 15% to each number in the list `14 5 78 145`.

c) Add the difference between `13` and `8` to `4 6 12 7`.

d) Multiply the result of `6` times `3` by the result of `4` times `8` and subtract `5` from the total.

e) Reverse the signs in this list: `3 ¯4 ¯12 6`

f) Compare these lists, selecting the larger number in each comparison:

```
2 7 0 55      33 1 10 13
```

Q2. Which of these statements cause error messages? Why?

a) `12 x9`

b) `3+¯2`

c) `19 0 3 4÷7 2 87`

d) `5 ¯8`

Q3. You're getting £200 worth of Dollars for yourself and £180 and £230 worth respectively for two friends. Enter a statement which calculates how many dollars each of you will get at 1.96 dollars to the pound.

Q4. Highest recorded temperatures for a week in August were:

79 84 83 78 74 69 70 (Fahrenheit)

Enter a statement to convert them into Centigrade. (One method is to subtract 32 degrees and multiply by 5/9.) Suppress decimal places in the result.

Q5. Enter a statement to find the difference in yards between 1500 metres and a mile. (1 yard = 0.9144m and 1760 yards in a mile)

Answers

(Your results should be the same. Your APL may be different.)

Q1 a)

```
      4+3 6 2 × 8
28 52 20
```

b)

```
      14 5 78 145×1.15
16.1 5.75 89.7 166.75
```

c)

```
      4 6 12 7+13-8
9 11 17 12
```

d)

```
      ¯5+(6×3)×4×8
571
```

e)

```
      - 3 ¯4 ¯12 6
¯3 4 12 ¯6
```

f)

```
      2 7 0 55 ⌈ 33 1 10 13
33 7 10 55
```

Q2. The only one to cause an error message is:

c)

```
      19 0 3 4÷7 2 87
LENGTH ERROR
      19 0 3 4÷7 2 87
      ^
```

(The lists aren't the same size.) d)

```
      5 ¯8
5 ¯8
```

APL thinks it's a two-element list. If the intention was to subtract the value `8` from `5`. the following would have been more effective:

```
      5-8
¯3
```

Q3.

```
      200 180 230 x 1.96
392 352.8 450.8
```

Q4.

```
      ⌈ ¯0.5+(5÷9)×79 84 83 78 74 69 70-32
26 29 28 26 23 21 21
```

(Work through it from right to left if it doesn't immediately make sense to you.)

Q5.

```
      1500-1760×.9144
¯109.344
```

Variables

As in the last practical session, please type in the examples as you encounter them. If you don't, you'll find parts of the session difficult to follow.

Assignments

This is an assignment:

```
      A ← .175
```

A location or 'variable' called `A` is set up in memory and the value `0.175` is assigned to it. An assignment doesn't automatically cause the value to be displayed, but it's there. If you want it displayed, just type the variable's name:

```
      A
0.175
```

Now `A` can be used in any expression in which `0.175` could be used. Here are three examples:

```
      200 × A
35
      A × 30.50 12,25 60.30 15.00
5.3375 2.14375 10.5525 2.625
      ⌈ A × 30.50 12.25 60.30 15.00
6 3 11 3
```

Variables are convenient if you want to use the same value in a number of operations, especially if it's a value that's a nuisance to key in, like this one:

```
      C ← .45359237
```

The variable `C` now contains a conversion factor for converting pounds to kilograms. (1 lb = 0.45359237 kg). You can use `C` whenever you want to do such a conversion; for example, to find out how many kilos there are in 17 lbs:

```
      17 × C
7.71107029
```

Or to calculate how many kilos there are in 11 stones and round up the answer:

```
      ⌈C×11×14
70
```

Variables are also useful if you want to keep results. The result of the calculation you've just done may still be on the screen, but it isn't preserved in the computer's memory. If you had wanted to keep it you could have typed:

```
      JOE ← ⌈C×11×14
```

The result is now saved in JOE. To see it type:

```
      JOE
70
```

Variable names

That last example showed that names needn't be a single letter. See if you can deduce some of the other rules about variable names by typing in the following statements and seeing which ones produce error messages or unexpected results:

```
      AAA ← 4
      ab ← 1
      5B ← 12
      C9999 ← 0
      JOHN SMITH ← 100
      JILL▵SMITH ← 100
      Jack_Smith ← 100
```

One error was:

```
      JOHN SMITH ← 100
VALUE ERROR
      JOHN SMITH ← 100
      ^
```

That particular error occurred because of the space between JOHN and SMITH. APLX assigned the value 100 to SMITH (type it again and type SMITH to see) but got stumped by the name JOHN, for which there is no value set at the moment.

Another was:

```
      5B ← 12
5 12
```

This time, B was given the value 12 and then the value of B was displayed after the number 5. This point is covered in more detail later in this chapter.

Names can't contain spaces or any other symbol except the Delta (▵), underlined Delta (⍙), the underline (_), or the high minus (¯). You will have found, however, that they can contain numerals, though not as the first character. (If you want to check up on these rules at any time, you'll find them in full in the APLX Language Manual.)

It's possible to produce duplicate variable names in the same workspace under special circumstances, as you'll find out when you write a user-defined function. But in calculator mode, names have to be unique; if you assign a value to A then decide to set up another variable A containing something else, all you'll do is give the original A a new value.

Assigning lists to variables

So far we've assigned single numbers to variables. This example puts a list of numbers into a variable called `PRICE`:

```
      PRICE ← 12.45 5.60 5.99 7.75
```

And this statement multiplies the numbers in `PRICE` by the variable `A` (to which you earlier assigned the value `0.175`) and puts the results in `VAT`.

```
      VAT ← PRICE × A
```

To see the contents of VAT type:

```
      VAT
2.17875 0.98 1.04825 1.35625
```

Incidentally, in the last chapter we mentioned that the one-argument version of `+` did have a use. It's a declarative, that is, it causes a value to be declared (i.e. displayed). In the previous example, if you'd wanted to see the contents of `VAT` without having to type its name, you could have put a plus in front of it:

```
      +VAT ← PRICE×A
2.17875 0.98 1.04825 1.35625
```

You've seen two ways of setting up a variable. One is to directly assign a value to it; the variable `A` was set up in this way by the statement `A ← 0.175`. The other is to assign a result to it: the variable `VAT` was set up in this way by the statement `VAT ← PRICE × A`.

What you can't do is use a variable name that hasn't had a value assigned to it. See what happens if you type:

```
      A-BB
VALUE ERROR
      A-BB
        ^
```

No value has been assigned to variable `BB` (unless you've used `BB` in your own examples). APL therefore declares a `VALUE ERROR`. How can you check on what variables do exist in your workspace? This is a convenient point at which to introduce a couple of system commands.

System Commands

Loosely speaking, system commands manage the environment in which you work rather than doing calculations or manipulating data. You've already met one system command (don't type it unless you want to end the session!):

```
      )OFF
```

The right parenthesis identifies system commands. If you use `OFF` or any other system command without this parenthesis, it won't work: the interpreter thinks it's a variable name and (probably)

produces an error message.

Here's a system command which lists the names of the variables you've got in your workspace:

```
      )VARS
A       C       JOE     PRICE   VAT
```

Your list may be different if you've set up all the examples and some other variables.

Now try this one;

```
      )WSID
CLEAR WS
```

You've asked the identity of the workspace and the system has told you it's called `CLEAR WS`.

The same command can be used to change the identity of the workspace. Here you're changing its name to `NEW`:

```
      )WSID NEW
WAS CLEAR WS
```

The system seems more concerned about the old name, but it has in fact carried out your request as you'll find if you type:

```
      )WSID
NEW
```

Though the name has changed, the variables are still there as you'll see if you again type:

```
      VARS
A       C       JOE     PRICE   VAT
```

We don't need to keep these variables, so we'll use another system command to clear the workspace:

```
      )CLEAR
CLEAR WS
```

The variables are no longer there, as you can confirm by typing:

```
      )VARS
```

And the name of the workspace has reverted to `CLEAR WS`, as you'll find if you again type:

```
      )WSID
CLEAR WS
```

We'll cover other system commands in later sessions. For now we'll return to assignments.

Character assignments

You could be forgiven for thinking that APL deals only in numbers. But it isn't so. Try this assignment, taking care to put the single quote mark round the piece of text:

```
      A ← 'APL WILL PROCESS TEXT'
```

Now check what's in `A`:

```
      A
APL WILL PROCESS TEXT
```

The characters between the quotes have obviously been put into `A`. If you omit the quotes, APL thinks it's dealing with an undeclared variable:

```
      C ← CHARACTERS
VALUE ERROR
      C ← CHARACTERS
          ^
```

This time include the quotes, then check that the assignment has worked:

```
      C ← 'CHARACTERS'
      C
CHARACTERS
```

In fact, APLX caters for the forgetful as far as quotes are concerned. If you put an opening quote, type your text, then press ENTER without putting the closing quote, APL will insert it for you:

```
      MORAL ← 'WELL BEGUN IS HALF DONE
      MORAL
WELL BEGUN IS HALF DONE
```

This prevents succeeding lines being treated as one very long text string, simply because you've forgotten to close a quote. You can use any characters on the keyboard in a character assignment. That includes space (which is a character, albeit an invisible one) and all the symbols. You can even include the single quote character itself, though that requires some special action.

Try this assignment:

```
      NAME ← 'WHAT'S IN A NAME? '
```

As you probably expected, you got an error message. APL can't distinguish between the apostrophe in the text and the single quote marks round the text. If you need to include an apostrophe (i.e. a single quote) insert another single quote beside it like this:

```
      NAME ← 'WHAT''S IN A NAME? '
      NAME
WHAT'S IN A NAME?
```

Alternatively, you can use double quote marks like this:

```
      NAME ← "WHAT'S IN A NAME? "
      NAME
WHAT'S IN A NAME?
```

APLX allows you to use single or double quote marks round the text. The opening and closing quote marks must be of the same type; a quote mark of the other type doesn't end the text, as the example above shows.

Now let's set up a variable containing the characters, `NET PRICE`:

```
      N ← 'NET PRICE'
```

Obviously a variable containing characters can't be used in arithmetic, but try this all the same:

```
      N×10
```

Well, as you saw you got a new kind of error:

```
      N×10
DOMAIN ERROR
      N×10
      ^
```

The domain of characters and the domain of numbers have different rules and their members don't mix. This applies even when characters look like numbers. Type this assignment, taking care to include the quotes:

```
      QTY ← '230'
```

The fact that `230` is in quotes ensures it is treated as three characters with no more numerical significance than `ABC`. You can prove this by trying to use it numerically:

```
      QTY+5
DOMAIN ERROR
      QTY+5
      ^
```

Multiple assignments

If you are getting a bit tired of making all these individual assignment statements, APLX allows you to set up several variables at the same time. Try typing:

```
      (ZAK YAK) ← 5
```

then see what values have been assigned to `ZAK` and `YAK`. They both should have the value `5`. Each separate name inside the parentheses has been given the value `5`. If you want to set up some other variables with different values, try typing:

```
      (YEN MARK BUCK) ← 10 20 30
```

There you are, three at a single blow!

Displaying variables together

Though you can't use character and number variables together in arithmetic, they are prepared to appear together in displays.

```
      N 10
 NET PRICE 10
```

(If your version doesn't contain the same space between `PRICE` and `10`, it's because you may have left more or less space between `PRICE` and the closing quote when you made the assignment to `N`.)

Here two character variables are displayed in tandem:

```
      NAME C
 WHAT'S IN A NAME?  CHARACTERS
```

And this example shows two numeric variables displayed together (but first you have to set them up):

```
      X ← 18
      Y ← 3 1985
      X Y
 18  3 1985
```

In this example we're mixing domains again:

```
      NAME X C
 WHAT'S IN A NAME?  18 CHARACTERS
```

That statement is true: the variable, `NAME`, does contain 18 characters, counting all spaces (including the one at the end). As you probably realise, any spaces you want have to be included between the quotes. Inserting them between the variable names like this achieves nothing:

```
      C          C          C
 CHARACTERS CHARACTERS CHARACTERS
```

You can use characters in APL without putting them in a variable first. For example:

```
      'NET PRICE: ' 10
 NET PRICE:  10
```

Joining lists together

The way we've been displaying lists together is more powerful than you may think. We've treated it up till now as a means of displaying the contents of lists together. In fact it actually joins lists so that two single numbers form a two-element list, or two character strings form one two-element character object.

So when you entered the statement `X Y` to produce the display `18 3 1985`, you in fact produced a two-element list which could be used in arithmetic. You can prove this now by typing:

```
      1+X Y
 19  4 1986
```

If you wanted to use the list formed by `X` and `Y` more than once, you could assign it to a variable:

```
      Z ← X Y
      Z
 18  3 1985
```

This has the advantage that `X` and `Y` exist independently as well as being components of the list which forms `Z`. So operations done to `Z` don't affect them. The following example adds `10` to `Z` and then displays `Z X` and `Y` to show that only `Z` has been affected:

```
      Z ← Z+10
      Z
 28  13 1995
      X
18
      Y
3 1985
```

Here's an example with characters for a change:

```
      CNAME ← 'BASIL '
      SNAME ← 'BRUSH'
      NAME ← CNAME SNAME
      NAME
 BASIL BRUSH
```

This example is actually rather complicated, as the contents of `NAME` are actually only **two** elements. The first element is a list which contains the characters `BASIL` and the second is a list which contains the character list `BRUSH`. This is an example of a special type of variable known as a **nested** variable - we'll be discussing this later in more detail when we've covered more of the basics (`Z` was nested too). If you want a preview of a later section, try using ⍴ (called **rho**) in front of some of these variables and see what it does.

Joining and merging variables

If we want to join the characters in the last example to form a single list, we have to use a new symbol , (comma). The comma tells APL to merge the data on its left and right. Let's try the last example again:

```
      NAME ← CNAME,SNAME
      NAME
BASIL BRUSH
```

This time `NAME` is a list of 11 characters. In the next chapter we'll be looking at how to use some new symbols to tell the difference between this simple variable and its nested variant. The comma

is in fact a function in its own right, and the function is performs is called **catenation**.

Simple and nested variables

So how does APL work out which variables or lists are lists of lists (which is another way of describing our nested variable) or just simple lists?

There are some easy rules to follow when typing in data.

Single numbers and characters are interpreted as making up simple lists. So here is a list of 4 numbers:

```
PIERRE ← 1 2 3 4
```

and here is another list of five characters:

```
MIREILLE ← 'FILLE'
```

If some of the numbers are enclosed by parentheses they are treated as a single item in the resulting list:

```
PIERRE ← (1 2 3) (4 5 6 7)
```

So here `PIERRE` is a list of two lists, one of which is three long and one of which is four long. To make a list of character lists, all you need to do is to use groups of characters, each of which are enclosed by quotes. So to make up a list of three character lists:

```
FRANCOISE ← 'UNE'  'JEUNE'  'FILLE'
```

Mixed variables

Just to complete the story, APLX allows a variable to have both character data and number data in it. You won't be able to carry out arithmetic on the character part, but this is a useful way of storing characters and numbers that 'belong' together:

```
PHONES ← 'BILL' 577332 'FRANK' 886331
```

`PHONES` will be four elements long, and will be alternately character lists and single numbers.

Summary

The symbol ← assigns numbers or characters to a named variable. The right argument is the value to be assigned. The left argument is the name of the variable.

Variable names are formed from any combination of the letters of the alphabet (in uppercase or lowercase) and the numerals 0 to 9, together with the characters ∆ ⍙ _ and ¯. They must start with a letter or ∆ or ⍙.

Any numeric value can be assigned to a variable. This includes a single number, a list of numbers

or the result of a calculation.

Any character on the keyboard can be assigned to a variable.

Character strings assigned to a variable are enclosed in single quotes. To include a single quote in a character string, type another single quote immediately beside it, or use double quotes.

Character variables can't be used in arithmetic.

More than one variable can be assigned at the same time.

The contents of variables can be displayed together merely by typing their names together.

Variables can contain a mixture of numbers and characters.

Variables are made up of single numbers or characters, in which case they are called simple variables, or their elements can be lists themselves, in which case they are called nested variables.

The comma performs the catenate function. Variables joined by it can be treated as a single variable.

System commands manage the environment provided by the system. The following were mentioned in this chapter:

```
)VARS
)WSID
)CLEAR
)OFF
```

Practice

Please experiment with setting up and displaying character variables for a few minutes before you do the problems. Clear the workspace when you've finished,

Problems

Q1. Enter statements which:

a) Assign the numbers `22 2 2007` to three variables called respectively `D M` and `Y`.

b) Assign the characters `TODAY'S DATE:` to a variable called `DATE`.

c) Produce the display: `TODAY'S DATE: 22 2 2007` (Use the correct date if you prefer.)

Q2. Set up a variable `CONV` which contains a constant for converting pounds to kilos. (1 lb = 0.454 kg and 14 lb = 1 stone) Use `CONV` to convert your weight (to the nearest stone) into kilograms. Reduce the result by 10%, round it down, and display it.

Q3. The cost prices of four items of stock are `£8 6 12 4` respectively. The markup on these items is 100%. Three other items cost respectively `£16 13` and 7. Their markup is 75%. Calculate the fully inclusive price of each item (with VAT at 17.5%). Display the prices (rounded up) with the

caption:

```
      'PRICE+VAT: '
```

Q4. `TEST1` contains a student's exam marks for each of seven subjects (65 72 54 80 67 60 59). `TEST2` contains his marks for the same subjects gained at a different test (75 70 60 74 58 61 50). Produce a list consisting of his higher mark for each subject.

Q5. Which of the following will produce error messages? Why?

a) `RATE ← '3.7x3'`

b) `10+10 '←21'`

c) `100×RATE`

d) `SYMBOLS ← '¯<≤=≥'`

e) `3+'232'`

Answers

Q1. a)

```
      D ← 22
      M ← 2
      y ← 2007
```

or

```
      (D M Y) ← 22 2 2007
```

b)

```
      DATE ← 'TODAY''S DATE:'
```

or

```
      DATE ← "TODAY'S DATE:"
```

c)

```
      DATE D M Y
TODAY'S DATE: 22 2 2007
```

Q2.

```
      CONV ← .454
      ⌊ .9×CONV×13×14
74
```

(the weight used in this calculation was 13 stone,)

Q3.

```
      LIST1 ← 2×8 6 12 4
      LIST2 ← 1.75×16 13 7
      VATPRICE ← ⌈ 1.175× LIST1,LIST2
      'PRICE+VAT: ' VATPRICE
PRICE+VAT: 19 15 29 10 33 27 15
```

Q4.

```
      TEST1 ← 65 72 54 80 67 60 59
      TEST2 ← 75 70 60 74 59 61 50
      TEST1 ⌈ TEST2
75 72 60 80 67 61 59
```

Q5.

b) produces a `DOMAIN ERROR`. You are trying to add `10` to the number `10` and also the characters `←21`. It is this last bit that doesn't work.

c) produces a `DOMAIN ERROR` - `RATE` was defined as a string of characters in example (a) and you can't multiply characters and numbers.

e) produces a `DOMAIN ERROR`. You cannot add numbers and characters.

Tables

This is another practical session, so please continue to type the examples you see in the manual.

The general subject is tables, and the first topic in connection with tables is how to set them up. That's a topic that could involve you in a lot of keying. (Imagine typing in 50 different values to fill a modest five-row by ten-column table!) To avoid such drudgery we'll look first at two functions that will generate the numbers for you.

The ? function

The ? function (usually called **Random**, **Roll** or **Deal**) generates random numbers. Here's an example:

```
      ? 100
53
```

You asked for a random number in the range 1 to 100. You probably didn't get 53. That was the one-argument form of ? It returns a single number between 1 and the number you specify as the right-hand argument.

The two-argument form will be more useful for filling tables because it generates as many numbers as you ask for:

```
      50 ? 100
```

On your screen you should have 50 numbers in the range 1 to 100. Look at your numbers carefully and see how many duplicates you can count in 20 seconds.

******

Given up? In fact you won't find any. The ? function generates unique random numbers in the given range. That's why this example produces an error message:

```
      20 ? 10
DOMAIN ERROR
      20 ? 10
      ^
```

The domain of numbers in the range 1 to 10 can't supply 20 different whole numbers. You can use a variable as the right-hand argument of ? We'll set one up then use it :

```
      RANGE ← 15
      3 ? RANGE
1 5 13
```

Equally, you can use a variable to specify how many random numbers you want:

```
      QTY ← 7
      QTY ? RANGE
5 14 10 1 15 2 4
```

And you can assign the result to a variable too:

```
      BINGO ← QTY ? RANGE
      BINGO
6 14 9 2 3 11 6
```

The ⍳ function

This is an example of the ⍳ function (called **Iota** or **Index** and found above the 'I' key):

```
      ⍳100
```

Your screen should now be filled with the numbers from 1 to 100 in ascending order. We're using the one-argument form of ⍳. It generates the series of numbers from 1 to whatever number you specify as its right-hand argument.

Here's an example which puts the series from 1 to 10 in a variable called X:

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

Now we can safely tackle the topic of tables. But for clarity, we'll start by entering values explicitly, rather than generating them randomly or producing them with the ⍳ function.

Setting up tables

This statement will take the 12 numbers on the right of the ⍴ symbol, and set them up as a four-by-three table:

```
      4 3 ⍴ 10 20 30 40 50 60 70 80 90 100 110 120
 10  20  30
 40  50  60
 70  80  90
100 110 120
```

⍴ (called **Rho**, **Shape** or **Reshape** and found above the 'R' key) is a function and like any other function, it operates on arguments. We're using the two-argument form of ⍴.

(We'll see what the one-argument form does later in the chapter.)

```
      4 3 ⍴ 10 20 30 40 50 60 70 80 90 100 110 120
```

The argument to the left specifies how many rows and columns are in the table. The argument to the right specifies what data is to be put into the table.

Here again is the table produced by the last example, this time with the rows and columns labelled:

	col 1	col 2	col 3
row 1	10	20	30
row 2	40	50	60
row 3	70	80	90
row 4	100	110	120

You always specify the number of rows before the number of columns, and APL fills the table row-by-row rather than column-by-column. (This may seem a trivial point, but if APL first filled column 1, then column 2 then column 3, the effect would be quite different).

The data to be put in the table can itself be in a variable. This next statement puts 12 random numbers in the range 1 to 100 into a variable called `DATA`:

```
      DATA ← 12 ? 100
```

Now use the ⍴ function again, but specify `DATA` as the right-hand argument:

```
      4 3 ⍴ DATA
15 57 30
51 50 97
18 26 38
67 22 69
```

(Your numbers are unlikely to be the same!)

The next example looks doomed to failure because there are insufficient numbers on the right to fill a table of the specified dimensions. But try it anyway and see what APL does:

```
      4 3 ⍴ 1 2 3 4 5
```

As you saw, when the numbers ran out, APL went back to the first number and went through them again, giving a table like this:

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

It follows that if you supply only one number, APL will use that to fill the whole table:

```
      3 5 ⍴ 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
```

On the other hand, if you supply too many numbers, APL uses what it needs to fill the table and ignores the rest:

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

Try setting up some tables before you read on. Remember that you can use the `?` and `⍳` functions.

Arithmetic on tables

Now please set up a 3-row 3-column table called `SALES`, containing the numbers shown:

```
      SALES ← 3 3⍴20 13 8 30 43 48 3 50 21
      SALES
20 13  8
30 43 48
 3 50 21
```

Arithmetic on `SALES` will automatically affect every number in the table. For example:

```
      SALES×10
200 130  80
300 430 480
 30 500 210
```

Suppose you now set up another table called, say, `PRICES`, will you be able to do operations like `SALES` times `PRICES`? Let's find out:

```
      PRICES ← 2 3 ⍴ 21 2 12 47 33 1
      SALES×PRICES
```

The attempt caused an error:

```
LENGTH ERROR
      SALES×PRICES
      ^
```

A `LENGTH ERROR` message means that the arguments contain different numbers of elements. The problem, obviously, is that `SALES` is three rows by three columns, and therefore contains nine numbers, while `PRICES` is two rows by three columns, and therefore contains six numbers. How could APL know which items you want multiplied by which? Since it doesn't, it gives an error message.

Let's redefine the `SALES` table so that it too contains six items:

```
      SALES ← 3 2 ⍴ SALES
```

(We used the numbers already in `SALES` as data. The effect of the statement is to take the first six numbers in the old `SALES` table and rearrange them as three rows and two columns to form the new `SALES` table.)

Now that `SALES`, like `PRICES`, contains six numbers, let's try the multiplication again:

```
      SALES×PRICES
LENGTH ERROR
      SALES×PRICES
      ^
```

We're still getting an error message. If we look at both tables the problem will be apparent:

```
      SALES
20 13
 8 30
43 48
      PRICES
21  2 12
47 33  1
```

The tables may now have the same number of items, but they're still a different 'shape'. It's impossible to know, with any certainty, which item in `SALES` corresponds to which item in `PRICES`.

We'll have to redefine `SALES` again so that it has the same number of rows and columns as `PRICES`:

```
      SALES ← 2 3 ⍴ SALES
```

(Again we've used the numbers already in `SALES` as data for the new version of the table.)

Now we'll check that `SALES` and `PRICES` are the same shape (i.e. have the same number of rows and columns):

```
      SALES
20 13  8
30 43 48
      PRICES
21  2 12
47 33  1
```

They now match exactly in shape and size, so we can try again.

```
      SALES×PRICES
 420   26 96
1410 1419 48
```

Success at last! The elements in the two tables now match-up, or conform, and arithmetic involving both tables is possible. Let's check by trying another operation on them:

```
      SALES - PRICES
 ¯1 11 ¯4
¯17 10 47
```

That worked too. (Remember that the previous multiplication didn't change the contents of the tables.)

Incidentally, you don't have to create every table explicitly. You can create one simply by assigning a result to it. Here you're creating a table called `TOTAL`.

```
      TOTAL ← SALES×PRICES
      TOTAL
 420   26 96
1410 1419 48
```

Before you read on, practice constructing tables and doing arithmetic on them. Make use of the ? and ⍳ functions to set the tables up. Don't forget about the ⌈ and ⌊ functions. They work on tables too.

Catenating tables

You can catenate tables just as you catenated other variables in the previous session:

```
      SALES,PRICES
20 13  8 21  2 12
30 43 48 47 33  1
```

The tables being catenated must have the same number of rows, but don't have to have the same number of columns.

This next example creates a two-row two-column table called `LITTLE` filled with `1`s, and a two-row six-column table called `MEDIUM` filled with `5`s. Then it catenates the tables and puts the result in `BIG`:

```
      LITTLE ← 2 2 ⍴ 1
      MEDIUM ← 2 6 ⍴ 5
      BIG ← LITTLE,MEDIUM
      LITTLE
1 1
1 1
      MEDIUM
5 5 5 5 5 5
5 5 5 5 5 5
      BIG
1 1 5 5 5 5 5 5
1 1 5 5 5 5 5 5
```

Again, notice that though the catenated tables have different numbers of columns, they both have the same number of rows.

Catenation supplies one of many solutions to the problem of arithmetic being possible only on tables of equal size. Suppose you wanted to add `LITTLE` to `MEDIUM`:

```
      LITTLE+MEDIUM
LENGTH ERROR
      LITTLE+MEDIUM
      ^
```

You can't because they're different sizes. They both have two rows, but `LITTLE` has two columns while `MEDIUM` has six:

```
      LITTLE
1 1
1 1
      MEDIUM
5 5 5 5 5 5
5 5 5 5 5 5
```

The following example shows how `LITTLE` can be catenated with a table of zeroes to pad it out to the same size as `MEDIUM` so that the addition can take place:

```
      ZEROES ← 2 4 ⍴ 0
      ZEROES
0 0 0 0
0 0 0 0
      LITTLE ← LITTLE,ZEROES
      LITTLE
1 1 0 0 0 0
1 1 0 0 0 0
      LITTLE+MEDIUM
6 6 5 5 5 5
6 6 5 5 5 5
```

The addition took place successfully. Presumably we wanted the original `LITTLE` to be added on to the left-hand end of `MEDIUM`. If we wanted it on the other side we should have specified (resetting `LITTLE` first!):

```
      LITTLE ← 2 2⍴1
      LITTLE ← ZEROES,LITTLE
      LITTLE+MEDIUM
5 5 5 5 6 6
5 5 5 5 6 6
```

It's because that kind of ambiguity exists that APL won't do arithmetic on data of unequal size.

Selecting elements

You may be wondering how you select single elements from tables.

First set up this table (using the numbers shown rather than the `?` function), then we'll select individual elements from it :

```
      +TABLE ← 4 3 ⍴ 2 12 15 4 11 7 1 16 8 20 19 9
 2 12 15
 4 11  7
 1 16  8
20 19  9
```

Remember that the table consists of four rows and three columns.

To select the `9` in the bottom row, right-most column, type:

```
      TABLE[4;3]
9
```

You've used the row number <4>, and the column number <3>, to identify which element of the table you want. Before you read on, see if you can enter a statement which adds the number in row 3 column 3, to the number in row 4 column 2. Make sure you use the square brackets and separate the row number from the column number with a semicolon.

******

You should have entered:

```
      TABLE[3;3] + TABLE[4;2]
27
```

Now see if you can replace the number in row 3, column 2 with the result of adding the numbers in row 1, column 2, and row 2, column 2. Here's the table again with the numbers marked:

```
 2 12 15
 4 11  7
 1 16  8
20 19  9
```

******

That shouldn't have been difficult as long as you counted the rows and columns correctly, and remembered the semicolons. You no doubt typed:

```
      TABLE[3;2] ← TABLE[1;2] + TABLE[2;2]
```

Check `TABLE` make sure that row 3, column 2 now contains the sum of rows 1 and 2 column 2:

```
      TABLE
 2 12 15
 4 11  7
 1 23  8
20 19  9
```

It's quite easy to select entire rows or columns. Here we select all three elements in row 1:

```
      TABLE[1;1 2 3]
2 12 15
```

As before, the number before the semicolon denotes the row while the number, or in this case numbers, after the semicolon denote the column(s). There is, however, a shorthand way of selecting whole rows or columns. The following statement does the same as the last, that is, it selects all columns in row 1:

```
      TABLE[1;]
2 12 15
```

Using the same principle, see if you can replace the numbers in column 3 with the sum of the numbers in columns 1 and 2.

******

(In the course of some of these operations you may be getting an error message saying that you've made an `INDEX ERROR`. The process of picking elements out of a table is called 'indexing'. A mistake is therefore referred to as an 'index' error.)

To add the first two columns and put the result in column three you could have typed:

```
      TABLE[1 2 3 4;3] ← TABLE[1 2 3 4;1] + TABLE[1 2 3 4;2]
      TABLE
 2 12 14
 4 11 15
 1 23 24
20 19 39
```

Alternatively you could have used the shorthand way:

```
      TABLE[;3] ← TABLE[;1] + TABLE[;2]
```

You can, of course, select elements from two separate tables and do arithmetic on them. If you still have the tables `SALES` and `PRICES` in your workspace, the following statement will multiply the number in row 1 column 1 of `SALES` by the number in row 2 column 3 of `PRICES`.

```
      SALES[1;1] × PRICES[2;3]
20
```

Incidentally, indexing can also be used to pick single elements out of lists. With lists, of course, only one number is needed to identify the element required:

```
      LIST ← 8 1 90 4
      LIST[2]
1
```

Dimensions

A quick digression about dimensions is in order before we tackle the next topic. APL regards data as having dimensions.

- A single number, or character is like a point in geometry. It exists but has no dimensions.
- A list is like a line. It has one dimension, length.
- The tables we've looked at are like rectangles. They have two dimensions, height and length.
- Three-dimensional 'tables', or 'arrays', as they are more often called, are like cubes. They have depth, height and length.

Arrays of up to sixty three dimensions are possible in APLX, but we won't attempt to represent them!

The thought of three-dimensional data may intrigue you, but in practice it's quite mundane - as the next example will reveal. Suppose the ordinary two-dimensional table you're about to create

represents the number of each of four products sold by each of six salesmen:

```
      SALES ← 6 4⍴24?50
      SALES
 5 34 22 36
46 40 18 10
39 23  4 41
50 27  8 13
12 42  9  3
19 47 30 35
```

The salesmen are the rows, the different products are the columns:

	product 1	product 2	product 3	product 4
salesman 1	5	34	22	36
salesman 2	46	40	18	10
salesman 3	39	23	4	41
salesman 4	50	27	8	13
salesman 5	12	42	9	3
salesman 6	19	47	30	35

Now suppose that this table relates to one sales region and that there are three such regions altogether. The following statement will create a three-dimensional array which represents this situation:

```
      +SALES ← 3 6 4⍴72?100
```

On your screen are (or should be) three blocks of numbers, each of six rows and four columns. These are the three planes, so to speak, of `SALES`. To create `SALES` you specified three dimensions as the left-hand argument of ⍴ (see above). To select a particular element from `SALES`, you also have to give three numbers:

```
      SALES[2;5;4]
20
```

You specified that from `SALES` you wanted plane 2, row 5, column 4. In other words you wanted to know the quantity of product 4 sold by salesman 5 in area 2. You've now seen what is meant by three-dimensional data, and are aware that APL treats data as having dimensions. But the key to understanding the next function is to remember that a single number or single character has no dimensions.

Enquiring about the size of data

As you've seen, the ⍴ function used with two arguments puts specified data into a specified number of rows and columns. The same function used with one argument allows you to enquire about the size (or 'shape') of existing tables and other variables.

To remind yourself of the size of SALES (the three-dimensional data structure you recently created), type:

```
      ⍴SALES
3 6 4
```

As you see, you've been given the size of each dimension of SALES. Now create a two-dimensional table and ask the size of that:

```
      TABLE ← 5 3 ⍴ 15 ? 20
      ⍴TABLE
5 3
```

You've been given the size of each of the table's two dimensions. The height of the table is five rows, the length is three columns.

Next create a variable containing a list of numbers and ask its size:

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

The list is six numbers long.

Finally put a single number into a variable and ask its size:

```
      NUM ← 234
      ⍴NUM
```

The variable NUM has neither length, height nor any other dimension. It is, as we've already said, the equivalent of a point. APL therefore gives an empty response. By the way, the item enquired about doesn't have to be in a variable. Here we enquire about the size of a directly quoted numeric value:

```
      ⍴12 61 502 1 26 0 11
7
```

And here we ask for the size of a string of characters:

```
      ⍴'SHAMBOLIOSIS'
12
```

Before you read on, use the one-argument form of the ⍴ function to enquire about the size of some variables in your workspace. Remember that you're really asking about the size of each variable's dimensions.

Tables of characters

Characters can be arranged in rows and columns too:

```
      ALF ← 3 5 ⍴  'ABCDE'
      ALF
ABCDE
ABCDE
ABCDE
```

But compare the effect of this next statement with the effect of the last:

```
      NUM ← 3 5 ⍴ 12345
      NUM
12345 12345 12345 12345 12345
12345 12345 12345 12345 12345
12345 12345 12345 12345 12345
```

The fact is that `12345` is **one** number, while `'ABCDE'` is five characters. So each single character in the first table is equivalent to each occurrence of `12345` in the second table. Despite their different appearances, both tables contain 15 elements. Notice, though, that while APL has no scruples about putting spaces between the occurrences of `12345` in the numeric table, it doesn't insert spaces in the alphabetic data. Since space is itself a character, it expects you to put in the spaces you require.

Here are a few examples to give you some experience of the way alphabetic data behaves:

```
      MYNAME ← 'GORSUCH'
      ⍴MYNAME
7
      3 7 ⍴ MYNAME
GORSUCH
GORSUCH
GORSUCH
```

In the last example the seven characters in `'GORSUCH'` were arranged as three rows each of seven columns.

In this example the same characters are arranged in three rows of 14 columns. Since `MYNAME` contains seven characters, this fits quite neatly, though a column of spaces between the present columns seven and eight would be an improvement.

```
      3 14 ⍴ MYNAME
GORSUCHGORSUCH
GORSUCHGORSUCH
GORSUCHGORSUCH
```

In the next example, the characters fill a three-row by eighteen-column table. This is not such a neat fit:

```
      3 18 ⍴ MYNAME
GORSUCHGORSUCHGORS
UCHGORSUCHGORSUCHG
ORSUCHGORSUCHGORSU
```

We'll try again. First we'll put a space at the end of the original character string, making it up to eight characters. Then we'll define a table with sufficient columns for the eight characters to be

printed in their entirety five times on each of three rows.

```
      MYNAME ← 'GORSUCH '
      ⍴MYNAME
8
      3 40⍴ MYNAME
GORSUCH GORSUCH GORSUCH GORSUCH GORSUCH
GORSUCH GORSUCH GORSUCH GORSUCH GORSUCH
GORSUCH GORSUCH GORSUCH GORSUCH GORSUCH
```

In this final example, this is the result we want to achieve:

```
ADAMS
CHATER
PRENDERGAST
LEE
```

See if you can define a table which achieves that result before you look at the solution below.

******

You want a table of four rows. The difficulty is working out the columns. The columns must accommodate the longest name, which is `PRENDERGAST` with 11 characters. However, merely to put the names into four rows of 11 columns won't achieve the desired result:

```
      4 11 ⍴ 'ADAMS CHATER PRENDERGAST LEE'
ADAMS CHATE
R PRENDERGA
ST LEEADAMS
 CHATER PRE
```

The other names must he padded out with spaces so that each name plus following spaces exactly fills 11 columns. (For clarity, each space is represented here by a dot.)

```
      4 11 ⍴ 'ADAMS......CHATER.....PRENDERGASTLEE........'
ADAMS......
CHATER.....
PRENDERGAST
LEE........
```

There's some good news for you if you found that a tedious exercise. APLX has a special facility called `⎕BOX` which arranges data into rows and columns for you without any of that bother. (It's a system function and you can read about it in the APLX Language Manual). Doing the counting yourself on this occasion has, however, given you a chance to see how character data behaves.

Mixed tables

You might remember that, in the last chapter, we made up lists which contained both characters and numbers. The examples that we have used so far in this chapter have been either characters or numbers. We can make up mixed tables in exactly the same way that we made up other tables. Here's one:

```
      MIXTURE ← 3 3⍴'A' 1 'B' 'C' 2 'D' 'E' 3 'F'
      MIXTURE
A 1 B
C 2 D
E 3 F
```

You can't, of course, carry out arithmetical operations on mixed tables, but you can reshape them with ⍴ and select elements just as you did with unmixed tables. Try making up some mixed tables yourself. APLX will try to display the contents of these tables in as neat a fashion as it can - no easy matter when you have mixtures of characters and fractional numbers in the same columns. If you want to investigate these rules, have a look in the APLX Language Manual.

Nested tables

Just to complete the picture, we can make up tables that contain other tables or lists. Again we follow the rules we discussed earlier when making up nested lists. We will use parentheses and quote marks as we did with lists. Here's an example:

```
      NEST ← 2 3⍴(2 2⍴⍳4) (⍳5) 'A NAME' (2 4⍴⍳8) 23 (3 4⍴ 'NAME')
      NEST
 1 2        1 2 3 4 5   A NAME
 3 4

 1 2 3 4            23   NAME
 5 6 7 8                 NAME
                         NAME
```

What is NEST made up of? It's two rows deep and three columns wide. The first entry in the first row is a 2 row 2 column table made up of numbers, then we have a list of 5 numbers and a list of 6 characters. The second row starts with a numeric table of 2 rows 4 columns, then has a single number and ends with a 3 row 4 column table of characters. Just to check, let's see what the shape of NEST is:

```
      ⍴NEST
2 3
```

Depth

In order to cope with the added complication of nested data, either tables or lists, we have to bring in a new function ≡, called **depth**.

Depth gives an idea of the degree of nesting that can be found in a variable. This becomes important when you bear in mind that we could make up another variable where some of the elements are themselves nested variables and so on.

A single number or character has depth 0

```
      ≡45
0
```

and a list has depth 1:

```
      ≡1 2 3
1
```

So does a table:

```
      ≡2 2⍴3 4 5 6
1
```

Lists and tables which are made up entirely of single numbers or characters will all have depth 1. When at least one element of a list or table already has a depth of 1 (when it is itself a list or a table), then the overall depth of the variable of 2. So our sample variable has a depth of 2:

```
      ≡NEST
2
```

This idea extends when we make more complicated examples. If one element is of depth 2, then the overall depth of the object is 3. The depth of a variable is always set by the deepest amount of nesting found within it.

Try this:

```
      BIG_NEST ← NEST NEST
      ⍴BIG_NEST
2
      ≡BIG_NEST
3
```

`BIG_NEST` is made up of variables that already have a depth of 2, so it has depth 3. In fact, it's made up of the two objects `NEST` forming a two element vector.

Summary

The functions introduced in this session were `?`, `⍳` and `⍴`. (See APLX Language Manual for definitions.)

Some points worth remembering are:

1. Tables are specified and filled in row order.

2. Tables involved in an arithmetic operation must have the same number of rows and columns.

3. Catenate (,) joins tables with equal numbers of rows.

4. Data has dimensions:

   - a single number or character has no dimensions

   - a list has one dimension, length

- a table has two dimensions, height and length

- data in APLX can have up to sixty-three dimensions.

5. The result returned by the one-argument form of ⍴ is the size of each dimension of the data you enquired about (e.g. how long it is, or how deep and high).

6. In character tables, every character, including space, is one column.

7. Tables can be made up of a mixture of numbers and characters.

8. Tables can be made up of lists and other tables.

9. Nested tables have depth.

Practice

The strength of APL is that almost any logical combination of functions is possible. This means (for example) that the result of an enquiry about a variable's size can be passed along the line to form the argument to the next function:

```
      (⍴'ABC','DEF') + ⍴'GHI'
9
```

Or to take another example, if you've defined a table like this:

```
      TABLE ← 10 10 ⍴100 ? 100
```

and you now want to select the first nine numbers in row 1, there is an alternative to typing laboriously:

```
      TABLE[1;1 2 3 4 5 6 7 8 9]
```

You can instead let the ⍳ function generate the index for you:

```
      TABLE[1;⍳9]
```

These examples are not particularly significant in themselves. They merely indicate the variety of possibilities that exists if you care to experiment. When you've finished your experimentation, try the problems provided.

Problems

Q1. Set up a four-row one-column table called `MILES` containing:

```
      300 42 25 140
```

And a similarly shaped table called `RATES` containing:

```
      27.5 15 27.5 27.5
```

Multiply `RATES` by `MILES`, then multiply the result by 0.01 to produce a table called `EXPENSES`.

Q2. Change the number in column 1 row 3 of `MILES` from 25 to 250. Again, multiply `RATES` by `MILES` and the result by 0.01 to give `EXPENSES`, then reformat `EXPENSES` to produce a one-row four-column table.

Q3. Define `X` as a three-row ten-column table containing random numbers, and `Y` as a three-row four-column table also containing random numbers. Add `X` to `Y`, first taking whatever steps you think necessary to enable the operation to take place.

Q4. Using table `X` created in problem 4, add the first and second rows and replace the third row with the result of the addition.

Q5. Create a table which will look like this when displayed:

```
M
I
C
R
O
A
P
L
```

Q6. What will be the result of each of these ⍴ statements? Predict each result before you press ENTER.

a)
```
⍴ 'ABC DEF'
```

b)
```
⍴ 480 0 1.2
```

c)
```
TABLE ← 10 10 ⍴ 100 ⍴ 1000

 ⍴TABLE
```

d)
```
⍴ 'R'
```

e)
```
⍴ '480 0 1.2
```

f)
```
TABLE ← 2 10 3 ⍴ 100 ⍴ 100

⍴TABLE
```

Answers

Q1.

```
      MILES ← 4 1 ⍴ 300 42 25 140
      RATES ← 4 1 ⍴ 27.5 15 27.5 27.5
      +EXPENSES ← 0.01×RATES×MILES
82.5
 6.3
 6.875
38.5
```

Q2.

```
      MILES[3;1] ← 250
      +EXPENSES ← 1 4 ⍴ EXPENSES ← 0.01×RATES×MILES
82.5 6.3 68.75 38.5
```

Q3.

```
      X ← 3 10 ⍴ 30 ? 100
      Y ← 3 4 ⍴ 12 ? 100
      X+( 3 6 ⍴ 0) ,Y
```

Since the problem didn't say which columns of `X Y` were to be added to, you may have put the zeroes on the other side:

```
      X+Y,3 6 ⍴ 0
```

Q4.

```
      X[3;] ← X[1;]+X[2;]
```

Q5.

```
      8 1 ⍴ 'MICROAPL'
```

Q6. You saw the answers to this problem when you entered the statements.

Writing a Function

The / operator

You may already have looked this up in the APLX Language Manual as suggested at the end of session 1. But since it's used in one of the functions you're going to write, we'll cover it briefly just in case. / is called **Slash** or **Reduce**.

Note that in the context in which we're describing it here, / is an operator not a function; it modifies or extends the operation of the functions it's used with. Here's an example of its use:

```
      +/ 1 6 3 4
14
```

What / did was to extend the operation of + so that the expression effectively became:

```
1+6+3+4
```

Here's an example using multiply:

```
      ×/ 1 2 3 4
24
```

Multiply was used on each number in the list like this:

```
1×2×3×4
```

Let's see how this works on a table:

```
      TABLE ← 3 3 ⍴⍳9
      TABLE
1 2 3
4 5 6
7 8 9
      +/ TABLE
6 15 24
```

It won't take you a moment to work out what APL has done to produce the three numbers 6 15 24.

Obviously they're the sum of the numbers in each row. (If you want to make the +/ operation apply to columns instead of rows, this is easily done. It's explained under the entry for [], **Axes**, in the APLX Language Manual.)

Remembering APL's rule of working from right to left, and the fact that one function's results are data to the next function down the line, enter an expression that will sum *all* the numbers in the table.

******

This is one solution:

```
      +/ +/ TABLE
45
```

We know that the result of the first (ie the right-hand) part of the statement is `6 15 24`. So these three numbers are the data for the left-hand `+/`. Their sum is `45`.

You can add the numbers in just one column by the usual method of indexing. Here's `TABLE` again followed by a statement which adds the numbers in the first column:

```
      TABLE
1 2 3
4 5 6
7 8 9
      +/ TABLE [;1]
12
```

Here's a useful combination: the function `⌈` used with `/` selects the largest number from a list:

```
      ⌈/75 72 78 90 69 77 81 88
90
```

While the equivalent `⌊` statement naturally produces the smallest number:

```
      ⌊/ 75 72 78 90 69 77 81 88
69
```

In case it crossed your mind that `⌈` and `⌊` were being used in their one-argument forms in the last two examples, remember that what `/` does is to put the function (`⌈`,`⌊` or whatever) between each element of the data it's applied to like this:

```
      75 ⌈ 72 ⌈ 78 ⌈ 90 ⌈ 69 ⌈ 77 ⌈ 81 ⌈ 88
```

Here's a final example of the use of `/` for you to ponder before you go on to the topic of function definition. What does this example do?

```
      X ← 1 2 3 4 5
      (+/X)÷⍴X
3
```

User Functions

Hitherto your APL statements have worked their way up the screen and, when they reached the top, have disappeared without trace. You're about to learn how to preserve statements, or groups of statements, so that you can summon them back for execution or amendment at any time. In other words you're going to find out how to write a function. A user-defined function in APL is like a program in another language. It has a name and consists of APL statements. When you quote the name, the statements are executed consecutively, or in whatever order you specified when you defined the function.

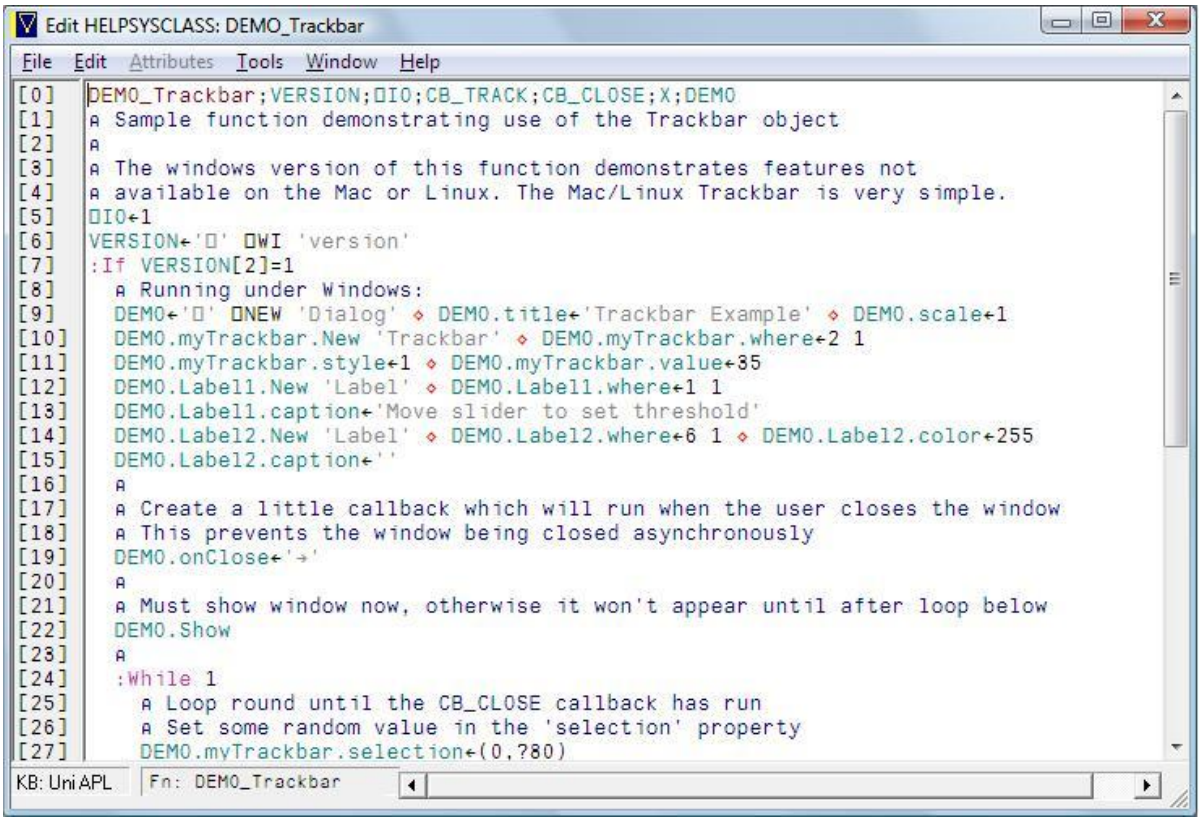
Writing a Function

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

The most commonly used way is to use a **full-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
```

Here is a snapshot of an editor window on a Windows system, showing a function called `DEMO_Taskbar` being edited:



For backward compatibility with old APL systems, APLX also supports a primitive line-at-a-time editor called the **Del 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
```

You will probably never need to learn the Del editor. If you do accidentally type a ∇ character to enter definition mode, just type another ∇ to get back to calculator mode.

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. Listing functions in this way makes it clear at a glance that you are looking at a function. It's also a convention commonly used in other APL documentation on the Internet.

If you are using the normal full-screen editor, you **do not type** the ∇ characters or the line numbers.

******

APL requires to know the name of the function you're defining. Your first function will be called `TRY1` so enter:

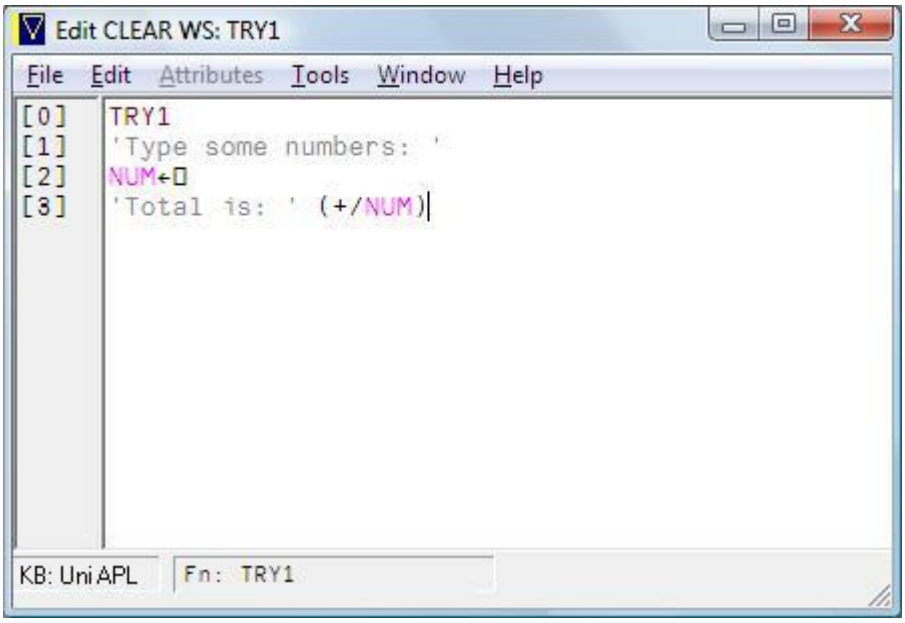
```
      )EDIT TRY1
```

Enter the following function (Remember that you don't type the ∇ or the line numbers):

```
      ∇TRY1
[1]   'Type some numbers: '
[2]   NUM ← ⎕
[3]   'Total is: '(+/NUM)
      ∇
```

(The ⎕ symbol is called Quad and is found on the 'L' on the keyboard. What the ⎕ does will be quite clear in a couple of minutes)

Here's the complete function shown in an APLX editor window. Don't worry about the colours for now; APLX uses syntax colouring to show different types of symbol.



To save the function, select 'Save (fix) in workspace' from the editor window's File menu and then close the window.

Running a Function

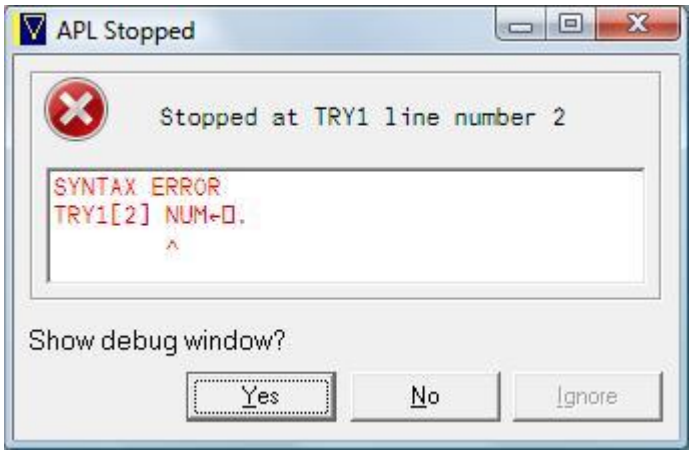
Now you're ready to run the function you've written. Type the function name to run it:

```
      TRY1
Type some numbers:
⍞:
```

Type in a few numbers as the message suggests:

```
      93 7 0 2
Total is: 102
```

If you get errors while running a function (e.g because you made a typing mistake), APLX will normally ask you whether to show the Debug window:



For now, just answer 'No' and then use `)EDIT TRY1` to get back to the edit window, where you can correct your mistake. Alternatively if you're ready to explore further, answer 'Yes' and then correct the mistake directly in the Debug window.

Now you've seen what the function does, we'll go very briefly through the APL statements that make it up.

- Line 1 was entered as:

  ```
  'Type some numbers: '
  ```

  It behaves exactly as you'd expect. When the function is executed, the text in quotes is displayed on the screen.

- Line 2 consisted of:

  ```
  NUM ← ⍞
  ```

  Quad (⍞), as used here, tells APL to wait for something to be typed in on the keyboard. So this line means 'Accept some input from the keyboard, then assign that input to the variable `NUM`'. Line 2 was responsible for the invitation to type (ie the Quad symbol and colon) which appeared after the line of text when the function was executed.

- Line 3 was as follows:

```
'Total is: '(+/NUM)
```

  Working from the right, it first adds the numbers in NUM,that is, the numbers which were typed in, then it displays the result preceded by the text "Total is: ". We have made up a nested list (remember the rules about parentheses).

Editing a function

Now suppose you want to add a line to this function, telling the user how many numbers he or she typed in. To modify the function, just open a new edit window:

```
      )EDIT TRY1
```

You want to insert a step to display a message saying how many numbers were entered. This step should go between the present lines 2 and 3:

```
'You have entered' (⍴NUM) 'numbers'
```

Run your amended function and check that it works:

```
      TRY1
Type some numbers:
⍞:
      93 7 0 2
You have entered  4  numbers
Total is: 102
```

How about inserting yet another step? You no doubt recognised that the statement you were left to ponder earlier in this chapter worked out the average of a group of numbers.

Why not insert a step at the end of TRY1 which displays the average of the numbers in NUM? Rejoin the manual when you've done that.

******

Line 5 of TRY1 should now look like this, though your text in quotes may be different:

```
'Average is:' ((+/NUM)÷⍴NUM)
```

See if it runs then read on regardless of the result!

Editing Practice

Now make use of the function editor to add a couple of lines to TRY1. You want the function to print out the biggest and smallest of the numbers entered, together with some suitable text.

******

Hopefully all went well, and you ended up with a function looking something like this:

```
      ∇TRY1
[1]   'Type some numbers: '
[2]   NUM ← ⍞
[3]   'You have entered' (⍴NUM) 'numbers'
[4]   'The biggest was: ' (⌈/NUM)
[5]   'The smallest was: ' (⌊/NUM)
[6]   'Total is: '(+/NUM)
[7]   'Average is:' ((+/NUM)÷⍴NUM)
      ∇
```

Saving a workspace

So far you have saved the function `TRY1` in the workspace, but the workspace itself has not been saved to disk. If you were to end your APL session now all your work would be lost.

You may have doubts about saving the workspace containing `TRY1`. But do it anyway as a dry run for when you have something you really want to save.

First check what variables you have in your workspace:

```
      )VARS
NUM TABLE X   (your list may be different.)
```

There's a system command for enquiring about functions which you can now use:

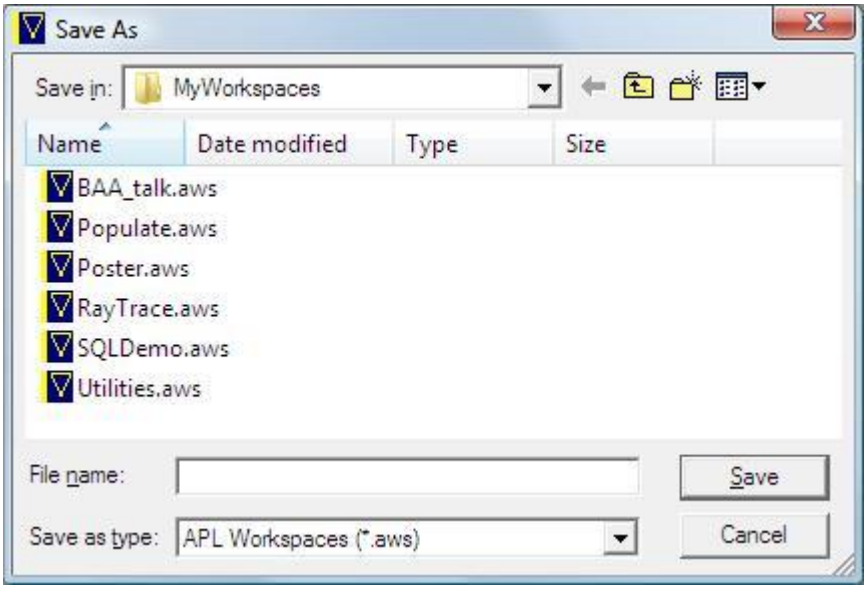
```
      )FNS
TRY1
```

You're going to save the workspace, so you may as well first erase `TABLE`, `X` and any other variables you have in your workspace which aren't needed by `TRY1`:

```
      )ERASE TABLE X
```

`)ERASE` is another system command. It lets you erase one or several variables or functions from the workspace in memory. You can check that the erasures took place by issuing another `)VARS` if you want.

The next step is to save your workspace. For GUI versions of APLX, the easiest way to save a workspace to disk is to use the "Save)SAVE" command in the File menu. If the workspace has not yet been saved, this will display a standard file selection dialog box asking you where you want to save it.



On subsequent occasions, the copy of the workspace on disk will be updated each time you select Save again.

Now you know that workspace is saved on disc, you can clear the workspace in memory using the familiar command:

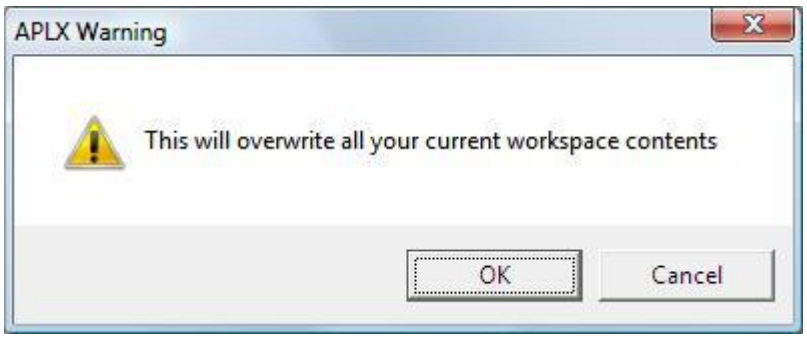
```
      )CLEAR
CLEAR WS
```

You can check that it's worked by typing:

```
      )FNS
```

The function `TRY1` is no longer there.

You can get the workspace back into memory by selecting the "Open...)LOAD" item in the File menu. When you choose this menu item, a second dialog box usually appears to warn you that the operation you are about to perform will overwrite your current active workspace (you can disable this feature using the APLX Preferences dialog). This will only happen if you are not in a clear workspace.



You will then be asked which workspace you want to load, using the standard file selection dialog box.

The workspace you select will be read into memory. Check that `TRY1` is back by issuing a `)FNS` command.

User functions with arguments

If you compare `TRY1` with the functions built-in to APL, one point should strike you: `TRY1` has no arguments.

User functions can, in fact, have no arguments, one argument or two arguments.

Let's take the expression used in `TRY1` to find the average of the numbers and define it as a one-argument function called `AV`. The intention is that once `AV` exists, we should be able to use it like a built-in APL function. We should, for example, be able to type:

```
      AV 12 7 3 1
```

and get a result which is the average of the numbers on the right.

Here's the first line of `AV`, known as the function header statement. When defining the function header, we have to indicate that the function will take an argument to its right:

```
∇AV X
```

We've used `X` to represent the right-hand argument, that is, the data `AV` will work on.

Here's the complete function:

```
     ∇AV X
[1]   (+/X)÷⍴X
```

We've assumed the numbers to be averaged are in the variable called `X`. That's all there is to `AV`, so create the function using `)EDIT AV` and try it out:

```
      AV 3 8 1 4
4
      AV 192 4534 12 0 2
948
```

It will work equally well if the right-hand argument is a variable:

```
      NUM ← 1 2 3 4 5
      AV NUM
3
```

As you can see, any value to the right of `AV`, whether a directly-quoted number, or a value in a variable, is substituted for the `X` we used in the function definition.

User functions with two arguments work very similarly. In the function header, the arguments are represented by variable names on either side of the function name. For example, for a function called `MPH` intended to work on two arguments, namely distance and time, the header might be:

```
∇D MPH T
```

The function `AV` printed its result directly to the screen. If you want the function to return a result which can be used in further APL expressions, you need to change `AV` as follows:

```
     ∇R←AV X
[1]   R←(+/X)÷⍴X
```

Here, the average is assigned to a result `R`, allowing you to do things like:

```
      ¯3 + AV 3 8 1 4
1
```

But you'll have to find out about this, and other types of user-defined functions, on your own. We will return to the subject in a later chapter.

Functions within functions

There's no reason why `AV` should not be used in `TRY1` to work out the average of the numbers input and held in `NUM`

Try editing `TRY1`, then check that the new arrangement works.

Overview of the APL System

ISO specification

APLX incorporates the International Standards Organisation (ISO) draft specification of APL. It conforms to that specification in all important respects. APLX also conforms closely to the specification for APL2 as detailed in the IBM Manual 'APL2 Programming Language Reference' (SH20-9227-3).

Whilst retaining compatibility with APL2, APLX Version 4 adds object-oriented language extensions, as well as a large number of other facilities, to the language.

The APLX Interpreter

Statements in a computer language have to be converted into the code used by the computer before they can be executed. APL is an interpreted language which means this process is done line-by-line at the time of execution. The program which does the conversion is the APL interpreter.

The workspace

This is an area in the computer's random access memory where the programs and data on which you're working reside. Other programs and data can be brought in from disc, but only items in the workspace are available for calculation or processing.

Data

You can type data in at the keyboard, or store it on a disc and load it into the workspace when required. It can consist of numbers or characters or both. (A character is any letter, numeral or symbol on the keyboard.)

Data is held in structures called arrays. An array can be:

- A single data item (called a scalar)
- A list of items, i.e. a one dimensional array, (called a vector)
- A table of data items i.e. a two dimensional array, (called a matrix)
- A higher dimensional array of data items

Each data item in an array can be either:

- A number
- A character
- Another array

Arrays containing both characters and numbers are called **mixed** and arrays which have elements

which are themselves arrays are called **nested arrays**.

There is also some special cases:

- A collection of data items and/or functions (called an **Overlay**)
- An object reference or class reference arising from the object-oriented language extensions in APLX

You can use numbers and letters directly in APL, or you can give names to them and use the names instead. An array to which you give a name is a variable.

Note:

The earlier section avoided the use of the formal names for APL data and variables, but these names will be used from now on.

Modes

There are three modes in APL. In **calculator mode** each APL statement is executed as you enter it. In **definition mode**, the APL statements you enter are not executed immediately, but are stored as a user-defined function or operator, the equivalent of a program. When you run a user-defined function or operator, you're in **function execution mode**.

Built-in functions and operators

These are the operations that are built into the APL language. There are about fifty functions, each invoked by a single symbol (+ × ⍴ ⍳ ⌸).

Functions are said to operate on *arguments*. Here the add function has a left and a right argument:

```
129 + 34
```

Here the ⍴ function has one argument, a vector of three numbers:

```
⍴ 18 67 2
```

Most of the functions can perform two different (though usually related) operations depending on whether they're used with one or two arguments. This effectively doubles the repertoire of operations available to you.

Five **operators** are also built-in to the language. You can use an operator with a function to modify or extend the way the function works. An operator has one or more **operands** (usually functions) which are applied to one or more arguments. The combination of an operator and its operand or operands is called a **derived function**. Derived functions may in turn be used as operands to operators.

System functions and variables

These are part of the APL system but strictly speaking aren't part of the APL language. They

extend the facilities provided by the original APL language and their implementation tends to be tailored to the computer on which the APL system is running.

For example, you can read and write data from files using `⎕NREAD` and `⎕NWRITE`.

You can use system variables and functions in your programs. Their names always start with `⎕` (Quad) to distinguish them from other function and variable names.

System commands

Again, these are part of the APL system but aren't part of the APL language itself. Most of them are concerned with managing the workspace. For example, you use system commands to clear the workspace or to save a copy of it on a disc - `)CLEAR` `)SAVE`.

System commands are normally typed in at the keyboard and executed directly, though it's possible to include them in programs if you want to. System commands are always preceded by a right facing parenthesis, `)`.

User-defined functions and operators

These are functions or operators you write yourself while in definition mode. They consist of APL statements and have a name. You can use this name in calculator mode, or you can include the name in another function or operator to cause execution while that function or operator is running.

You can write a function or operator to supplement the repertoire supplied by the system. For example, if you had to use the statistical measure *standard deviation* frequently, you might write a standard deviation function, and use it subsequently exactly like a built-in function.

You can also write functions which are the equivalent of programs in other languages.

A function editor enables you to create functions or operators and subsequently make insertions, deletions and amendments. All versions of APLX will have a simple line editor and usually will also have a full-screen editor (see implementation notes for details).

Files

Most programming languages use external files of data if there's too much data to embed in the program itself.

When you use APL, any data you want to process will usually be in the workspace in memory. Occasionally you may have to bring in more data from a workspace held on a disc. But the workspace is so extremely convenient for holding both simple and more complicated data structures, that only with bigger projects will you find it necessary to set up files in the traditional sense.

In multi-user environments, you may wish to use files to share data between users.

When you do need to work with files, APLX has the facilities for handling them.

Error handling

An error in a statement will usually cause an error message to be displayed. There are various messages which identify the most common errors. If an error occurs during execution of a user-defined function, information which will help locate the error is automatically displayed. There are facilities for error trapping, together with various other testing and diagnostic aids.

Syntax

APL doesn't have many rules about the way expressions are written. You can use almost any logical combination of variables and functions together on the same line:

```
RESULT ← ⌈1000,3×4÷22+QTY
```

There are some common sense rules about spacing. These, and the few other syntax rules that exist in APLX, are given in the APLX Language Manual.

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 the 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's either expressed in 'K' or 'M' , where:

- One 'M' represents a Megabyte, approximately 1000 'K'
- One 'K' 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.

For users of GUI versions of APLX (Windows, Macintosh and Linux), the initial workspace size can be changed using the APLX preferences dialog.

You can also change the workspace size for a session by using the `)CLEAR` command with a parameter specifying 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. For example:

```
)CLEAR 50MB
```

The initial size of the workspace also depends on how much random access memory (RAM) you have in your system and the amount of disk space reserved for virtual memory. Some of this is used by the operating system and other tasks, so you may not get a workspace as large as the one you requested.

Do not set the workspace size to a value larger than the amount of RAM installed in your system or performance will be affected.

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, e.g. `)CLEAR`.)

There are also system commands for copying the current workspace to disc, reloading it into memory and doing other similar operations. These are mentioned below under the heading 'External workspace commands'.

The system variables and system functions also supply some useful facilities in this area, and are discussed in this chapter.

Internal workspace commands

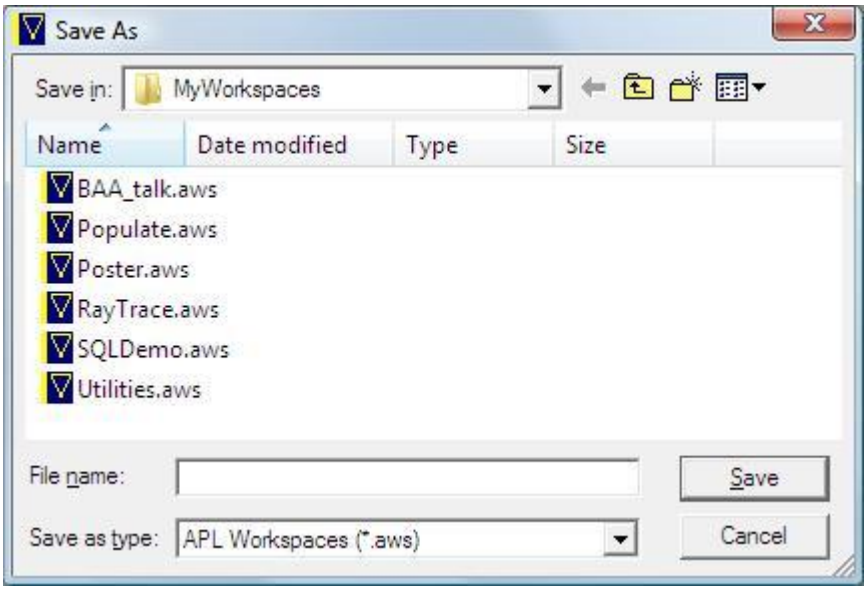
At the start of a session, you're given an empty workspace which has the name `CLEAR WS`. At any time you can return to this state by issuing the system command `)CLEAR`. Any variables or functions you have set up in the workspace are wiped out by this command, so if you want to keep them, you should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

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 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 objects by using the command `)ERASE` followed by the name(s) of the object(s) you want to remove.

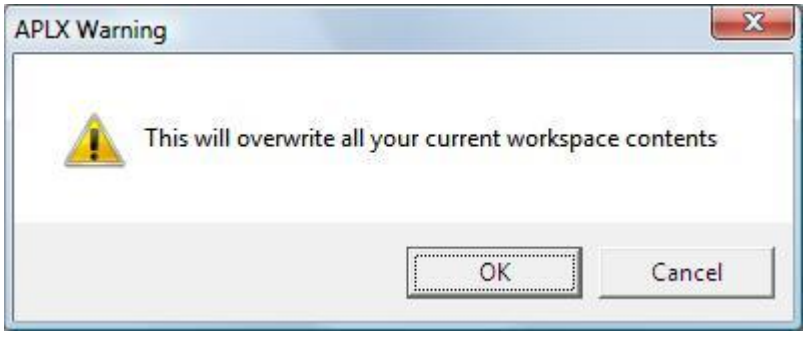
Loading and saving workspaces using the File menu

For GUI versions of APLX, the easiest way to save a workspace to disk is to use the "Save)SAVE" command in the File menu. If the workspace has not yet been saved, this will display a standard file selection dialog box asking you where you want to save it.



On subsequent occasions, the copy of the workspace on disk will be updated each time you select Save again.

To load a new workspace, select the "Open...)LOAD" item in the File menu. When you choose this menu item, a second dialog box usually appears to warn you that the operation you are about to perform will overwrite your current active workspace (you can disable this feature using the APLX Preferences dialog). This will only happen if you are not in a clear workspace.



You will then be asked which workspace you want to load, using the standard file selection dialog box.

The "Save As..." menu item allows you to save your current workspace under a new name. You will be asked to confirm if you try to overwrite an existing workspace.

External workspace commands

Another way to save a workspace is by typing the `)SAVE` system command, e.g.

```
      )SAVE myWorkspace
2008-08-06 12.10.54
```

APL responds by saving the workspace to disk under the specified file name, and displays the date and time at which the save operation happened, known as the timestamp.

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. (There's also a system function, `⎕OV`, which allows you to group functions and variables together as an 'overlay' which can be stored and retrieved as a single entity.)

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

To find out the names of the workspaces already stored on a disc, use the command `)LIB`.

You may be wondering *where* on the disk the workspaces are stored by a `)SAVE` operation? The answer is that APL makes use of something called the **Mount table**. This is a ten row table, in which each row contains the name of a folder on your computer.

For example, if the first row of the mount table on a Windows version of APLX contains 'C:\Users\Fred\Documents', then the command `)SAVE myWorkspace` will create the file 'C:\Users\Fred\Documents\myWorkspace.aws', and the command `)LIB` would list all the APL workspaces in that folder.

A collection of workspaces in a folder is called a library. Each row of the mount table is numbered from 0 to 9, so you can select other rows by specifying a library number, e.g.

```
      )SAVE 3 myWorkspace
2008-08-06 12.22.07
```

This will save the workspace in the folder named by the fourth row of the mount table. The workspace ID includes the library number:

```
      )WSID
3 myWorkspace
```

If a row of the mount table is blank (which is normally the case when you first install APLX), then the user's home folder will be used.

The initial mount table when APLX starts up is set by your preferences. You can also change the mount table at any time using the Preferences dialog, or you can use the APL command `⎕MOUNT`. See the APLX Language Manual for details.

If you specify a full path name when loading or saving a file, the mount table is bypassed and the

file name is used as specified. For example:

```
      )SAVE 'C:\My Workspaces\Utilities\searchWS.aws'
2008-08-06 12.32.06
      )WSID
C:\My Workspaces\Utilities\searchWS.aws
      )LIB 'C:\My Workspaces\Utilities'
searchWS
```

In addition to the libraries 0 - 9, **Library 10** is defined to refer to the folder where the APLX interpreter is located. It contains a number of **demo workspaces** which ship with the APL interpreter, e.g.

```
      )LIB 10
CONVERTQWI  DISPLAY     HELPDOTNET  HELPDRAW    HELPIMAGE   HELPJAVA
HELPOBJECTS HELPQWI     HELPSYSCLASS HELPTRANSFORM PACKAGEDEMO SAMPLEEXCEL
SAMPLESCHART SAMPLESQWI  SAMPLESSYSCLASS TOOLKIT

      )LOAD 10 SAMPLESCHART
SAVED  2007-12-12 10.47.50
APLX SAMPLESCHART Workspace Version 4.0 November 2007
This workspace contains functions showing various charts
programmed using the Chart and Series objects.
```

We recommend that you explore these demo workspaces as you will find a lot of useful material.

System variables

What goes on in the workspace is conditioned to some extent by the current settings of system variables.

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 change 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
```

Other system variables you may occasionally want to enquire about or (in some cases) alter are (the full list is given in the APLX Language Manual):

- ⎕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 normal setting is 10.
- ⎕PW Print width: the number of characters to the line. On most systems, the normal setting is 80.

- ⎕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.

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 change the mount table as described above.

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)CLASSES)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.)

System functions and variables are a mixed bag and have other purposes besides control of the workspace. They will be mentioned again under various other headings.

When you finish reading about Data in the next section, you should take a quick look at the system functions and variables in the APLX Language Manual. Some of them have somewhat specialised applications, but many of them help with everyday tasks and are well worth knowing about.

Data

APL owes a considerable amount of its power and conciseness to the way it handles data. Many lines of code in non-APL programs are devoted to 'dimensioning' the data the program will use and to setting up loops and counts to control data structure. With APL you can create variables dynamically as you need them and the structure you give a data item when you create it determines how it will be treated when it's processed.

Data is an important subject in APL. The rest of this chapter is a survey of its main characteristics.

Variables

As in most programming languages, data can be directly quoted in a statement, for example:

```
234.98 × 3409÷12.4
```

or it can be 'assigned' to a name by the symbol ←, in which case it's called a variable:

```
VAR ← 183.6
```

We concentrate on variables in this chapter, but the comments on data type, size and shape are equally applicable to directly quoted numbers and characters.

Names

Variables, user-defined functions and user-defined operators have names which are composed of letters and digits. The full rules are in the APLX Language Manual, but here are some examples:

```
PRICE
A
albert
A999
ITEM∆1
THIS_ONE
That¯One
```

APL uses upper-case and lower-case characters. APL regards the symbol ∆ (Delta) as a character that can be used in names, and also allows ⍙ (Delta-underbar). In addition the _ (underline) character and the ¯ (high-minus) character may be used, but not as the first character in a name. Names may be up to 30 characters long and must start with an alphabetic character or delta or delta-underbar.

Note:

On some non-GUI implementations of APLX, the lower-case characters may be replaced by underlined upper-case characters. Please check in the Supplement which covers your implementation of APLX.

Types of data

Data can be numbers, characters or a mixture of the two. Characters are enclosed in single quotes and include any letter, number or symbol you can type on the keyboard, plus other, non-printing characters. The space counts as a character:

- this item is 7 characters long: `'1 ABC. '`
- this is a single number: `84724.869`
- this is a number and a character: `12.3 'E'`

Numeric digits, if enclosed in quotes, have no numeric significance and can't be involved in arithmetic.

- this is a numeric value: `2876`
- this variable is composed of 3 characters: `'749'`

Size, shape and depth

An array in APLX can be anything from a single letter or number to a sixty-three dimensional array. Elements within the item may themselves be arrays. Here are some examples of data items:

- a single number or a single character. formally known as a **Scalar**

  e.g.
  ```
  294
  ```
  or
  ```
  'A'
  ```
- a list of numbers or characters, formally known as a **Vector**

  e.g.
  ```
  23 8 0 12 3
  ```
  or
  ```
  'A B C'
  ```
  or
  ```
  28 3 'A' 'BC'
  ```
- a table of numbers or characters, formally known as a **Matrix**

  e.g.
  ```
   7 45  2 89
  16 15 10 21
   8  0 13 99
  83 19  4 27
  ```

or

```
WILSO    393
ADAMS   7183
CAIRN     87
SAMSO   8467
```

As you'll have gathered, data is considered to have dimensions.

A single number or character scalar (like a point) has no dimensions. A vector has one dimension, length. A matrix has two dimensions, height and length. The word 'array' is a general term applicable to a data structure of any dimension. Arrays of up to sixty-three dimensions are possible in APLX.

An array which contains other arrays is called *nested*. An array which does not is called *simple*.

This is how APL displays a three-dimensional array:

```
23 30 11  8
30 22 23 20
 3 19 27  9

14 23 15  8
 9 11  5 15
27 28  2 28

16 16 10 30
15  8  3 29
 3 16 12  9
```

Each of the three blocks of numbers has two dimensions represented by the rows and columns. The three blocks form three planes which constitute another dimension, depth. You will notice that the array is displayed on the screen in such a way that you can identify the different dimensions. No spaces are left between the rows of each plane. One blank line is left between each plane. A four dimensional array would be displayed with two blank lines between each set of planes.

More complicated arrays, where some of the elements are themselves arrays, will also have a 'depth' which measures the degree of complexity of the structure. Thus a simple scalar has a depth of 0 and a structure whose elements are purely simple scalars (such as the array shown above) has a depth of 1. If any element of an array is itself an array, the array has a depth of 2. The depth will go on increasing with the complexity of the structure. An array which has an element which in turn has a non-scalar element has a depth of 3, and so on.

Setting up data structures

It isn't always necessary to explicitly define the size or shape of data:

```
X ← 23 9 144 12 5 0
```

In the case above, X is a six-element vector, by virtue of the fact that six elements are assigned to it. Vectors which contain both characters and numbers may be set up by enclosing the characters in ' (quote) characters. Here is another six-element vector, this time containing four numbers and two

characters.

```
      X ← 1 2 'A' 'B' 3 4
```

Explicit instructions would be necessary if we wanted the six elements to be rearranged as rows and columns. The two-argument form of the function ⍴ (Rho) is used to give such instructions:

```
      2 3 ⍴ 23 9 144 12 5 0
23 9 144
12 5   0
```

The left argument specifies the number of rows (in this case 2) and the number of columns (in this case 3). The right argument defines the data to be arranged in rows and columns.

Notice that the dimensions are always specified in this order, that is: - columns are the last dimension - rows precede columns and, if there are only two dimensions, are the first dimension. In the case of data with more than two dimensions, the highest dimension comes first. So in the three-dimensional example used earlier, the plane dimension is the first dimension followed by the rows, then the columns. (The ordering of dimensions is an important point and will be discussed again later in this chapter.)

To return to the ⍴ function, if the data in the right argument is insufficient to fill the matrix as specified, APL simply goes back to the beginning of the data and uses it again. If too much data is supplied, APL uses it in the order given and ignores superfluous data.

Arrays of three or more dimensions are set up in a similar way to matrices. The following statement specifies that the data in a variable called `NUMS` is to be arranged in three planes, each consisting of three rows and four columns:

```
      3 3 4⍴NUMS
```

The result would look like the three-dimensional array shown in the previous section.

The ⍴ function can also be used to set up vectors. This statement specifies that the number `9` is to be used to form a six-element vector:

```
      6⍴9
9 9 9 9 9 9
```

Arrays of arrays (or 'nested arrays') may be set up by a combination of these rules. Here we set up another vector, some of whose elements are themselves vectors or matrices. Note the use of parentheses to indicate those elements which are actually arrays.

```
      VAR ← (2 3⍴9) (1 2 3) 'A' 'ABCD' 88 16.1
```

The variable `VAR` is another six-element vector, but its first element is a 2 by 3 matrix, the second a three-element vector, the third a single character, and so on.

Data structure versus data value

A data structure has certain attributes, regardless of the specific data it contains. For example, a vector has one dimension while a single number has no dimensions.

You can take advantage of this fact.

If you intend to use a single number for certain purposes, it may be convenient to set it up as a one-element vector. In this next example `X` is defined as a one-element vector containing the value `22`:

```
      X ← 1 ⍴ 22
```

For contrast, here `22` is assigned to `Y` as a single number:

```
      Y ← 22
```

The difference between `X` and `Y` will be seen if we apply the one-argument form of `⍴` to each of them. (This form of `⍴` tells you the size of each dimension of a data item.)

```
      ⍴X
1
      ⍴Y
empty response
```

Both variables contain the value `22`. But `X` is a vector and has the dimension of length, so the `⍴` enquiry produces the answer `1` indicating that `X` is one-element long. On the other hand, `Y` is a single number with no dimensions. The answer `1` would be inappropriate since it would suggest that it had the dimension of length. So an empty answer is displayed.

The result of the `⍴` enquiry can itself be used as data in an APL statement. It might, for example be the basis of a decision about what to do next. For this reason, it may suit you to define a value sometimes as a one-element vector and sometimes as a single number.

Similarly, it may be convenient in certain situations to define a vector as a one-row matrix. Here `Z` is defined as a matrix of one row and five columns:

```
      Z ← 1 5 ⍴ 12 5 38 3 6
```

It looks like a vector when displayed:

```
      Z
12 5 38 3 6
```

But an enquiry about its size returns information about both its dimensions:

```
      ⍴Z
1 5
```

Empty data structures

Variables which have a structure but no content may also be useful, for example as predefined storage areas to which elements can be added. An 'empty vector' is a variable which has been defined as a vector, but which has no elements. Similarly, an 'empty matrix' has the appropriate structure, but no elements.

There are many ways of creating empty data structures. To take one example, the function ⍳ (**Iota**) produces a vector of the number of numbers in right hand argument. So ⍳0 produces the vector of no numbers, that is, a vector in which there are no elements:

```
      X ← ⍳0
```

X contains no elements, as can be demonstrated by displaying its contents (nothing is displayed):

```
      X
```

But it is a vector (albeit an empty one) and does have the dimension of length. If the one-argument form of ⍴ is used to enquire about the size of its dimensions, the answer 0 is returned:

```
      ⍴X
0
```

This indicates that its length is zero elements. Contrast this with the answer returned if you apply ⍴ to a single number (which has no dimensions):

```
      ⍴ 45
```

An empty answer is displayed since the item has no dimensions.

An empty matrix can be created in the same way as an empty vector. In the following example, an empty matrix is created consisting of 3 rows and no columns:

```
      TAB ← 3 0⍴⍳0
```

Dimension ordering

When a function is applied to an item with more than one dimension, you need to know which dimension the function will operate on. If you apply an add operation to a matrix, for example, will it produce the sums of the rows or the sums of the columns?

```
          COL 1     COL 2     COL 3     COL 4

ROW 1       1    +    2    +    3    +    4    =  10
ROW 2       5    +    6    +    7    +    8    =  26
ROW 3       9    +   10    +   11    +   12    =  42
           ==        ==        ==        ==
           15        18        21        24
```

The rule is that unless you specify otherwise, operations take place on the last dimension.

The 'last' dimension is the one specified last in the size statement:

```
TABLE ← 3 4⍴DATA
```

The `4` above is the last of the two dimensions specified. It represents the number of columns.

An add operation 'on' the **columns** adds each element in column 1 to the **corresponding** element in columns 2, 3 and 4.

```
  COL 1      COL 2      COL 3      COL 4

    1    →     2    →     3    →     4    =  10
    5    →     6    →     7    →     8    =  26
    9    →    10    →    11    →    12    =  42
```

So, as can be seen, an add operation 'on' the **columns** produces the sum of the elements in each **row**.

Similarly, if you were to apply the add operation to the **first** dimension of the matrix, that is to the **rows**, it would add all the items in row 1 to the corresponding items in rows 2 and 3:

```
ROW 1   |    1        2        3        4
        |    ↓        ↓        ↓        ↓
ROW 2   |    5        6        7        8
        |    ↓        ↓        ↓        ↓
ROW 3   |    9       10       11       12
             ↓        ↓        ↓        ↓
            15       18       21       24
```

So an add operation applied to the rows produces the sum of each column.

As already described, by default operations are applied to the last dimension (the columns). If you want to specify a different dimension, you can do so by using the **axis** (`[]`) operator which is discussed in the chapter on operators in this section.

Indexing

To select elements from a vector or matrix a technique called indexing is used. For example, if you have a ten-element vector like this:

```
X ← 1 45 6 3 9 33 6 0 1 22
```

the following expression selects the fourth element and adds it to the tenth element:

```
X[4] + X[10]
```

Note that square brackets are used to enclose the index.

To index a matrix, two numbers are necessary, the row number and the column number:

```
      TABLE
12 34 27
 9 28 14
66  0 31
      TABLE[3;2]
0
```

In the last example the index selected the element in row 3, column 2. Note the semicolon used as a separator between the rows and columns. Note also the order in which dimensions are specified. This corresponds to the order used in the ⍴ statement.

Items can be selected from data with three or more dimensions in exactly the same way:

```
      DATA[2;1;4]
```

selects the item in plane 2, row 1, column 4 of a three-dimensional data structure.

To select an entire row from the matrix above you could type:

```
      TABLE[1;1 2 3]
```

That is, you could specify all three columns in row 1. A shorter way of specifying this is:

```
      TABLE[1;]
```

Similarly, to select a column, say column 2, you would enter:

```
      TABLE[;2]
```

The expression you put in square brackets doesn't have to be a direct reference to the item you want to select. It can be a variable name which contains the number which identifies the item. Or it can be an expression which when evaluated yields the number of the item:

```
      (3 8 4)[1+2]
4
```

The above statement selects item 3. The item selected by the following statement depends on the value of `P` when the statement is obeyed. If `P` contains `2`, say, then the letter `B` is selected:

```
      'ABCDE'[P]
B
```

You can also use indexing to re-arrange elements of a vector or matrix:

```
      'ABCDE'[4 5 1 4]
DEAD
```

Finally note that the data or variables used within an indexing expression may be of a higher dimension than the object being indexed. Thus:

```
      'ABCDE'[2 2⍴4 5 1 4]
DE
AD
```

For more details on this point check the entry for `[]` in the APLX Language Manual. In addition to the `[]` (bracket) symbols, the `⌷` (squad) function can be used for indexing. The left argument to `⌷` indicates the element or elements to be indexed.

```
      2⌷ 'ABCD'
```

selects the second element from `'ABCD'`.

Built-in Functions

APL provides 50 or so built-in functions that do all the conventional operations and some rather sophisticated ones as well. It also provides 5 operators that modify and extend the way the functions work. These functions and operators can be used together very flexibly. Consequently it's possible to do a great deal in APL without ever leaving calculator mode: one APL line can be the equivalent of an entire subroutine in another language.

Arguments

Most APL functions are capable of doing two different but related tasks. The number of 'arguments' the function is given to work on determines which task it will perform. Here's the ⌈ function used first with one argument, then with two:

```
      ⌈12.625
13
      2 ⌈ 8
8
```

In the first (monadic or one-argument) form, the function rounded up the number on its right to the next whole number. In the second (dyadic or two-argument) form it selected the bigger of the two numbers.

Here's another example:

```
      ÷1 2 3 4 5
1 0.5 0.3333333333 0.25 0.2
      100÷1 2 3 4 5
100 50 33.33333333 25 20
```

In the first example the function ÷ was used with one argument. In this form it yields the reciprocal of the number, or numbers, on the right (i.e. the result of dividing `1` by each number). In the second example it was used with two arguments. In this form it does ordinary division. The left argument, `100`, was divided in turn by each number in the right argument.

Execution order

A line of APL may consist of several functions, and arguments. There are therefore two points you must bear in mind:

- Expressions are evaluated from right to left.
- The results of one function become the argument of the next function.

This simple example illustrates both of these points:

```
      50×2-1
50
```

The expression is evaluated from right to left so `2-1` is evaluated first, giving `1`, and `50×1` is

evaluated second, giving a final result of `50`. The result produced by the subtract function became the right-hand argument of the multiply function.

Numbers or text

Some functions work on numbers only. The arithmetic functions are in this category. You'll 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.

Shape and size of data

Some functions can be used only on data of a certain shape. Matrix divide (⌹), for example, demands data structured in an appropriate way. Other functions work on data of any shape or size. Arithmetic, for example, can be done on single numbers, vectors of numbers, matrices of numbers, or on numeric arrays of higher rank (up to sixty-three dimensions).

Any two data items involved in arithmetic and certain other operations must, however conform in size and shape, that is, it must be possible for APL to match each element in the left-hand argument with each equivalent element in the right-hand argument. The arguments in this example don't match and an error results:

```
      29 51 60 27÷3 11
LENGTH ERROR
      29 51 60 27÷3 11
      ^
```

The reasonable exception to this rule is the case where one argument consists of a single element and the other argument consists of a vector, matrix or multidimensional array:

```
      39 5 91×2
78 10 182
```

APL simply applies the single element to each element in the other argument. There's no ambiguity about which element matches which, and no error results.

Groups of functions

In the rest of this chapter we take groups of functions that do related things and discuss briefly what each group does. A few examples are given, partly to illustrate particular functions and partly to 'acclimatise' you to APL.

To find out more about any function, see the APLX Language Manual which provides a definition

and examples for each function.

The groupings chosen are:

- Arithmetic functions
- Algebraic functions
- Comparative functions
- Logical functions
- Manipulative functions
- Sorting and coding functions
- Miscellaneous functions and other symbols
- System functions

Arithmetic functions

All the usual arithmetic operations are provided. What makes APL arithmetic interesting is not the functions themselves, but the fact that they can be applied to simple or complicated data.

A multiplication, for example can take place between two numbers. Alternatively every number in one matrix can be multiplied by the corresponding number in a second matrix producing a third matrix of the products. (The scope of operations is extended further by the APL operators, but they are dealt with in the next chapter.)

Remember that each function is capable of two different operations, depending on whether it's used with one or two arguments. The different operations are identified in the table below.

Function	One-argument form	Two-argument form
+	Identity (i.e. value)	Add
−	Negation	Subtract
×	Sign of	Multiply
÷	Reciprocal	Divide
⌈	Round up to integer	Bigger of two numbers
⌊	Round down to integer	Smaller of two numbers
|	Make positive	Residue (remainder) of division

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

Examples of arithmetic functions

1. A vector of numbers is multiplied by a single number.

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

2. A vector of numbers is divided by a single number:

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

3. 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
```

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
```

5. 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
```

6. The remainder of dividing 17 into 23 is displayed:

```
      17 | 23
6
```

7. 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

These are functions for doing more advanced arithmetic. The ⍟ (**Log**) function, for example, gives you either natural logarithms or logs to a specified base, while ○ (**Circle**) gives you access to sine, cosine, tangent and seven other trigonometric functions. Factorials and combinations can be obtained with ! (**Factorial**) and if you have a simultaneous equation to solve, or want to try a little matrix algebra, ⌹ (**Domino**) is your function.

Some functions have more everyday uses. The ⍳ (**Iota**) function will produce a series of numbers

from 1 to the number you give it.

The ?(**Query**, **Roll** or **Deal**) function, for its part, 'rolls' you a single random number, or 'deals' you a hand of such numbers.

Function	One-argument form	Two-argument form
⍳	Index generator	
?	Random number	Random deal
*	'e' to power	Any number to power
⍟	Log to base 'e'	Log to any base
○	pi times	Sine, cosine, etc
!	1×2×3×..	Combinations
⌹	Matrix inversion	Matrix division

Examples of algebraic functions

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

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

2. 3 random numbers between 1 and 10, with no repetitions.

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

3. The logarithm to the base 2 of 2 4 8.

```
      2 ⍟ 2 4 8
1 2 3
```

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

```
      2 ! 4
6
```

Comparative functions

Comparative functions naturally take two arguments. The arguments are compared and if the condition specified by the function ('less than', 'equal to' or whatever) is true, the result is 1. Otherwise the result is 0.

Comparative functions are useful for finding items of data that meet certain conditions (e.g. employees whose sex equals 'M', or whose age is less than 55). They are also useful for making decisions about sequence in user-defined functions. For example, if a variable is greater than a certain value, you may want to branch to a particular point in the function.

Function	Two-argument form only
<	Less than
≤	Less than or equal
=	Equal
≥	Greater than or equal
>	Greater than
≠	Not equal
≡	Match
∈	Contained in
⍳	Searches for match
⍷	Find

Examples of comparative functions

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

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

2. Are the corresponding characters in two strings equal?

```
      'ABC' = 'CBA'
010
```

3. Is the first number greater than the second?

```
      10 > 5
1
```

4. Is each number in the first vector less than the matching number in the second vector?

```
      3 9 6 < 9 9 9
101
```

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

```
      12 ∈ 6 12 24
1
```

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

```
      'B'  ∈  'ABCDE'
1
```

7. 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
```

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

```
      13 7 9 0⍳9
3
```

9. Are two matrices exact matches?

```
      (2 2⍴⍳4) ≡  (2 2⍴⍳4)
1
```

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

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

Logical functions

These are also comparative functions, but they work with data composed exclusively of `1`s and `0`s. Since the comparative functions you've just looked at produce results composed exclusively of `1`s and `0`s, you'll appreciate that a logical test is often applied to the result of a comparison.

For example, if you've used the comparative function `>` (greater than) to produce a vector of employees over the age of 25, you'll have a vector in which such employees are represented by a `1`. If you've also applied the `=` (equals) function to your employee data to find cases where sex equals `'M'`, you'll have a separate vector in which `1`s represent males. Now if you were to apply the logical `∧` (**and**) function to these two vectors, it would compare the vectors, and produce a `1` each time **both** vectors contained a `1` in the equivalent position. The result would be yet another vector in which employees who were over 25 **and** male were represented by a `1`.

Logical comparisons are also used in user-defined functions to control branching.

Function	One-argument form	Two-argument form
~	Makes 1 into 0 and 0 into 1	
∨		1 if either/both 1 (Or)
∧		1 if both 1 (And)
⍱		1 if both 0 (Nor)
⍲		1 if either/both 0 (Nand)

Examples of logical comparison functions

1. ~ used to reverse 1's and 0's:

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

2. The same data submitted to various logical functions:

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

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

```
      1 0 1 ∧ 0 0 1
0 0 1
```

4. 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 functions

These functions do a variety of useful operations on data.

The one argument form of ⍴ (**Rho**) enables you to find out the size of data. In its two argument form it enables you to specify how data is to be shaped or arranged. (There were many examples of this in the earlier chapter on data.)

≡ (**Depth**) tells you about the depth (or degree of nesting) of an array.

, **Comma**, in its one-argument form, can take a multi-dimensional object like a matrix, and unravel it into a single-dimensional vector. In its two-argument form, it can catenate separate data items to form single larger items, so that two matrices, for example can be amalgamated. The two-argument form of , can also join (or laminate) data items in such a way as to give the data an extra dimension. Two laminated vectors, for example, would form a matrix.

∈ (**Enlist**) turns any array, even nested arrays, into a simple vector.

⌽ (**Rotate**) and ⍉ (**Transpose**) can be used to reverse the ordering of data within arrays.

The ↓ and ↑ (**Drop** and **Take**) functions are useful for selecting data. You might, for example, wish to examine the first few members of a list with ↑.

~ (**Without**) can be used to remove items from a vector.

⊂ (**Enclose**) will turn arrays into nested scalars, useful for making complicated arrays, whilst ⊃ (**Disclose**) will reverse the process or select elements from nested arrays.

Function	One-argument form	Two-argument form
⍴	Returns dimensions	Creates vector or array
≡	Depth of an array	
,	Converts matrix or array to vector	Catenates (joins) data items
∈	Makes into vector	
~		Removes items
⌽	Reverses elements	Rotates elements
⍉	Transposes columns and rows	Transposes according to instructions
↑	Takes first element of an array	Takes from an array
↓		Drops from an array
⊂	Encloses an array	Creates an array of vectors
⊃	Discloses an array	Picks from an array

Examples of manipulative functions

1. An enquiry about the size of a character string:

```
      ⍴ 'ARLINGTON A.J, 22 BOND RD SPE 32E'
33
```

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

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

3. The matrix `'DOZEN'` is ravelled into a vector:

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

4. 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
```

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

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

6. Even numbers are removed from a vector.

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

7. First 3 characters are selected from a vector:

```
      3 ↑'AWFULLY'
AWF
```

8. An array is made into a nested scalar.

```
      ⊂999 34
 999 34
      ⍴⊂999 34
empty
```

The two element numeric vector is 'enclosed' into a single nested scalar, with an empty shape.

Sorting and coding functions

The sorting functions enable you to arrange data in ascending or descending order. You can either use the computer's collating sequence to determine sorting order, or you can specify your own collation sequence.

When looking at the examples of ⍋ and ⍒ (**Grade up** and **Grade down**), bear in mind that they don't produce a sorted version of the data to which they're applied. Instead they produce what is effectively an index which when applied to the original data will sort it into the required order.

The coding functions ⊤ and ⊥ (**encode** and **decode**) enable you to take data represented in one system and convert it to the equivalent value in another system. For example, you could convert numbers from hexadecimal (base 16) to decimal.

Function	One-argument form	Two-argument form
⍋	Ascending sorted indices	Sort with specified collating sequence
⍒	Descending sorted indices	Sort with specified collating sequence
⊤		Convert to a new number system
⊥		Convert back to units

Examples of sorting and coding functions

1. 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
```

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

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

3. 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]
,./;\"
```

4. To convert the hex number 21 to its decimal equivalent:

```
      16 16 ⊥ 2 1
33
```

Miscellaneous functions and other symbols

Many of these involve the input of data from the keyboard, or affect the way data is displayed or printed.

Function	
⍞	Accept numbers from keyboard or display result
⍠	Accept characters from keyboard or display data

⋄	Statement separator
⍕	Format data for display
⍺⍕	Use picture to format data for display
⍝	Comment
⍎	Execute an APL expression
⌷	Index
⍬	Empty numeric vector (Zilde)

Examples of miscellaneous symbols

1. To carry out more than one APL statement on a single line, then to format the data (turn a number into its character equivalent) and make up a single character vector using the catenate function:

```
      'TYPE A NUMBER'  ⋄ NUM ← ⎕
      'YOU TYPED', ⍕NUM
```

2. 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
```

3. 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
```

4. To index the third item from a vector:

```
      3 ⌷ 1 2 3 4 5
3
```

(The miscellaneous symbols are treated in detail in the APLX Language Manual)

System functions

Another very powerful type of built-in function in the System Function. This is a large topic which is introduced in the next chapter.

System functions (Quad functions)

The built-in System functions extend the power of APLX by providing easy-to-use cross-platform tools for achieving many common and more complex tasks.

In general, the same system functions are supported by all versions of APLX, so that code written using them will run under Windows, Macintosh and Linux. However you should note that APL interpreters from other vendors may implement a different set of functions which is often much less rich.

There are too many system functions to cover in detail here, but here are just a few...

System functions for manipulating data

There are a number of system functions that manipulate data, including:

Function	
⎕BOX	Vector to matrix and vice versa
⎕DBR	Delimited blank removal
⎕SS	String search and replace

For reshaping vectors as matrices (and visa versa) a system function, ⎕BOX is available:

```
      ⎕BOX 'COGS WHEELS SPRINGS'
COGS
WHEELS
SPRINGS
```

and for deleting spaces from character data, there's another system variable, ⎕DBR (Delimited Blank Removal):

```
      ⎕DBR '       WIDE     OPEN           SPACES'
WIDE OPEN SPACES
```

Another function is ⎕SS (String Search and Replace). In this example, all occurrences of the characters ARE in a text variable are replaced by ARE NOT:

```
      MANIFESTO ← 'ALL ANIMALS ARE EQUAL'
      ⎕SS (MANIFESTO; 'ARE'; 'ARE NOT')
ALL ANIMALS ARE NOT EQUAL.
```

⎕SS is extremely powerful. It can also be used to perform *regular expression* searches.

There are further details and examples of these three system functions and others in the APLX Language Manual.

System functions for reading and writing files

APLX includes a large number of functions for reading and writing files.

Function	
⎕NCREATE	Create native file
⎕NTIE	Open native file
⎕NREAD	Read data from native file
⎕NWRITE	Write data to native file
⎕NUNTIE	Close native file
⎕NERASE	Erase native file
⎕NERROR	Get last native file error
⎕NLOCK	Lock/Unlock native file
⎕NNAMES	List names of tied native files
⎕NNUMS	List native file tie numbers
⎕NRENAME	Rename native file
⎕NREPLACE	Replace data in native file
⎕NRESIZE	Resize native file
⎕NSIZE	Get size of native file
⎕NTYPE	Set native file type (MacOS)
⎕EXPORT	Export APL array to file in specified format
⎕IMPORT	Import data from file in specified format

Examples of native file functions

1. To read an existing text file into an APL variable:

```
'C:\Documents\MyFile.txt' ⎕NTIE 1
TEXT ← ⎕NREAD 1 4
⎕NUNTIE 1
```

The whole of the specified file is read into the APL variable `TEXT` with characters converted to APL's internal representation.

2. To create a new text file containing the text 'Hello new file':

```
'C:\Documents\MyNewFile.txt' ⎕NCREATE 1
'Hello new file' ⎕NWRITE 1 4
⎕NUNTIE 1
```

3. To write out a table of data as a web page:

```
SalesData ⎕EXPORT 'C:\Documents\SalesData.html' 'html'
```

4. To read a table of spreadsheet data in Comma-Separated Variable (CSV) format:

```
data ← ⎕IMPORT 'C:\Documents\SpreadsheetData.csv' 'csv'
```

Some more useful system functions

Function	
⎕CHART	Draw chart
⎕DISPLAY	Display array structure
⎕DL	Delay execution
⎕FMT	Use specification phrases, qualifiers and decorators to format data for display
⎕HOST	Issue command to host
⎕NEW	Create new object (See the chapter on Classes)
⎕PFKEY	Set up Function keys
⎕SQL	Interface to external database
⎕TIME	Time/Date text
⎕UCS	Convert text to/from Unicode

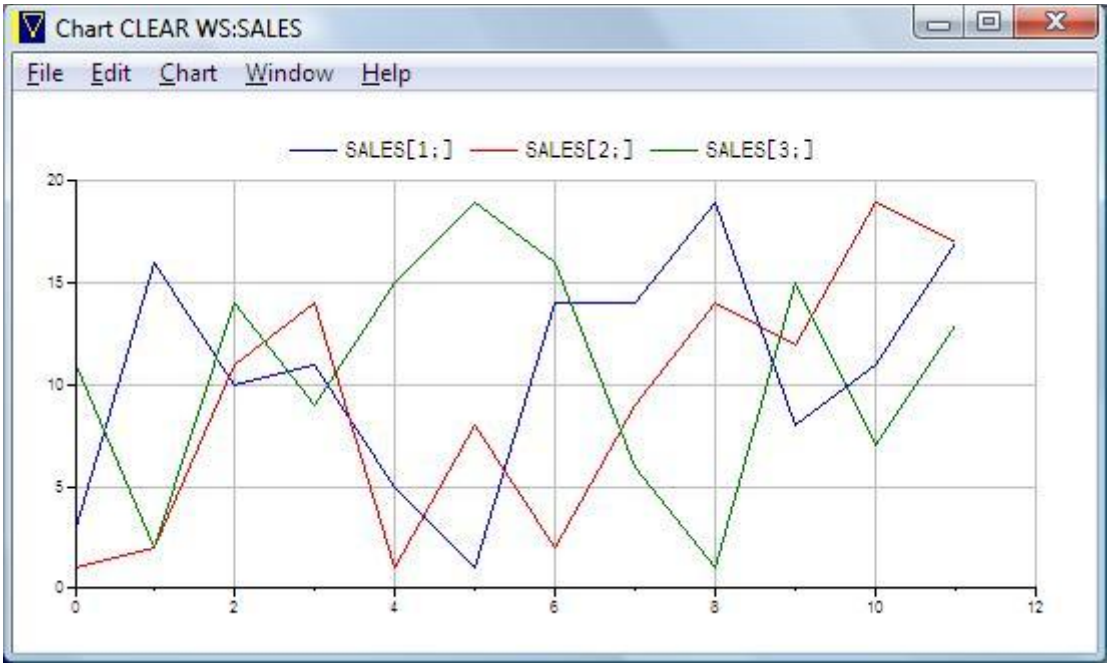
For full details of all system functions and many more examples, see the APLX Language Manual.

Examples

1. To display a chart of sales data:

```
SALES ← ?3 12⍴20
⎕CHART SALES
```

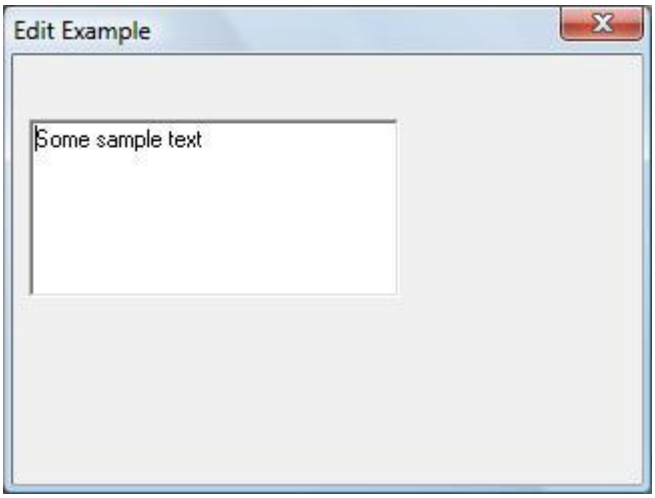
Here we generate some random sales data and display it in chart form in a new window:



2. To create a new window containing an Edit field.

```
DEMO ← '□' □NEW 'Dialog' ⋄ DEMO.title ← 'Edit Example'
DEMO.myEdit.New 'Edit' ⋄ DEMO.myEdit.where ← 2 1
DEMO.myEdit.text ← 'Some sample text'
```

This will create the following window (the appearance will be different if you try this on a Macintosh or Linux version of APLX):



3. To issue a Windows command to list the contents of a folder:

```
      □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
```

5. Here we use SQL to connect to a database and read a table of data into the APL variable `employeeData`, check how many results were returned, and then display the first five:

```
      1 ⎕SQL 'connect' 'aplxodbc' 'DSN=BigCorpDB;PWD=xx;UID=root;'
0 0 0 0
      1 ⎕SQL 'do' 'use bigcorp'
0 0 0 0
      fields ← 'FIRSTNAME,LASTNAME,SEX,EMAIL,EMPLOYEENUMBER,DEPARTMENT'
      (rc errmsg employeeData) ← 1 ⎕SQL 'do' 'select',fields,' from employeedata'
      ⍴employeeData
350 6
      5 6↑employeeData
 Bert     Brown      M Bert.Brown@bigcorp.com      1 C
 Claude   Ptolemy    M Claude.Ptolemy@bigcorp.com  2 B
 Zak      Smith      M Zak.Smith@bigcorp.com       4 E
 Sian     Jones      F Sian.Jones@bigcorp.com      5 C
 Eric     Smallhorse M Eric.Smallhorse@bigcorp.com 6 C
```

Operators

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.

They can be thought of as instructions to built-in and user-defined functions on how to carry out their operations. You can even define your own operators!

The operators available are:

Operator	Name
/	Slash
\	Backslash
.	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)
```

There are more examples of reduction and scan in the APLX Language Manual.

Compress and Expand

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

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 0s. (Note also the use of the **axis** operator - see below).

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 over 400 possible inner products! Obviously 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

When a function operates on data of more than one dimension, there is a choice as to which dimension it should work on.

The default which APL takes if you don't specify otherwise is to operate on the **last** dimension. The order of dimensions is the order in which they are defined in a ⍴ statement, so the last dimension is that of the columns. (This was discussed more fully in the Data chapter under the heading 'Dimension Ordering'.)

If you want to specify a different dimension, you can do so by putting the number of the required dimension in square brackets after the function or operator.

Here's a data structure of two dimensions, called `TAB`, which we'll use in some examples;

```
      TAB ← 2 3 ⍴ ⍳6
      TAB
1 2 3
4 5 6
      +/TAB
6 15
```

Since no dimension was specified, the summing operation requested by `+/` was done on the last dimension, the columns. The elements in column 1 were added to the corresponding elements in columns 2 and 3. This gave the sum of the elements in row 1 and the sum of those in row 2.

This statement specifies that the operation should be done, instead, on the first dimension, the rows;

```
      +/[1]TAB
5 7 9
```

The elements in row 1 were added to the corresponding elements in row 2.

As you would expect, the following statement (which specifies the second dimension, the columns) is equivalent to the first example where no dimension was specified:

```
      +/[2] TAB
6 15
```

Here's an example using the ⌽ function which (when used with one argument as here) reverses the order of the elements in the data specified by the argument:

```
      ⌽TAB
3 2 1
6 5 4
```

Again it has been applied to the last dimension (the columns) by default. What was column 3 is now column 1 and visa-versa.

Here the first dimension is specified:

```
      ⌽[1]TAB
4 5 6
1 2 3
```

Row 1 has changed places with row 2.

Below is a three-dimensional structure such as would be set up by this ⍴ statement:

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

 7  8  9
10 11 12
```

The first dimension consists of two planes (the two blocks of numbers). The second dimension is the rows, two in each plane. The third is the columns, three in each row.

The following statement specifies a multiplication operation on the second dimension. In other words, row 1 is to be multiplied by row 2 in both planes

```
      ×/[2]DATA
 4 10  18
70 88 108
```

The first line is the result of multiplying row 1 by row 2 in the first plane. The second line is the result of the equivalent operation for the second plane.

The first dimension consists of the two planes. A multiplication on the first dimension will multiply each element in plane 1 by the corresponding element in plane 2:

```
      ×/[1]DATA
 7 16 27
40 55 72
```

You'll find other examples of the use of [ ] (**Axis**) in the APLX Language Manual.

Axis Specifications

The list of built-in functions and operators that accept an axis specification is:

Mixed Functions: ↑ ↓ ⊂ ⊃ , ⌽ ⊖

Operators: / ⌿ \ ⍀

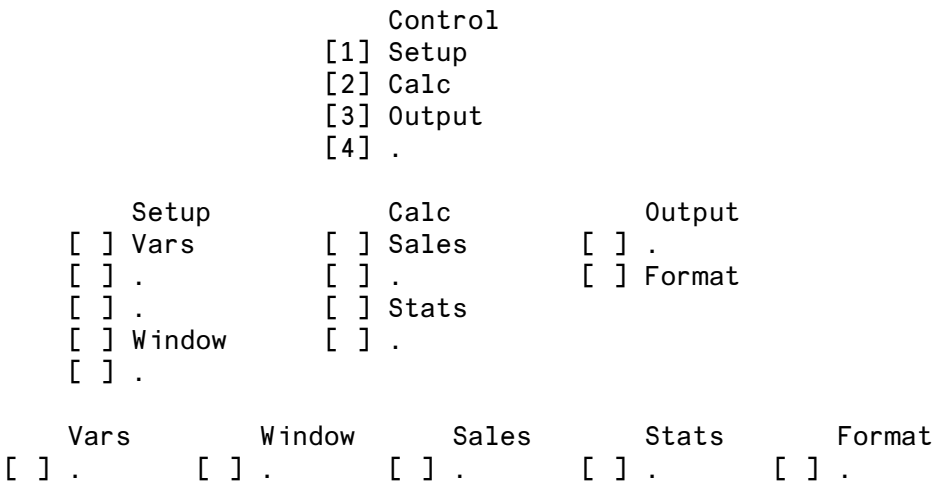
For more details see the APLX Language Manual

User-defined Functions and Operators

A user-defined function can be regarded as equivalent to a program in another language. Like a program, it consists of statements and has a name. When the name is typed in at the keyboard, the statements are executed.

A function can call other functions. Since several functions can exist in the workspace, this makes it possible to adopt a very modular approach to design.

The diagram below shows how a task might be split into functions. The function called `Control` at the top calls each function on the level below to perform a specific sub-task. These functions call other functions in the course of their execution.



Any of the functions could of course be used with other functions to do a different overall task.

`Window` for example, might create a new window and give it a title. Given that the title could be whatever text was currently in a particular variable, such a function might be useful in a number of different applications.

Functions are often only a few lines long, so the structure shown doesn't necessarily represent some vast commercial project. With APL a modular approach comes naturally even for smallish programming tasks.

Arguments and results

User functions need not have arguments. A user function may be a series of APL lines with a name which, when entered, causes the lines to be executed. The name on its own calls the function and no arguments are specified. Such functions are called **niladic** functions.

Alternatively, functions can be defined in such a way that when you call them, you must provide arguments, just as you would with a built-in APL function. Here for example, the built-in function

⌈ is being invoked to round up some numbers:

```
⌈6.5009 12.7 33.33333 909.01
```

The numbers are supplied as the right-hand argument.

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 ⌈:

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

Functions with one argument, like SD are called **monadic** functions. You can equally well define and use functions with two arguments - **dyadic** functions. Indeed, if you want you can write a function which (like a built-in function) can sometimes have one argument and sometimes two, and you can make the action the function takes depend on whether one or two arguments are submitted. (The first line of such a function would normally be a test to determine how many arguments had been submitted.) Another useful option is the ability to return a result from a function.

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.

User-defined operators

User-defined operators are rather more complex, in that they will have one or two **operands**, that is functions that they will apply to data and the function that results from the combination of operator and operands may itself have one or two arguments. Since operators exist to modify the behaviour of functions, a user-defined operator must have at least one operand.

User-defined operators can be treated like user-defined functions for the purposes of editing and entry.

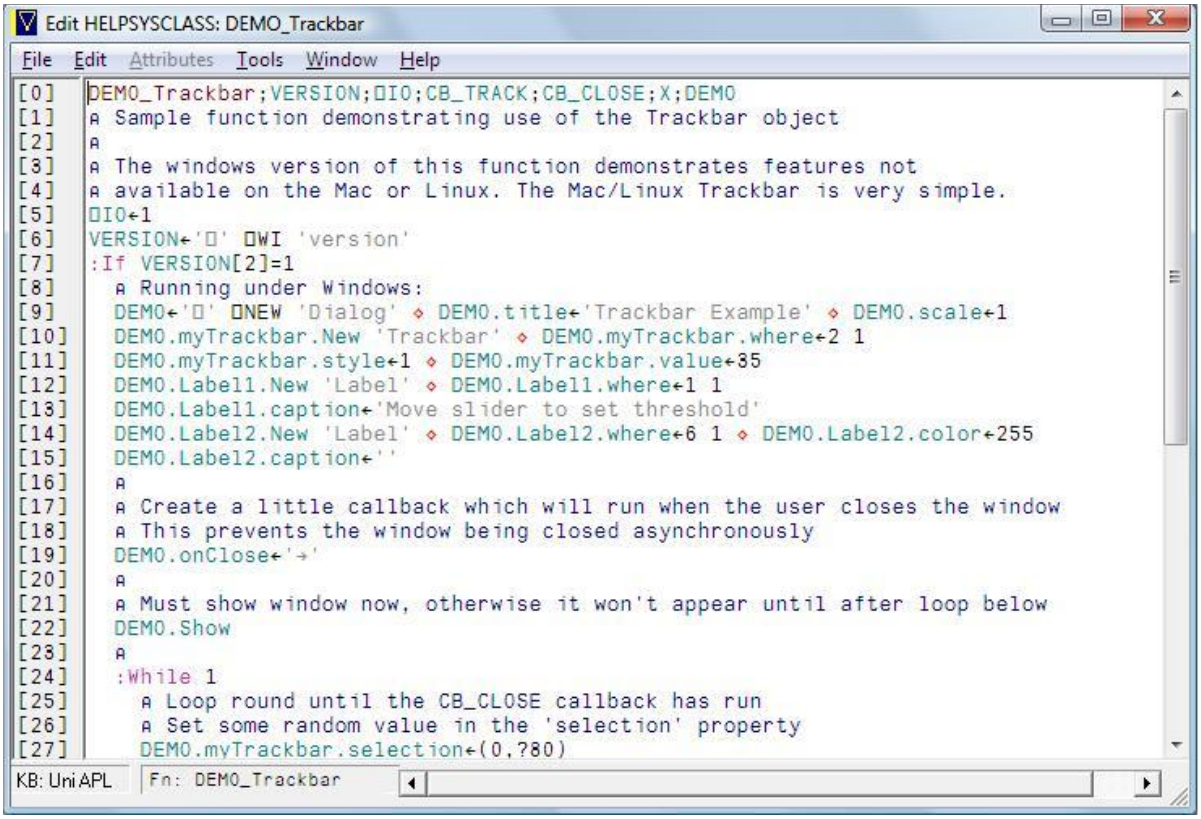
Editing functions

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

The most commonly used way is to use a **full-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
```

Here is a snapshot of an editor window on a Windows system, showing a function called DEMO_Taskbar being edited:



For backward compatibility with old APL systems, APLX also supports a primitive line-at-a-time editor called the **Del 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. Listing functions in this way makes it clear at a glance that you are looking at a function. It's also a convention commonly used in other APL documentation on the Internet.

If you are using the normal full-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
```

(Remember that you don't type the `∇` del character if you are entering the function in an editor window).

When you subsequently use `CALC` you must 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`.

The operator header

The operator header must accommodate operands as well as the arguments of the function derived from it and so the header is more complex. The operator name and its operands are enclosed in parentheses. Thus a monadic operator whose derived functions will take two arguments and return a result, has a header:

```
∇R ← X (LOP OPERATE) Y
```

where `LOP` is the left operand and `X` and `Y` the left and right arguments. A dyadic operator whose derived function will take two arguments and return a result, has a header:

```
∇R ← X (LOP OPERATE ROP) Y
```

Other than its special header line, user-defined operators obey the same internal rules as detailed below for user-defined functions.

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's 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 don't 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 'localise' 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.

Branching

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.

We'll start by introducing the traditional APL branching technique, which is supported by all APL dialects, before considering the modern APLX alternative of using structured-control keywords like `:IF` and `:WHILE`.

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.

You will rarely use → on its own, that is, unconditionally. Consider the following case:

```
[1] ...
[2] →4
[3] ...
[4] ...
```

When this function is run, line 1 is obeyed, then line 2 then line 4. Line 3 is always omitted because the → branches round it. This seems pointless. Similarly, the unconditional → in the following sequence seems to have created a closed loop of instructions that will repeat forever:

```
[1] ...
[2] ...
[3] ...
[4] → 1
```

It's more common to use → conditionally as in the following example:

```
[3] →(MARK<PASS)/7
[4] 'YOU PASSED. CONGRATULATIONS.'
[5] ...
[6] ...
[7] 'BAD LUCK. TRY AGAIN.'
```

The condition `(MARK<PASS)` will generate a 1 or a 0 depending on the values contained in the two variables, `MARK` and `PASS`. If the condition is met, the result is 1. Using the `/` function in its selection role, as was illustrated earlier, the right argument of the → is 7. Thus execution 'goes to' or 'branches to' line 7. On the other hand, if the condition is not met, we *do not* select 7, in other words an empty vector is generated as the right argument to → and execution carries onto the next line.

The statement on line `[3]` could thus be read as:

```
[3] goto 7 if MARK<PASS
```

There are very many different ways of generating branches within an APL function, and these are discussed in more detail in the APLX Language Manual. For now, the expression used in the example above will be used to generate branches in a function.

The last example provides a situation where an unconditional branch may be appropriate. If `MARK` is not less than `PASS`, we proceed with line 4, but it looks unlikely that we would also want to execute line 7, We put a → before line 7 and branch round it:

```
[3] →(MARK<PASS)/7
[4] 'YOU PASSED. CONGRATULATIONS.'
[5] ...
[6] →9
[7] 'BAD LUCK. TRY AGAIN.'
[8] ...
[9] ...
```

Looping

Branching in many programming languages is used to set up loops: sequences of instructions that are obeyed repeatedly till a count reaches a certain value. The count, of course, is incremented each time the loop is executed.

Loops are rarely necessary in APL, since much of the counting that has to be specified in other languages is implicit in the data structures used in APL and is done automatically, For example, the following statement will add the values in `SALES`, whether there are two values only, or a thousand:

```
      +/SALES
```

If a loop is necessary, it can be constructed using a statement similar to the branch statement shown above, the condition test being the value of a loop count. (The entry for → in the APLX Language Manual gives some examples). Alternatively you can use structured control statements like `:WHILE` and `:REPEAT`.

Labels

After an editing session in which you've inserted or deleted lines, the function editor renumbers the function to make sure lines are whole numbers and there are no gaps. So next time you edit or run the function, the line numbers may be different. For this reason it's much safer to 'goto' labels rather than to line numbers.

Here's an earlier example, this time with →s referencing labels rather than line numbers:

```
[3] →(MARK<PASS)/FAIL
[4] 'YOU PASSED. CONGRATULATIONS.'
[5] ...
[6] →NEXT
[7] FAIL: 'BAD LUCK. TRY AGAIN.'
[8] ...
[9] NEXT: ...
```

Labels are names followed by colons, They are treated as local variables and have the value of the line numbers with which they are associated. For example, the label `FAIL` in the extract above will be set up when the function is run and will have the value `7`.

Ending execution of a function

When the last line in a function is executed, the function stops naturally (unless, of course, the last line is a branch back to an earlier line). To end a function before the last line is encountered, you can go to a line number which doesn't exist in the function. The safest line number for this purpose (and the one conventionally used) is 0.

The following statement causes a branch to 0 (in other words, terminates the function) if a variable called `X` currently has a value less than 1.

```
[4] →(X<1)/0
```

Structured control keywords

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

```
[3] :If MARK ≥ PASS
[4]     'YOU PASSED. CONGRATULATIONS.'
[5]     ...
[6] :Else
[7]     'BAD LUCK. TRY AGAIN.'
[8]     ...
[9] :Endif
```

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.

Structured control keywords include:

Function	**Keyword**
Conditional execution	`:If / :ElseIf / :Else / :EndIf`
For loop	`:For / :EndFor`
While loop	`:While / :EndWhile`
Repeat loop	`:Repeat / :EndRepeat`
Case selection	`:Select / :Case / :CaseList / :Else / :EndSelect`
Branch	`:GoTo`
Terminate current function	`:Return` (equivalent to `→0`)

Here is a simple example:

```
     ∇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
      ∇
```

The amount of indentation does not affect the execution of the function, but it does make it easier to read. If you are using the editor window, you can select 'Clean up indentation' from the Edit menu to ident the function appropriately.

See 'Control Structures' in the APLX Language Manual for further details.

Comments in functions

If you want to include comments in a function, simply type them in, preceded by a ⍝ symbol (known as 'lamp')

```
     ∇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's possible to lock a function. A locked function can only be run. You can't 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:

```
  [12] ⍫
```

There may be occasions when you want to make code secure from tampering or scrutiny. But be certain that it's error-free and in its final form - a locked function cannot be unlocked.

Ambivalent or 'nomadic' 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
     ∇
```

Component Files

*This section discusses **component files**, which are used as a very easy way to store and retrieve APL data in a proprietary format. APLX is also able to access other types of files (e.g. text and pictures) via a number of system functions. For example, see:*

- *'APLX Native File Support' for operations on non-APL files.*
- *The ⎕SQL command for interfacing to external databases*
- *The system classes 'picture', 'movie' and 'image' objects for image display and manipulation*
- *The ⎕EXPORT and ⎕IMPORT commands for conversion between a number of file formats - e.g. Comma-separated variable (CSV) files used for spreadsheets.*

Note: APLX supports two different component file systems. The first of these is based on the file-access primitives ⎕FHIST ⎕FRDAC (as implemented in APL. 68000), and is discussed in this section.

The second is based on system functions such as ⎕FTIE and is provided primarily for compatibility with APL interpreters from other vendors. For more information, see the section 'Component File Systems' in the APLX Language Manual.

APL was originally implemented without a filing system because for simple purposes the facilities provided by the workspace are quite sufficient. In general you can keep all the data you need in the current workspace with occasional 'imports' from a saved workspace.

However, occasionally 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 wouldn't want to replicate the functions in each of these workspaces.

You could get round this by having the functions in the active workspace and using)COPY to copy in the data for each month from a stored workspace. (Copying doesn't obliterate what's already in memory, while using)LOAD does). But a more efficient method is to store each month's data in a file, and read that file into the workspace when it's needed.

APLX has a very flexible and simple filing system available for such situations.

Files

APLX files are identified by number rather than by name. Any integer can be used. An APL file can be regarded as a series of numbered pigeon-holes with the useful feature that (subject to space being available) each pigeon hole can hold as much or as little as you like.

Components

The pigeon-holes are called components. A single letter of the alphabet may constitute one component while a matrix containing several thousand numbers may be its next door neighbour.

Functions can be stored in files as well, but they must first be converted into character-matrices by the system function ⎕CR, or stored via the overlay system function, ⎕OV.

Basic file operations

Creating a file is simply a matter of writing a component to it. The command for this is ⎕ (**Quad-Write**). If the file already exists, the component is added to it in the appropriate position. If the file doesn't exist, APL creates it and then writes the component to it:

```
TABLE ⎕ 3 1
```

The above command puts the contents of TABLE into component 1 of file 3, creating file 3 first if necessary.

Here's another example:

```
PRICES ⎕ 3 2
```

The information held in PRICES will become component 2 of file 3, overwriting component 2 if it already exists.

Reading data from a file follows the same principles. To get component 1 from file 3 you would type:

```
⎕ 3 1
```

In the form shown, the command ⎕ (**Quad-Read**) will cause the contents of component 1 to be displayed on the screen.

It's more likely that you'll want to put the data into a variable and carry out some operation on it. The following statement reads component 2 from file 3 and assigns it to a variable called A:

```
A ← ⎕ 3 2
```

Deleting a component or file is specified in much the same way using a command ⎕ (**Quad-Drop**):

```
⎕ 3 2
```

deletes component 2 from file 3, renumbering components in the same way lines in a function are renumbered after deleting a line. Similarly, components can be inserted by writing to a number that lies 'between' two components:

```
PRICES ⎕ 3 1.5
```

The last example, will write a component containing the information in PRICES between

components 1 and 2. The components in the file will then be renumbered in the same way lines in a function are renumbered after inserting a line.

For a fuller explanation of reading and writing component files, see 'Component File Systems' in the APLX Language Manual.

Error Handling

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
      ^
```

As the example shows, the statement containing the error is displayed with an error indicator (^) marking the point at which the APL interpreter thinks the error occurred.

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 recall 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

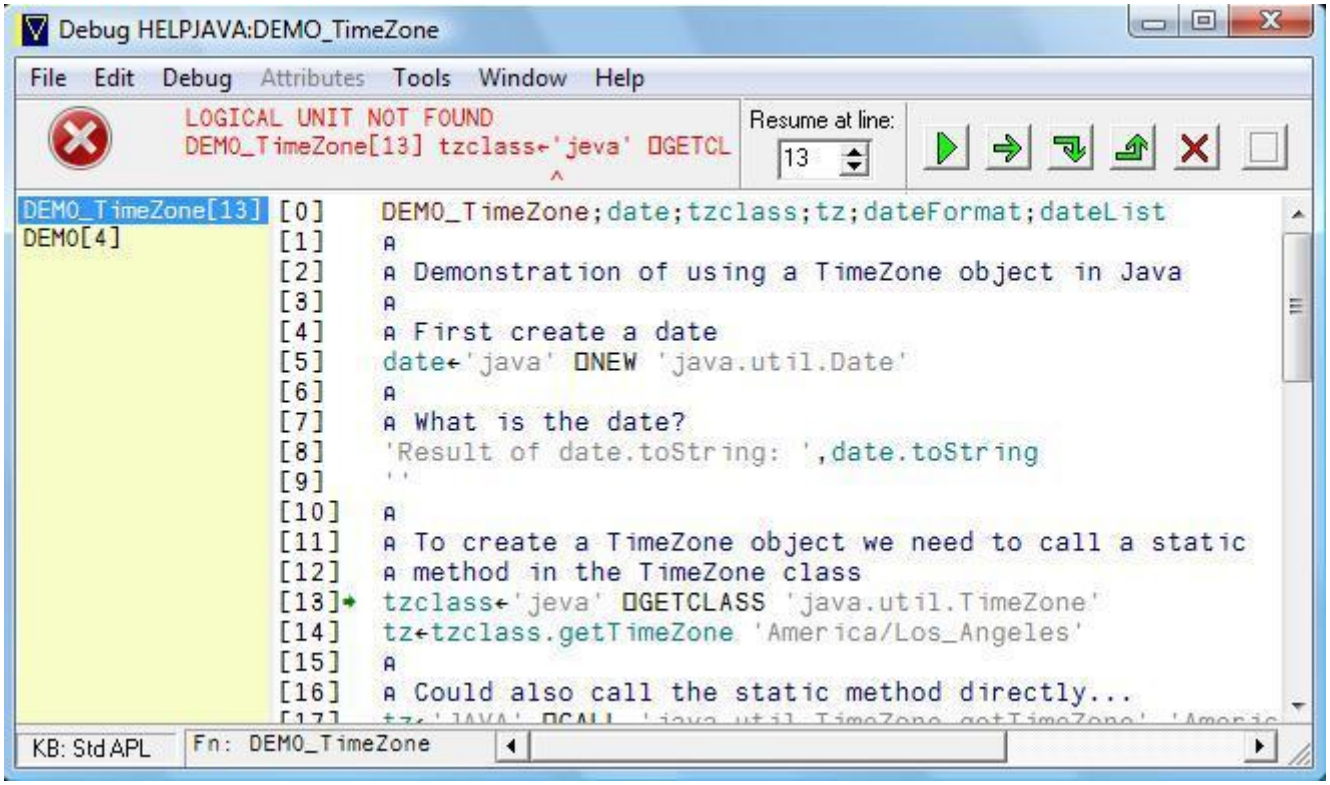
If an error is encountered during execution of a user-defined function or operator, execution stops at that point. The appropriate error message is displayed, followed on a separate line, by the name of the function containing the error, the line number at which execution stopped and the statement itself:

```
LENGTH ERROR
C[2] 1 2 - 1 2 3
     ^
```

The above example shows that execution stopped in function `C` 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:



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 switched off (see the APLX Language Manual).

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**, to see the state of play:

```
      ⎕SI
C[2] *
B[8]
A[5]
```

This display 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 `)SICLEAR`.

Action after suspended execution

Apart from examining the state indicator, what can you do when execution is suspended?

If you want to resume execution at the point where it stopped you can do so 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
→⎕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's 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)SICLEAR to clear the state indicator.

What you cannot do after a halt in execution is to edit any of the pendant 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 `)SICLEAR`.

Error trapping and tracing

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.

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

There are three functions in APLX which both:

- convert the format of data from numbers to characters
- allow you to specify how (converted) numeric data will be laid out.

The functions are ⍕ (**Format** or **Thorn**), ⍺ (**Alpha**) and the system function ⎕FMT. They can be used purely to convert numeric data to characters. The converted data looks the same, but has the properties associated with character data.

Additionally, each function lets you specify how many character positions a number should occupy when it's 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.)

⍺ has the optional extra facility of allowing you to use editing characters to define a 'picture' of how data is to look when 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
```

More details of these functions are given in the APLX Language Manual.

User-Defined Classes in APLX Version 4

Introduction

APLX Version 4 adds object-oriented programming facilities to APLX's core APL2-compatible language. These facilities are broadly similar to those implemented in other object-oriented programming languages (such as C++, C#, Java, Ruby, or R), 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. In addition, the APLX implementation is fully dynamic, that is to say classes and objects can be changed at run-time or when debugging.

Classes can be written in APL, or they can be written in other languages (such as C#), or they can be built-in to the APLX interpreter as System Classes (analogous to the familiar System Functions). In this tutorial, we'll focus on classes written in APL. (If you want to try out the tutorial yourself, you can download a demonstration version of APLX Version 4 from http://www.microapl.co.uk/apl/

The object-oriented extensions to APLX are not part of the International Standards Organisation (ISO) specification of the APL language, although some other APLs have similar facilities.

Jargon

The fundamental building block for object-oriented programming is the **class**. For example, in a commercial invoicing application, a given class might represent the attributes and behaviour of an invoice, and another class might represent a credit note. 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**).

Getting Started

To make this clearer, let's use an example of a class representing a circle. To keep things simple initially, we will give the `Circle` class a single property representing the radius of the circle.

There are a number of ways to create the `Circle` class, but for now just we'll start with a `CLEAR WS` and enter the following APL line, which creates a class and inserts a property called 'radius' into it:

```
      'Circle' ⎕IC 'radius'
```

This has created a new class, which we can see by using the `)CLASSES` system command, which is

analogous to `)FNS` and `)VARS`:

```
      )CLASSES
Circle
```

To create an instance of the `Circle` class, you typically use the

`⎕NEW` system function:

```
      FirstCircle ← ⎕NEW Circle
```

This has created a new instance (object) and assigned a reference to it in the variable `FirstCircle`:

```
      )VARS
FirstCircle
```

If you try to inspect the new object, APL will display it in the following format by default. Note that (unless you change the default display), the object is displayed by showing its class name in square brackets.

```
      FirstCircle
[Circle]
```

Now let's assign a value to the new `Circle` object's radius:

```
      FirstCircle.radius ← 10
      FirstCircle.radius
10
```

Note the **dot-notation** used to specify the object and property being assigned. Except that it's a member of the object `FirstCircle`, the radius property behaves like any other APL variable:

```
      ⍳FirstCircle.radius
1 2 3 4 5 6 7 8 9 10
      FirstCircle.radius=20
0
```

We can also create a vector containing five different circles:

```
      CircleList ← ⎕NEW ¨5⍴Circle
      CircleList
[Circle] [Circle] [Circle] [Circle] [Circle]
```

It is possible to set the radii of all of the circles in the same statement:

```
      CircleList.radius ← 10 50 20 30 20
      CircleList.radius
10 50 20 30 20
      CircleList.radius=20
0 0 1 0 1
```

Notice that APLX lists the radius values for all five circles in a single vector, which can be used in

expressions.

We could also specify all the radii using a single scalar, in which case the scalar is assigned to the radius property in each object using scalar extension:

```
      CircleList.radius ← 20
      CircleList.radius
20 20 20 20 20
```

You cannot access a property if it does not exist in the class definition. For example:

```
      FirstCircle.Colour ← 'Red'
VALUE ERROR
      FirstCircle.Colour ← 'Red'
      ^
```

To add a new property it is necessary to modify the class definition, which we will cover in more detail later.

Now let's add a method called `Area` to our `Circle` class. Again, there are many ways of editing classes in APLX, but for now you can use the following:

```
      )EDIT Circle.Area
```

Use the editor window to create the following method:

```
      ∇R←Area
  [1] R ← (○1)×radius*2
      ∇
```

(You do not enter the `∇` characters or line number when using the editor window).

We can now call our new method using the following notation. (Notice that the new method can be applied to existing objects - try doing that in Java!)

```
      FirstCircle.Area
314.1592654
```

Within the method, the object's radius can be referred to directly as just `radius`, without using dot notation. APL knows that the `Area` method has been called for the object `FirstCircle`, and hence uses the radius value contained in the object.

Again, if you have an array of objects you can apply the same method to each one.

```
      CircleList.radius ← 10 50 20 30 20
      CircleList.Area
314.1592654 7853.981634 1256.637061 2827.433388 1256.637061
```

Internally, APL will call the `Area` method on each of the objects in turn, in the order in which they occur in the object list.

It is also possible to have vectors containing references to objects of different classes. Suppose that

we have a new class called `Square`, in addition to our `Circle` class, and that `Square` also has a class method called `Area`:

```
      FirstSquare ← ⎕NEW 'Square'
      ShapeList ← FirstCircle,FirstSquare
      ShapeList
[Circle] [Square]
      ShapeList.Area
314.1592654 100
```

Note that there is not necessarily any relationship between the `Area` methods in the `Circle` and `Square` classes.

What would happen if we create a new `Circle` object and immediately try to display its `radius` property?

```
      AnotherCircle ← ⎕NEW Circle
      AnotherCircle.radius
VALUE ERROR
      AnotherCircle.radius
      ^
```

Because the radius property of the new object has not been assigned, we get a `VALUE ERROR`; the new object is incomplete. Is there some way we can ensure that only completely-built objects are created by `⎕NEW`?

One way in which this can be done is to add a new class method called a **Constructor**. This is an APL method which has the same name as the class. The APL interpreter will call the constructor method automatically when a new object is created.

The Constructor for the Circle class might look something like this:

```
      ∇Circle R
  [1] radius ← R
      ∇
```

Note: We haven't worried too much yet about how to edit APL classes. APLX has a powerful class editor, which is what you would use in practice. For now, if you want to experiment with adding this constructor to your class, try:

```
      )EDIT Circle.Circle
```

The constructor method we added takes a right argument, which is the initial value of the new circle object's radius. To create the circle, the value is passed as an argument to `⎕NEW`:

```
      AnotherCircle ← ⎕NEW Circle 100
      AnotherCircle.radius
100
```

In some object-oriented languages, constructors are the only way of assigning initial values to properties. In APLX, it is also possible to specify the default value of any properties in the class definition. Instead of using a constructor, we could have changed our Circle class to specify that

the radius property of all new Circle objects should have an initial value of 10. This is discussed further in the section *Types of Property* below.

System Methods

As well as user-defined methods like `Area` above, each object/class can make use of a number of **system method**. There are around a dozen of these, but for now we will just note two. The system method `⎕CLASSNAME` returns the name of an object's class as a character vector:

```
      FirstCircle.⎕CLASSNAME
Circle
```

The system method `⎕NL` can be used to discover the names of the methods and properties in a class (or an instance of that class):

```
      Circle.⎕NL 3
Area
      FirstCircle.⎕NL 2
radius
```

Inheritance

Suppose that we wanted to re-write our example to handle two types of shape, circles and squares.

One way to do this would be to use a general class called `Shape`. We could add a property which specified the type of shape (0 for `Circle`, 1 for `Square`), and we could add an `Area` method, something like this:

```
    ∇R←Area
[1] :If type=0
[2]    ⍝ Circle
[3]    R ← (○1)×radius*2
[4] :Else
[5]    ⍝ Square
[6]    R ← sidelength*2
[7] :EndIf
    ∇
```

However, in object-oriented APL there is a much more elegant way to do this, by using **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.

So you might have a class `Shape`, representing an abstract geometric shape. This might have properties called `X` and `Y` giving the centre 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.

Definition of the `Area` method in the `Circle` class:

```
    ∇R←Area
[1] R ← (○1)×radius*2
    ∇
```

Definition of the `Area` method in the `Square` class:

```
    ∇R←Area
[1] R ← sidelength*2
    ∇
```

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. For example, if you are running a method defined in the base class `Shape`, and that method calls `Area`, the version of `Area` which gets called will be `Circle.Area` or `Square.Area` as appropriate.

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, an advanced feature known as **mixins** allows your objects to incorporate functionality from classes other than their ancestors, which is rather like multiple inheritance – see the *APL Language Reference* manual for further details.

Object References and Class References

When you create an object, i.e. an instance of a class (using the system function `⎕NEW`), the explicit result that is returned is not the object itself, but a **reference** to the object. This reference is held internally as just 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. This is discussed in more detail below.

Of course, because APL 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.

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 or System method may return a Null object, for example if you are looping through a linked list of objects and reach the last one. And 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), or to define a constant (such as a text string to appear on all invoices) for all members of the class.

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. Thus, 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.

Methods and properties are, by default, public, just as in conventional APL variables referenced in a function are global. (You can make them private by using the class editor, or localising them in the class header using the del editor.)

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 don't normally need to use dot notation. This is because the search order for symbols encountered when executing a method is as follows:

- First, APLX looks to see if the symbol refers to a member defined in the class of the object.
- 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.
- If it is not found in any of the ancestors, it then looks in the local variables of the method.
- 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
   ∇
```

Canonical Representation of a Class

The canonical (text) representation of a class is returned by ⎕CR, in exactly the same way as applies to ordinary APL functions and operators. 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 the whole line is enclosed in curly braces.

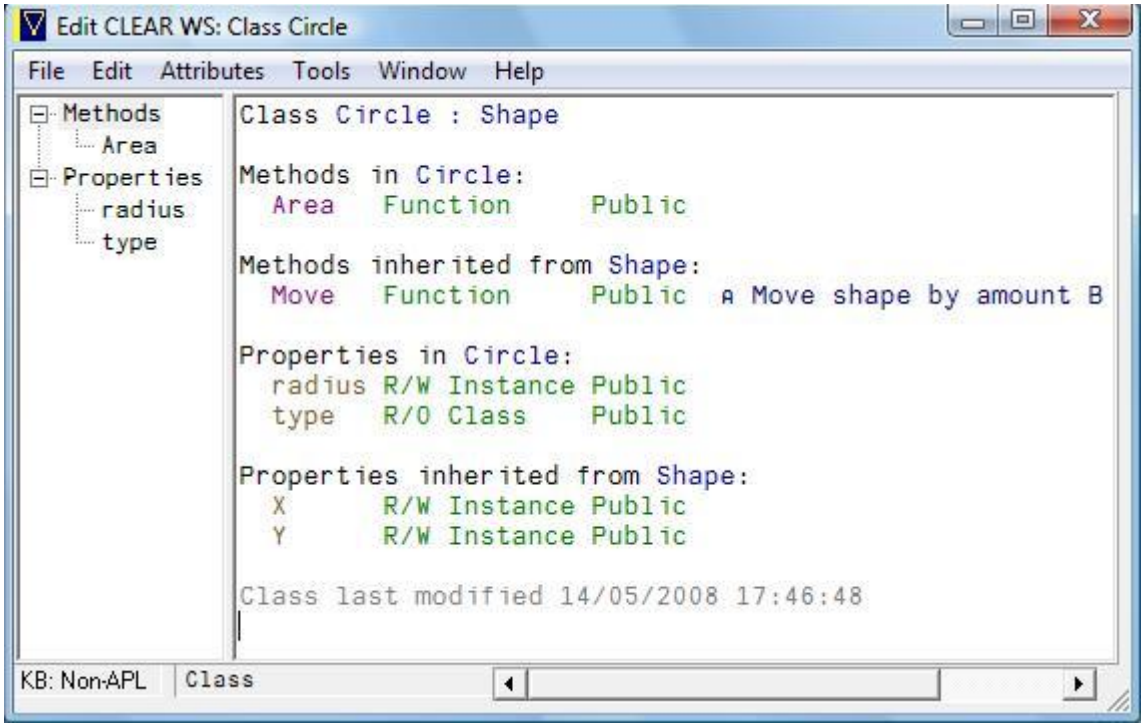
Any methods then follow, delimited by del characters, and a closing right curly brace ends the definition.

For example:

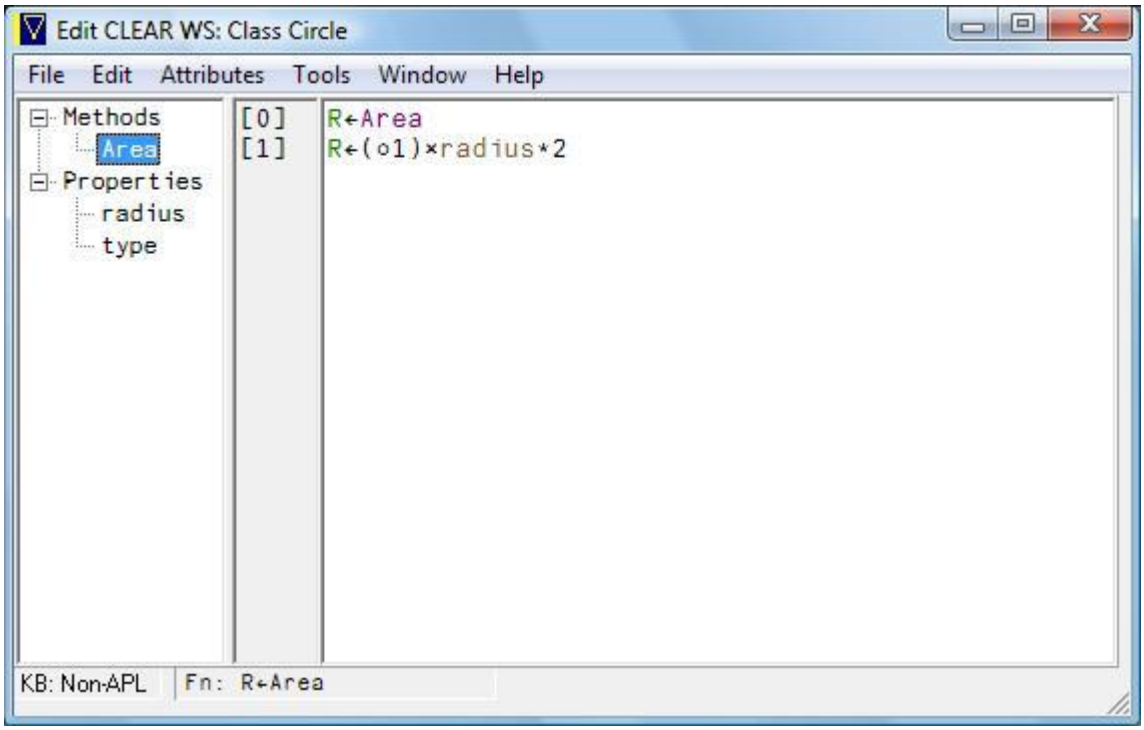
```
      ⎕CR 'Circle'
Circle : Shape {
radius
{type←←'circle'}

∇R←Area
R ← (○1)×radius*2
∇
}
```

As you would expect, ⎕FX can be used to turn the text form into a class definition in the workspace. In practice, though, you'll normally tend to use the APLX Class Editor, which shows the class as a whole:



...or allows you to edit a specific method or property:



You can also use the del editor, or you can edit individual methods or properties directly.

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. 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:

```
  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 can 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
```

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.

Creating objects (instances of classes)

As we have seen, 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'
```

Why do we need this alternative syntax? The reason is that you can specify an external class name (i.e. a class written in a different language, and existing outside the workspace). For this case, the left argument specifies the external environment, and the right argument the class name. For example:

```
      JAVADATE ← 'java' ⎕NEW 'java.util.Date'
```

We won't discuss external classes any further here - that is a whole subject in itself!

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 an APLX 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.

Using Classes without Instances

So far in this tutorial, 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 constants, and keeping a namespace tidy.

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 that the number of global functions was very large. By placing related functions in a class, the workspace can be kept tidy, without having to resort to local or dynamic functions.

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.

Finding out more

There is a lot more to APLX classes beyond this simple introduction. To find out more, take a look at the MicroAPL website and particularly the document New Features in Version 4.0 which includes a longer version of this tutorial.