I-APL Ltd.　　IAPL/Mac User Guide　　8/29/24
CONTENTS

IMPORTANT

I-APL is intended for home and educational use, and to allow businesses to evaluate APL for very little cost. It is not designed for commercial programming or exploitation.

I-APL takes more computer time than BASIC to perform some tasks, but the competent APL user needs less time to formulate those tasks.

Program development time is the sum of programmer thinking time and computer execution time. I-APL may be slower than BASIC for the latter, but is much faster for the former, so that overall I-APL is faster than BASIC.

Where I-APL is designed to be small and portable, commercial APLs are designed to be fast and to integrate well with their system environments. They are available for most major makes of mainframe, mini and micro computer. They offer larger workspaces and better program development facilities than I-APL, as well as providing filing systems, commercial formatting, and many other extras.

IAPL/Mac
Instruction Manual

I-APL for the Apple Macintosh
Version 1.

by

Anthony Camacho
Paul Chapman
Ian Clark
David Ziemann

Preface

This book is provided as part of the IAPL/Mac product.

Chapters 1 to 5 contain an overview of the International APL project (I-APL) and the IAPL/Mac interpreter, which runs on the Apple Macintosh computer.

Appendix A contains the standard conformance document required by the draft ISO International APL Standard DIS 8485. This includes a description of the areas where IAPL/Mac exceeds the requirements of the Standard.

Appendix B contains a list of the results produced by system function quadID for all I-APL implementations current at the time of printing, to aid writers of portable I-APL workspaces.

Chapter 1: History and aims of the I-APL Project

At the international APL conference in Manchester in July 1986 an idea of Paul Chapman's was taken up by a group of enthusiasts. The idea was that it would be possible to write a full ISO compatible APL interpreter in 25K which would run on school and home computers. The I-APL project was founded with a committee of five: Ed Cherlin, Editor of APL Market News, and Anthony Camacho, then secretary of the APL Association, were to be joint chairmen; Norman Thomson and Howard Peelle were to be education officers and David Ziemann was the technical expert. Fundraising began that summer and by October there was sufficient to authorise Paul to begin work.

Project Aims

The project aims were to write and issue an APL interpreter which would run on as many school and home computers as possible, and be available in any part of the world. There would be no charge for the software but there would have to be some charge for the medium and for copying and also for
the books which would go with it. The project and its supporters believe that if APL is to grow it must be made available in schools. Experience has taught us that efforts to interest school teachers cease to be effective as soon as the teachers discover that APL would be very expensive for them to try. By removing the cost barrier it was hoped that many people could be persuaded to try APL and that many APLers would be encouraged to introduce APL to teachers. The project has always seen the production of the interpreter as the first step and probably not the hardest step in the difficult job of getting APL into widespread educational use.

Progress

Paul Chapman finished the interpreter on 4 July 1987 and debugging, optimisation, improvements and customisation to the PC occupied the next six months. Version 1.0 was issued in January 1988 and we sold out of manuals in August 1988. Version 1.1 was completed in October 1988 and cures all the V.1.0 bugs reported to us as well as adding some new facilities. The most important of the new features is the option to send output to the screen through the BIOS; this allows PC clones whose display is not hardware-compatible to scroll the display correctly, though slower than the
standard version.

The job of porting I-APL requires intimate knowledge of the operating system and reasonable competence at producing machine code programs for the chosen machine. What a porter has to do is to write an interpreter for the specially invented language DE in which the APL interpreter is written, and link it to the operating system of the machine for input from keyboard, for output to screen, printer etc. and to give at least)SAVE and)LOAD access to the filing system for workspaces.

In addition to the porting job we like to have some workspaces which use the quad-MC feature to give access from APL to the machine's filing system, graphics, colours and sound.

If you would like to have a go at a port please contact the project at the address below. If there is already someone working on your chosen machine we will put you in touch. They may give you a flying start or you may find you have just the knowledge which is holding

things up. There is 8086, Z80, 6502 and 68008 machine code already available for most of DEGO.

To get yourself on the mailing list for your machine, write to the I-APL Project, 2 Blenheim Road, St Albans, Herts, AL1 4NR, UK and mention the machine you want to put it on.

The Interpreter

The I-APL interpreter was written to take the minimum program space. This is necessary so that it can be used on the small computers generally in use in schools. For example addresses are held in 16 bits, so a BBC B, a CP/M machine (Z80), a CBM64 or a Spectrum can use all its directly addressable memory.

If your computer has more than 64K of memory I-APL will not be able to use the extra for the workspace itself. A clever porter may be able to store fast machine code routines in such memory and access them from within APL by a far jump; this is why the PC version uses 256K of memory - the DFILE, PGRAPH, FSCREEN workspaces are very small and most of the code is outside the 64 K range. In the Apple Macintosh version, all extended code facilities are held in separate machine-code resources, which are read-into memory as needed. Whenever there was a choice whether to make I-APL fast or small we chose to make it small.

Variations between versions

Because the interpreter is itelf interpreted, the APL is exactly the same on every machine. The only things that vary are the keyboard, the screen, the operating system (and the access to it), the details of how one can write machine code within APL and the method of dealing with printers and other peripheral devices. The porter writes the interpreter for DE in the machine code for the CPU chip at the heart of your machine. This program is called DEGO.

Workspaces are compatible

When you save an APL program, what is saved is the complete workspace. A workspace contains functions (the program), variables and all the APL system settings. If your program needs a print width of 192 you can set it to that, save the workspace and when it is reloaded the print width you set is also reloaded. A saved workspace will also contain the APL stack so a program that has been interrupted by an error or your intervention can be saved and then when reloaded can be restarted at exactly the point where it was stopped.

Apart from use of the machine code call function, all I-APL workspaces will run on any I-APL machine. All you have to do is transfer the complete file from the disk or tape format of one machine to that of another (note that the first two bytes in the file give the length in bytes of the rest of the file. For example if the total file is 15 bytes long the first two bytes will be 0D 00. If your file transfer mechanism pads the length you will have to readjust it.

Thus a group of people using quite different computers can still use exactly the same APL, loading the same workspaces and getting the same results. The only difference will be in the keys they have to press and in the display and perhaps the print they get.

The essential books for the project are as follows:

—   This instruction manual.

—   A tutorial for total beginners in APL. Norman Thomson produced the Tutorial and Linda Alvord and others helped with material.

—   An APL reference manual, known as "An Encyclopaedia of APL" by Garry Helzer. Garry had been considering the desirability of producing an APL encyclopaedia with the primitives arranged in alphabetical order and the project provided the need and incentive to do it.

There are already several I-APL workspaces. These are available from the British APL Association Software library. For details see a recent copy of Vector, the Association's magazine. Some may be available through the CIX or other bulletin boards (join the APL conference).

I-APL can be used with any of the many APL books. In particular it includes both the "Del editor" which is part of the ISO standard and "Direct Definition" which is used by many of the best textbooks on APL. APL booklists are offered by the APL Association, and by Renaissance Data Systems, PO Box 20023, Cathedral Finance Station, New York, New York 10025-1510, U. S. A. For a British source of APL books see *Vector*.

Chapter 2: The APL Notation

APL is primarily an imperative notation. This means that what you write in it is instructions or commands (not statements or questions or interjections). In this it is like other computer languages. It was invented by Ken Iverson because he saw the need for such a notation for such purposes as description of the detailed internal workings of computers. Computers have such a great many potential states that engineering drawings cannot show them in a comprehensible way. Attempts to describe the states in English become far too bulky - they begin to read like legal documents.

The many anomalies in conventional mathematical notation (convenience has been more important than consistency to mathematicians on many occasions over the last three or four hundred years) made it unsuitable for the expression of algorithms. Furthermore each variable in conventional notation usually only holds a single item and attempts to indicate a range (typically by using a row of dots to give the idea) are unsuitable for rigorous interpretation. APL extends the core of mathematical notation in two ways.

    —   Its variables can be arrays. This calls for consistent rules to handle the different shapes and for making selections from or altering the shape of such variables.

    —   The range of functions is extended to include all of logic, many new functions for processing arrays and operators which enhance and vary the effect of functions.

An APL variable is a name associated with a rectangular array of any size and number of dimensions. The functions of logic and mathematics are extended to deal with such array variables. The single value becomes a special case and consistency is maintained for arrays which are empty.

APL has a great many primitive functions which are not available in other notations or other computer languages. For example a single symbol specifies matrix division and another specifies conversion between decimal and any multi-radix arithmetic. It also includes operators which enhance the simple functions so that matrix multiplication is specified by three symbols or adding across or down a table is specified with two symbols. The consequence is that APL is at least as concise as any other notation and much more concise than any other programming language, but with a far wider choice of functions.

The first major task APL was used for was to define the internal workings of the IBM 360 computer. APL was used for six or eight years before anyone thought it might be possible to program a computer to obey a reasonable selection of APL functions. APL began as a means of communication between people and is still used for that purpose.

In the evaluation of expressions APL is like mathematics; the result of evaluating any expression replaces the expression for the next stage of evaluation. In a formula (3+4) is replaced by 7. What you write in APL is more functions which, like plus, have a left and a right argument (the 3 and the 4) or only one argument like factorial or even no argument. The functions you write are called 'defined functions'. Each defined function returns a result or by performance of some action gives an implicit result in the form of a screen display or output on a printer or other device. In APL the syntax for defined functions is very similar to that for primitive functions: the same rules of precedence apply, the maximum number of arguments is two and in general they return a result for the rest of the expression to use.

When APL was implemented many people realised that it was the most concise and analytic of all the computer languages. During the implementation characters which

would not fit on a single line and superscripts and subscripts were excluded. Its character set was partly determined by the technical needs of the golf-ball typewriter which was originally used to print APL.

The character set has changed very little since then: many people have regretted that it causes difficulty in implementing APL on some computers but the several attempts to replace the symbols by keywords have never shown any hope of replacing the symbols for most users. The symbols are an essential part of the notation; it is only people who view APL as just another programming language that want to or could use keywords.

Chapter 3: What I-APL is good for

I-APL is designed for educational use. It is primarily a learning tool.

It is a complete APL and conforms to the International Standards Organisation standard for APL (which is also expected to become a British Standard).

Its execution speed is adequate for learning purposes provided that array sizes are kept small enough to be displayed on a single screen. Its workspaces are large enough for complex work to be done on such arrays, but as workspace and interpreter (and, in some machines, operating system scratch memory, buffers and screen memory) all have to be contained in a 64K address space, no machine will be able to provide a clear workspace of more than 32K, however large its memory.

To reduce the size of arrays of real (floating point) numbers they are held in I-APL less precisely than in commercial APLs. The accuracy of logarithmic and trigonometrical functions is also less than most commercial systems; these limitations are to make the interpreter as economical of space and as fast as possible.

The ISO standard does not require that a filing system is provided and I-APL does not include one. Nor does it include commercial formatting or a full screen facility.

I-APL does not cater for nested or heterogeneous arrays or the extended functions and operators of enhanced APLs.

When you have learned APL and begin to notice the limitations of I-APL, you should consider using one of the commercial APLs advertised in this manual or which you may learn about from the British APL Association and its magazine *Vector*.

In the USA contact SigAPL via the ACM. SigAPL's journal is called *Quote-Quad*.

Commercial APLs offer greater speed, precision, accuracy and space. They are often beautifully integrated into the operating philosophy of the machines they run on and most offer filing systems and a wide range of enhancements - new and exciting functions and operators and heterogeneous and nested arrays. By the time you are ready for them you will know enough about APL to be able to choose between the rival attractions of the many alternatives open to you.

I-APL is compatible with all known complete APLs and what you learn here will be valuable to you whichever APL you use next. This manual highlights the few features in I-APL which may not work on every APL. If you avoid these it will be easy to transfer your I-APL workspaces into any other APL environment and get them working.

An I-APL workspace which does not use the quadMC and quadTX functions can be transferred between any I-APL machine and any other and will produce exactly the same results wherever it is run, provided there is enough memory.

The knowledge of APL you acquire while using I-APL will similarly be transferable to any other APL environment and with equally little loss.

Chapter 4: Using IAPL/Mac
Getting Started with IAPL/Mac

When you load your distribution diskette you will see a window like this:
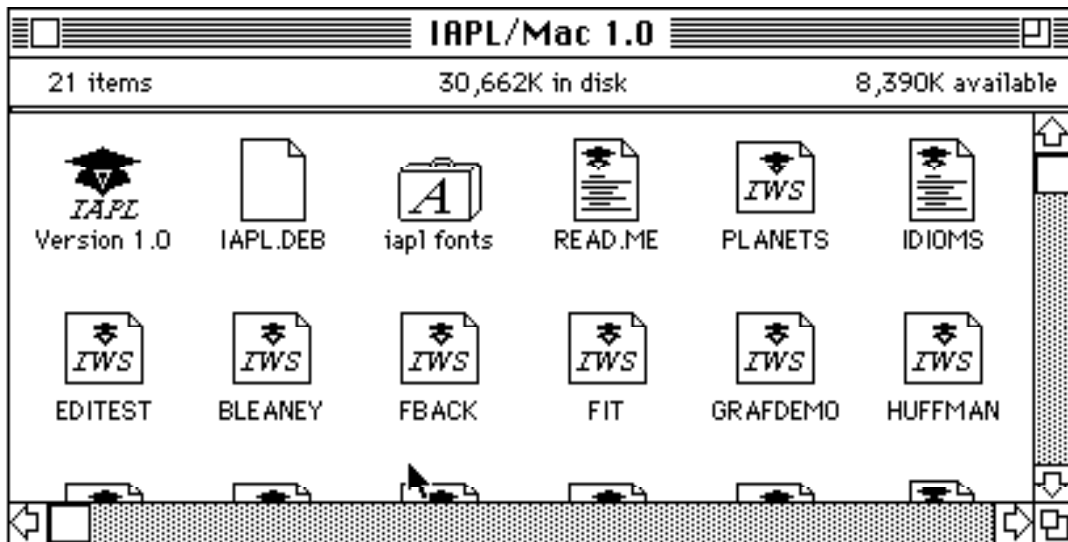
Figure 1. The distribution disk desktop with program and workspace icons.

You can see examples of all the different sorts of icons along the top row. In turn they are:

(Mortarboard). The IAPL/Mac Interpreter itself, here called Version 1.0.

(Plain). IAPL.DEB. You needn't bother what this file does. But it must be in the same folder or desktop as the Mortarboard Icon (the Interpreter) when you try to run it. It contains the "portable" part of I-APL which is common to all the ports on different machines. It is NOT "double-clickable".

(Suitcase). The file: iapl fonts. This is a standard Font file. It contains fonts called Hex and St Albans. You needn't bother with Hex for now, but in time you may be pleased you've got it. If you're a programmer it helps you decypher funny characters in text files. The other two fonts are both St Albans, 9- and 18-point. If you want to print documents containing APL symbols you need to move this font family into your System file by the usual method. If you simply want to run IAPL/Mac for viewing on the screen then don't bother to do this yet. The IAPL/Mac Application has its own copies of St Albans 9 and 18. It will use these whether or not they are (also) in the System file.

(IAPL Document) READ.ME . This is a so-called "text only" file which all word-processors will accept. However it happens to be "double-clickable". Opening this type of file will run IAPL and load the contents of the file into the Session Window. See **Using Text-Only Files** below for what you can do with such files.

(IWS Workspace) PLANETS. This is what corresponds to a "user program" in APL. It is an integrated file containing data, functions and environment. IWS files are easily transferred between the different ports of I-APL. Data and functions can easily be copied, one at a time, from one workspace to another, or entire workspaces can be merged. There are several other sample workspaces on the distribution diskette.

Start IAPL/Mac by double-clicking on one of the following kinds of icon:
— On the Mortarboard icon (to start off with a clear workspace)
— On one of the workspace icons, e.g. PLANETS, which will load PLANETS immediately
   after starting
— On one of the IAPL Document files, which starts IAPL/Mac with a clear workspace but
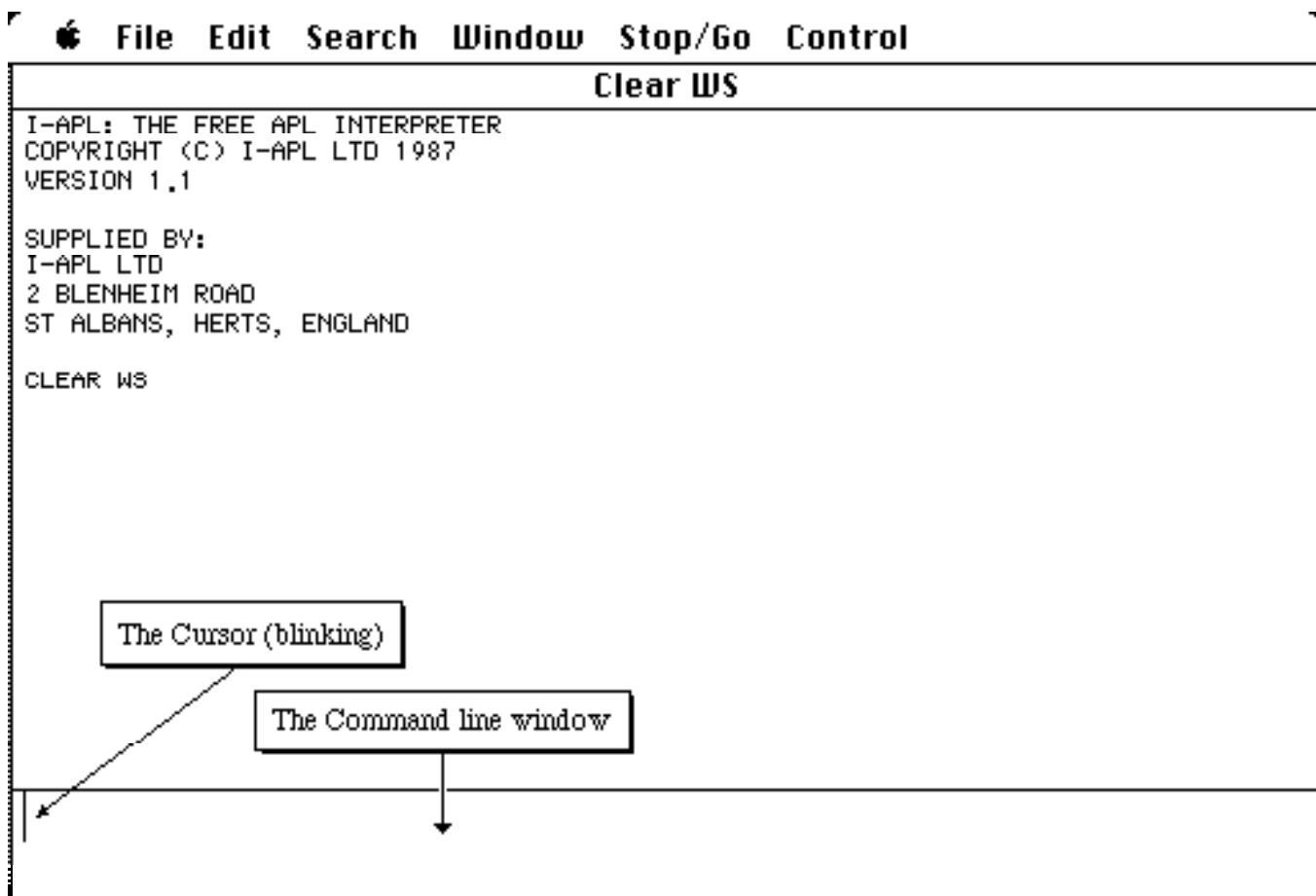   with the file's contents visible in the Session Window.

Figure 2. Initial appearance of IAPL/Mac.

Start IAPL/Mac by double-clicking on the mortarboard icon. You will shortly see a display like that of figure 2 (above).

Conventional APL gives you a six-blank prompt, but IAPL/Mac prompts you by displaying a window called the Command line, with no leading blanks. It is now waiting for you to type something on the keyboard. Alternatively you can pull-down a menu and select an item.

Type-in an expression, e.g. 123+456, and press the L-shaped key, {return}. Your keyboard may also have a key labelled {enter}, which IAPL/Mac accepts as the same thing. As you type-in the expression, you see the letters appear twice-sized in the Command line, as shown in figure 3.

Figure 3. Typing an expression into the Command Line.

On hitting {return} your input, and IAPL's reply, will appear in the Session Window, as figure 4 shows.
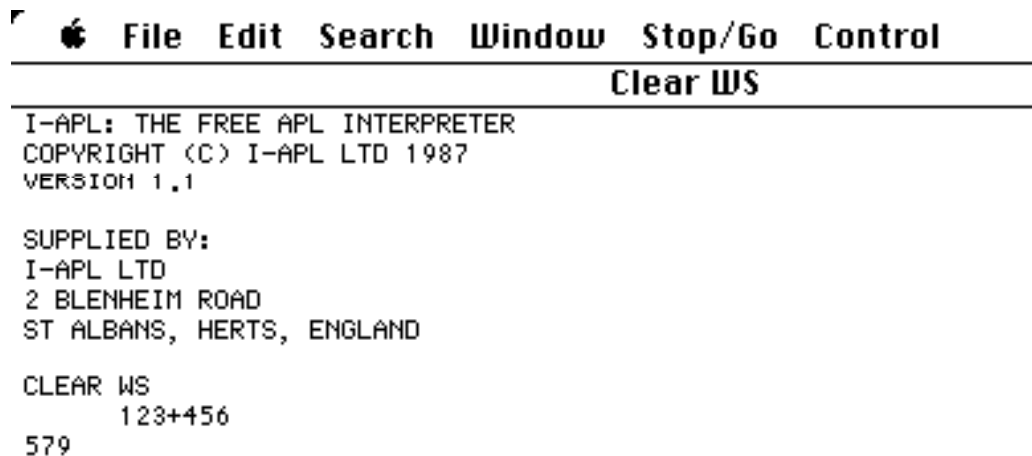
Figure 4. The result of inputting an expression.

At this level of use, IAPL/Mac is just a glorified calculating machine, with the advantage of logging the calculations. However it can be used to do a little more than that. It lets you write teaching materials to convey advanced mathematical, scientific or engineering concepts to students. If you teach in terms of the APL notation, then the student can play with your examples on a machine which actually executes the formulae you write on the blackboard (or on a worksheet, or in a textbook).

To prove the point, how about a quick lesson in statistics?

Here is a sequence of numbers. It might be rainfall on successive days, or the daily closing prices of given shares, or any series of observations you like. Type them in (with a space between each one) and press {return}.

```
× 7.25 7 ×.8 ×.×7 5 4 ∧ 4.2 ×.⊂
```

APL computes your input, but in this case ends up with the same thing. So it simply echoes what you typed in.

× 7.25 7 ×.8 ×.×7 5 4 ∧ 4.2 ×.⊂
× 7.25 7 ×.8 ×.×7 5 4 ∧ 4.2 ×.⊂

This is a time-series. People usually refer to a time-series as X. So let us place this sequence of numbers in X. Get the numbers back into the Command line. To do this, double-click on the line of numbers in the session log. Or if you're not handy at double-clicking, sinply click once on the line (this is called "selecting" it — the whole line blackens) and choose Bring Down from the Edit menu (or ⌘B for short).

The Command Line now looks like this (see figure 5):
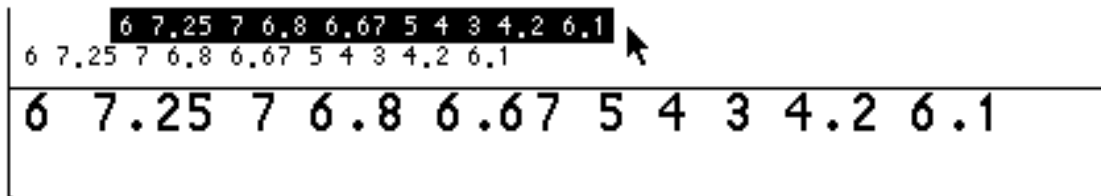
Figure 5. Bringing back a line from the Session Log.

Put the text cursor at the front of the Command Line, either by clicking there with the mouse or by using the up-arrow key. Now type X¢ in front of the string of numbers and press {return}.
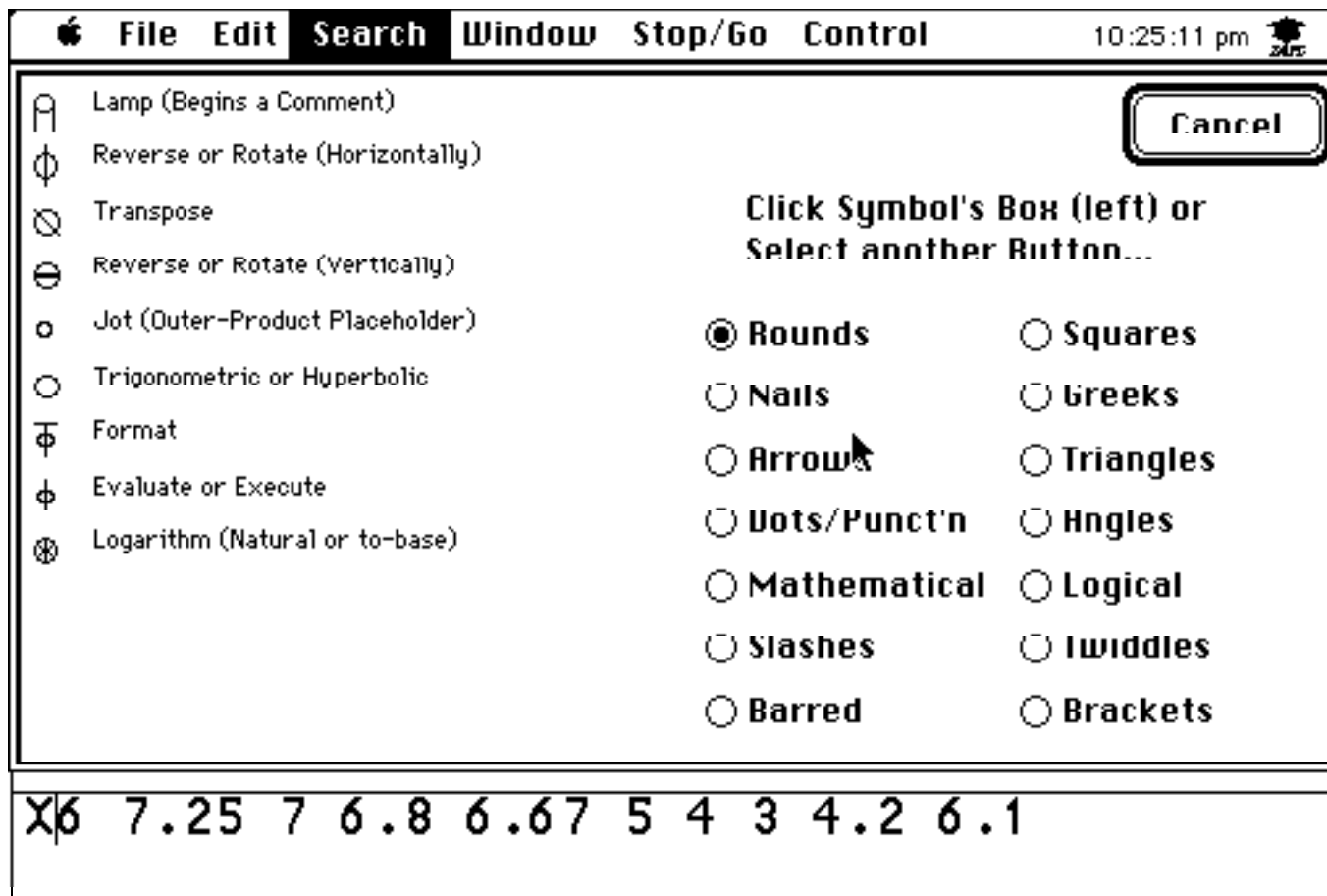
Figure 6. Using Symbols... in the Search menu (1).

What? Can't find the ¢ key? You novice, you. Fortunately there is a menu to help you (there are two if you include Help... in the Window menu, which shows you the IAPL keyboard layout). But unless you like hunting for keys on a keyboard layout,

why not pull down Symbols... in the Search menu. This is what you see (figure 6):

The special APL symbols are here classified by shape. You remember you want an arrow. So click the Arrows button. A new set of symbols arppears. The one you want just happens to be at the top left corner (see figure 7).
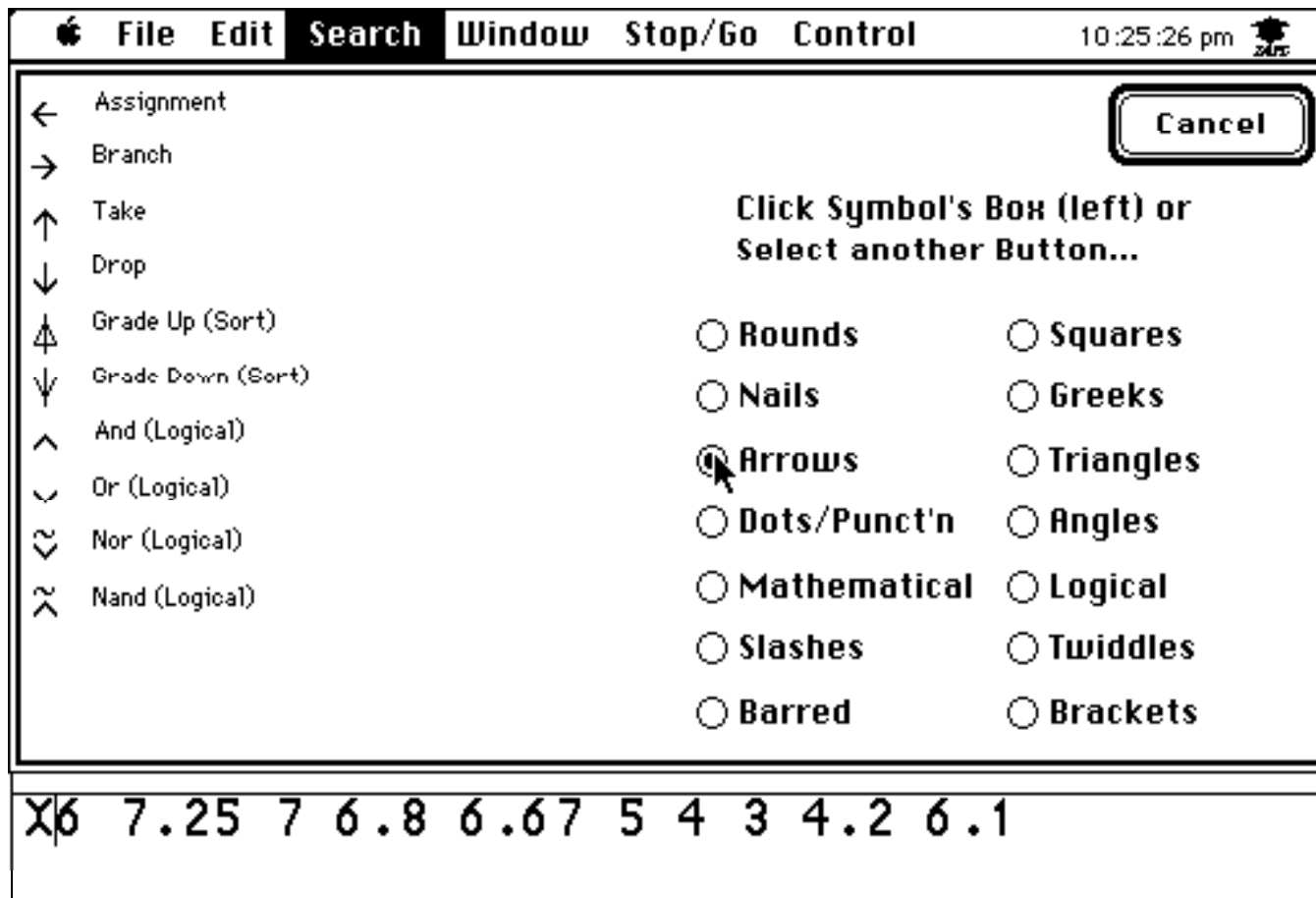
Figure 7. Using Symbols... in the Search menu (2).

Click on it. The dialog box goes away and the ¢ symbol obligingly appears as if you had typed it (see figure 8).
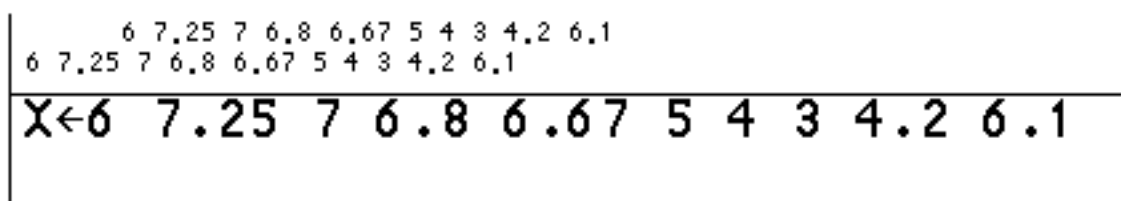
Figure 8. Composing new input by overwriting a brought-back line.

Since you will need this symbol a lot, you might as well learn right now that it is {shift-minus}.

Now you have assigned the time-series to X. Inputting just X will output the numbers again. Try typing X+1

```
    X≠⊂
7 8.25 8 7.8 7.×7 × 5 4 5.2 7.⊂
```

APL is just as happy dealing with whole series as it is with single numbers. It adds 1 to each number.

How many numbers does X contain? You find the "shape" of any variable or expression by placing ⍴ (Greek "rho") in front of it. Rho is got by {shift-R} — that's easy to remember. But if you can't remember it, choose Symbols... again from the Search menu and click the button for Greeks.

```
    ⍴X
⊂0
```

However, you're supposed to be giving a statistics lesson, not a Greek lesson. You don't want to have to explain to the students what ⍴ is. So write yourself a function called something sensible like COUNT to count up the terms of any variable (or expression).

Turn that last expression into a function. Double-click the line ⍴X to re-work it. There's little point in avoiding typing it in afresh with so simple an expression, but there might be with a much longer one.

This time, overtype the line to read:

```
    COUNT: ⍴X
```

and enter it. APL makes no reply (which is generally a good thing) but IAPL/Mac shows it is working by changing the mouse pointer to a spinning "beachball". Test COUNT by typing:

```
    COUNT
⊂0
```

Great. But COUNT is only ever going to count the elements of the variable X. How do you make it count up anything? Well, double-click on the line saying COUNT: ⍴X once more and turn the x into ⍵ ("omega"). To save you using Symbols... again, ⍵ is {shift-W}. It is a special symbol which behaves like a variable name, but inside a function definition it takes the value of the right-hand argument of that function (but it can be reassigned). So now you have:
COUNT: ⍴⍵

Test it with various different right-hand arguments:

```
    COUNT X
⊂0
    COUNT ⊂ 2 ∧
∧
    COUNT 4 5 ×
∧
    COUNT X,X                    (— that's X concatenated with itself)
20
    COUNT ⊂⊂/5                   (— that's 99 repetitions of the number 5)
⊂⊂
```

So far, so good. Another thing you can do is sum all the terms of X. Try it:

$\neq/X$

$5\times.02$

That gives us the basis for another useful function:

    SUM: ≠/˘

Now we can start some statistics. Notice that there's no need to teach the class APL in order to do this. However we need to warn those keenies who taught themselves BASIC at home that APL uses the conventional mathematical symbols for addition, subtraction, multiplication and division, namely (≠ - ∂ ∞). Asterisk (∗) means "to the power of", not "times" and Slash (/) is not "divide", but is the replication operator which we have already used in ≠/X and ⊂⊂/5.

The average, or sample mean, of X is:

    ∩SUM X)∞∩COUNT X)
5.×02

We can define the function MEAN as:

    MEAN:∩SUM ˘)∞∩COUNT ˘)

After that, variance and standard deviation come pretty naturally, not to mention all the other statistics defined in terms of expectations:

    SQUARED: ˘∗2
    ROOT: ˘∗.5
    VARIANCE: MEAN SQUARED ∩˘-MEAN ˘)
    STANDARDDEVIATION: ROOT VARIANCE ˘

Incidentally, the parentheses in VARIANCE: are redundant, but put them in for the students' sake, or they'll be asking "Please Sir, does SQUARED apply just to the ˘ or to the whole lot?" In fact SQUARED squares each of the elements in the expression, ˘, it operates upon. Since you'll be summing squares a lot, you could write more little functions, getting the students to assure themselves by trials that they come to the same thing:

    SUMSQ: ≠/∩˘∗2) Ω=SUM OF SQUARES
    RMS: ∩MEAN ∩˘∗2))∗.5 Ω=ROOT-MEAN-SQUARE
    STANDARDDEVIATION: RMS ∩˘-MEAN ˘)

It's easier than all those sigmas, isn't it?

You see that IAPL/Mac lets you assign your own variables and define your own little functions, which you can use in place of numbers in the expressions you type in. You can embed expressions in functions which call other functions, building up your application bit-by-bit. If you choose, you can end up with a single function called ΓO (or anything you like — there are no "reserved words" in APL) which runs a highly sophisticated program requiring no expressions to be typed-in at all. Finally you can ask APL to run an expression consisting of nothing but the word ΓO as soon as your workspace is loaded. Then you have built what they call in the trade a "turnkey" system, or in MacSpeak, a "double-clickable application". Your lucky recipient can double-click the icon and run your application and need never know it's APL (though he'll need the IAPL/Mac files around).

Your distribution diskette contains just such a workspace, called PLANETS. It is designed to

give you a "taste of APL" (strongly flavored, as it happens — you are not being invited to write APL in this way). It runs on whatever port of IAPL you are using without you having to do anything except hit the {return} key.

Let's load the workspace called PLANETS.

Pull down the File Menu and select Load... (or hit ⌘O for "open"). IAPL/Mac will then invite you to choose a workspace from a scrollable list. This is a standard facility with Macintosh applications, allowing you to navigate through different folders and even through different disks and drives to find the data file you want. It does so by showing you a so-called "dialog box" resembling Figure 9:
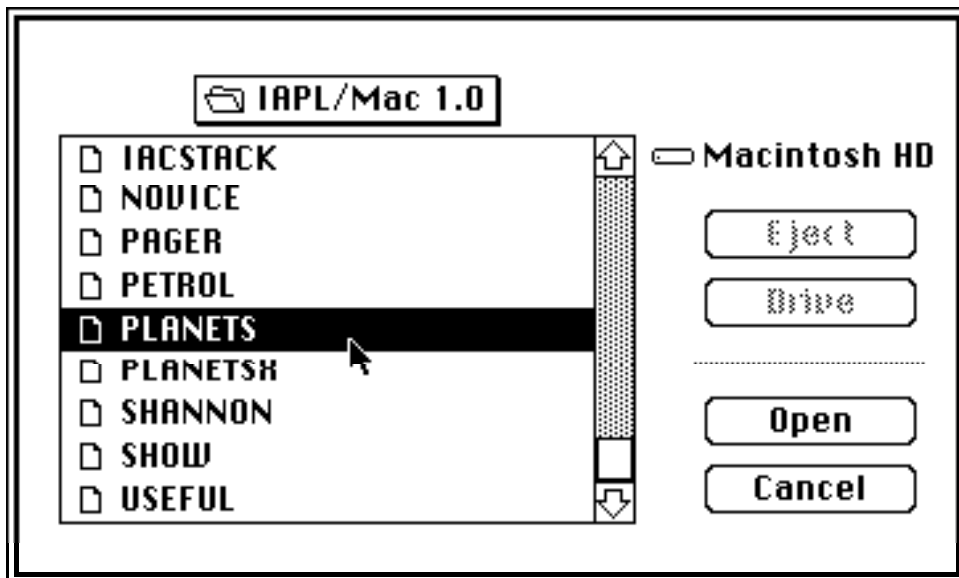
Figure 9. Dialog box shown by IAPL/Mac when helping you to locate a workspace.

You must interact with the dialog box to choose a workspace or else make it go away by hitting the Cancel button.

Suppose you locate PLANETS as figure 9 shows, by double-clicking on the line shown blackened. Then IAPL/Mac will automatically generate the command:)LOAD PLANETS as if you had typed it yourself. It is better than typing it yourself, however, because the dialog also tells IAPL the correct folder you found your workspace in.

After a short interval, the screen will look like figure 10 (below). PLANETS has started immediately after loading (because the writer of PLANETS has used the πLX facility of ISO-standard APL). The function in control is now waiting for you to hit {enter} to execute the next step of the demonstration.

Instead of hitting {enter}, hit ⌘. or choose Abort from the Stop/Go menu. This interrupts the running function and causes APL to prompt you with the Command line for input. You have now broken into the demo, which would otherwise roll on to its end, disregarding whatever you typed-in and treating it as if you had just been hitting {enter}. Now you can, if you choose, re-start the demo by typing:

TASTER

which happens to be an APL Expression. It is an expression in terms of a function which takes no arguments, but runs the whole demo. The workspace is "turnkeyed" by placing the expression TASTER in the so-called "latent expression" of the workspace before saving it. You

can do this yourself by typing:

πLX¢∪TASTERℝ∪

Note that πLX must be assigned a string value, hence ᴜTASTERᴜ instead of TASTER.


```
   ✕  File  Edit  Search  Window  Stop/Go  Control          10:03:48 pm

                              PLANETS

SUPPLIED BY:
I-APL LTD
2 BLENHEIM ROAD
ST ALBANS, HERTS, ENGLAND

CLEAR WS
       )LOAD PLANETS
PLANETS
This workspace gives a taste of APL using only  command mode to do some
simple things with three variables already stored in the workspace.
The three variables are PLANETS a list of planet  names, PTABLE a table
of numeric facts about the planets (accuracy not guaranteed) and PHEAD
a heading to a table of names and the facts.

To go through the demonstration just press the enter key whenever you
are ready.
CLEAR WS
       )LOAD PLANETS
PLANETS
       )VARS
PHEAD PLANETS PTABLE
       PLANETS
                                              Press ENTER when ready
```

Figure 10. Screen appearance after loading the distributed workspace PLANETS

Instead of using Load... in the File menu you can load a function the conventional APL way
by typing the command:

)LOAD PLANETS

However IAPL/Mac will only succeed in finding the workspace you want to load if it is
"near" the Mortarboard icon you opened to run IAPL/Mac, or "near" the last workspace you
successfully loaded. In this context, "near" means residing in the same folder, or visible in the
same window.

```
To go through the demonstration just press the enter key
are ready.
CLEAR WS
       )LOAD PLANETS
PLANETS
       )VARS
PHEAD PLANETS PTABLE
       PLANETS

→
SHOW[7]
```

)OFF

Figure 11. Partial

screen appearance after aborting execution and typing:)OFF

Now let's end this session by typing:)OFF and pressing {return}. See figure 11. IAPL/Mac will terminate and you will see the window showing the Mortarboard icon once again.

Using the pull-down menus

IAPL/Mac can be used like a traditional APL system, except that)LIB gives you a display like that of figure 9 instead of the conventional list of workspaces in the session log. However IAPL/Mac allows you to work in a more Macintosh-like way, by using pull-down menus and dialogs.

The menus in IAPL/Mac allow you to do all the things you can do by typing APL commands like)LOAD)SAVE and)OFF. Usually these generate the appropriate commands for you (to mark the session log and preserve compatibility with standard APL generally), but often they do more for you than simply typing the command would do.

Hence if you are new to APL, or don't particularly care to type-in commands, use the pull-down menus.

Besides offering an alternative interface to the command sub-language which is part of APL, certain pull-down menus help you compose APL expressions. In particular there are menu items to do the following:

• help you find (and input) a particular APL symbol
• help you identify a particular symbol whose meaning you might have forgotten
• help you locate a suitable idiom and type it in for you
• help you re-input an expression or part of an expression (with or without changes) which you have at some time already typed-in
• attach whole expressions to menu items and their corresponding ⌘-keys, to give you a small stock of instant inputs.

Menu: **File**

New   ⌘N
This generates the)CLEAR command. The result is an empty workspace called CLEAR WS.

Load...   ⌘O
This puts up the standard dialog box of figure 9 inviting the user to select a workspace by name. Only the names of workspaces are shown in the list, not other files.

Save   ⌘S
This simply generates the)SAVE command. The current workspace will be copied to disk.

Save As...
This puts up a standard kind of dialog box inviting the user to save the workspace under the current workspace name, but providing the usual options to change the name or navigate to a different disk or folder before saving. Pressing the Save button makes IAPL/Mac generate the command:
)SAVE NNNN using the existing or altered name NNNN.

Drop...   ⌘D

This puts up a similar dialog box to Load... but the result of selecting one of the listed file is to erase it, not to load it. Only workspaces are listed.

Copy Item...

This puts up a list of workspaces like Load... but as soon as you select a workspace to copy from (call it MYWS), IAPL/Mac "obtains" a name of an item (function or variable, call it MYITEM). IAPL/Mac then generates the command:)COPY MYWS MYITEM

When we say IAPL/Mac "obtains" a name, this means that it will look for a valid item name in the following places in turn:

— the selected word in the Command line

— (if no selection) the first word comprising a valid item name in the Command line

— (if the Command line is empty) the first word comprising a valid item name in the selected line of the Session window

— (if no Session line currently selected) the response from a dialog box looking like figure 12.

The dialog box only accepts the ASCII keyboard, which means that normally you must type in uppercase, e.g. MYITEM must be typed into the box as: MYITEM not: myitem.

If you simply press {return} in answer to the dialog box, i.e. you input a blank name, then IAPL/Mac generates the command:)COPY MYWS

The effect of this is to copy all items (functions and variables) of workspace MYWS into your current workspace.
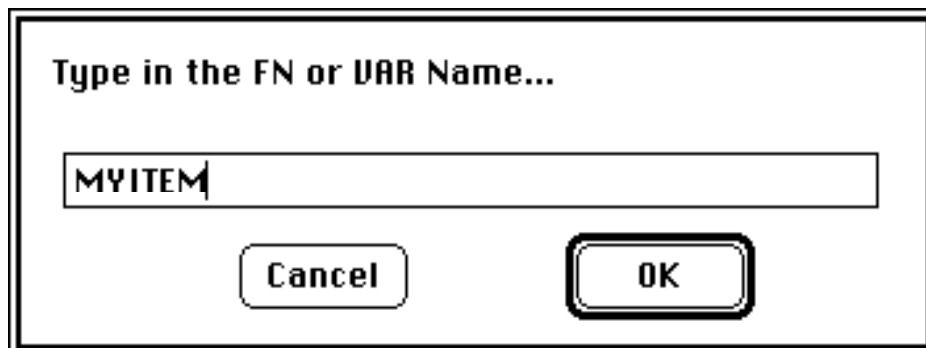
Figure 12. Dialog box which appears when no Copy Item implicitly given.

Launder...

This puts up a similar dialog box to Load..., but now all files in the current folder are visible, not only those files with the "workspace" icon . Selecting a file gives it the "workspace" icon. Thereafter it will be visible using Load.... This facility is provided for importing workspaces from a non-Mac system. No translation is needed when copying across an IAPL workspace in binary form. However, having done so, the resulting files will not show the proper *IWS* icon on the desktop and will not be visible to Load... until they have been 'laundered' by using this menu item. No check is made to see if the file being "laundered" is in fact a genuine candidate IAPL workspace — it is your responsibility to ensure this.

Page Setup...
Print   ⌘P
These menu facilities set up and print the contents of the frontmost window (the Session window or the Edit window) on the attached printer. They behave like they do in a normal Macintosh word processor or editor. Any so-called "text-only" file can be printed in this way (in the special IAPL/Mac font St Albans) if you first load it into the Session log or Edit window by the Load Window... menu option (see below).

Clear Window   ⌘*
This empties the frontmost window. Typically this is either the Command Line, the Edit window or the Session window. You can use this to discard the unwanted contents of the Session log if you feel this is getting too large, or wish to start afresh in order to compose a sample session.

Load Window...
This shows you a list of files resembling figure 9, but consisting of text-only files (including any files saved by the Save Window option to be described).

Save Window
This saves the Session log (or the contents of the Edit window, if that is currently to the fore) as a recallable file, making the lines and expressions in it re-inputtable. See section below on the use of saved session logs, also the Bring Down and Re-Input items of the Edit menu.

The saved file is given a desktop icon looking like this: It behaves, for all intents and

purposes, like a text-only file such as most word-processors and the Edit utility recognise. It carries no font information, but if loaded into a word-processor such as Microsoft Word and converted to font St Albans in 9-point size, it will reappear as it did in the Session log. It will start IAPL/Mac and pre-load itself if you double-click it (but IAPL/Mac will nevertheless start with a Clear WS).

Save Window As...
This saves the Session log (or the contents of the Edit window, if that is currently the frontmost window) as a recallable file as above, but allowing you to give it a new name first.

Quit   ⌘Q
Terminates the IAPL/Mac session, first inviting you to save the workspace if it has been altered.
Menu: **Edit**

Undo   ⌘Z
Works only for cut and paste operations on the Edit Window. It will not back-out of an APL operation.

Cut   ⌘X
Copy   ⌘C
Paste   ⌘V
The usual Macintosh facilities. The most frequent cutting and pasting will take place between the Session window, the Command line window and the Edit window. You

cannot Cut or Paste the session log, but you can Copy an entire line from it into the Clipboard, and thence into another window.

Bring Down   ⌘B

A fast copy/paste from the Session window. Click on a Session window line, which highlights the entire line. Then choose this menu item, or hit ⌘B, and the line will be copied into the Command line window. NB: only entire lines can be copied from the session log, but once in the Command line or the Edit window they can be cut, copied, pasted or re-typed in whole or in part.

NB: Double-clicking on a line of the Session window will bring it down into the Command line, i.e. it is equivalent to selecting a line and hitting ⌘B. When you get confident with the mouse, this is the normal way you'll use to "Bring Down" a line.

Re-Input   ⌘Y

Like Bring Down, but the entire selected line is immediately re-executed without the opportunity to overtype it. Best used in conjunction with the arrow keys when the Session window is frontmost.

Edit Function...   ⌘=
Edit Char Array...
Edit Num Array...

IAPL/Mac "obtains" the name of an item to edit (see above under Copy Item... to learn what "obtain" means). It then decides what type of item you have chosen and shows you its contents in the Edit window for you to alter. You may then edit the item as if using a typical Macintosh text editor (except that you cannot change font). Then select Finish Edit (⌘E) (see below) and the chosen item will be updated.

The item will NOT be updated in the workspace unless you select Finish Edit. If you simply click the go-away box, or start editing another item, the previous item will simply stay as it originally was in the workspace. If you click the go-away box by accident (or intentionally, to look beneath the window), then just choose TextEdit Window (⌘T). The window will reappear with its contents intact and you can carry on editing as if nothing happened.

If the item exists already, it doesn't matter which of the three you choose: IAPL/Mac will look at the item to decide how best to edit it. However, if the item doesn't yet exist, i.e. the name is an unused name, then you need to tell IAPL/Mac what sort of item to create: function, char array or num array. That's the only reason why you have three menu items, not one.

The way in which IAPL/Mac "obtains" the name it needs allows you to specify the item you want to edit just by pointing to it. Use the menu item: WS Names... to give you a list of variables and functions in the workspace. Then edit any desired item by scrolling the Session Window back to that list, click on a name (to select its Session log line), then hit ⌘=.

If the item you want to inspect or edit appears in a listing, select the whole line. If it is the first item name in the line, then ⌘= will find it correctly (IAPL/Mac will ignore punctuation and special APL symbols). If it is NOT the first item, then Bring Down the line and select (hilite) the name by dragging the cursor across it.

Finish Editing   ⌘E

The edit window disappears and the item being edited is updated. IAPL/Mac remembers what it was you were editing. NB: clicking the go-away box of the Edit window does not update the item being edited. It simply makes the window go away. You can make the window come back again, with its contents intact, by choosing TextEdit Window (⌘T) from the Window menu. The advantage of this is that you can break off editing a function in order to try out an APL expression, then copy the successful expression from the Session log into the function you are editing.

Abort Edit   ⌘-

Allows any of the three edit choices above to be aborted. The Edit window will be cleared, it will disappear and the item being edited will remain unchanged. This is tidier than simply clicking the go-away box of the Edit window.

WS Names...

Displays a list of names of variables in alphabetic order, then functions. This list appears in the Session log and can be referred-to repeatedly and used for selecting items to edit.

WS Settings...

Displays an alterable window showing the current values of all the so-called "quad" system variables.

Control...

Displays a dialog window showing the current list of named expressions which can be input by hitting a single ⌘-{digit} key, ⌘1 to ⌘9, corresponding to the menu items in menu: Control. Individual expressions and their corresponding labels in menu: Control can be altered.

However, for technical reasons, the dialog box requires the use of ASCII only. This means that any APL symbols in the expressions fields are unrecognisable as such. This does not matter if the "Control" expressions are simply function calls. If you wish to cause Control to emit an expression in APL symbols, then you can change the value of a "Control" expression by using a πMC expression in APL. Study the library workspace CONTROL to see how to do this. CONTROL contains functions SETMENU and SETCONTROL which are used as the following example shows to set up the first three menu items:

```
⊂ SETMENU ∪Functions∪
⊂ SETCONTROL ∪)FNS∪
2 SETMENU ∪Variables∪
2 SETCONTROL ∪)VARS∪
∧ SETMENU ∪Vars≠Fns∪
∧ SETCONTROL ∪πNL 2 ∧∪
```

Preferences...

Displays a window giving overall run-time options (figure 13). Use this to alter the behaviour of some of the menu facilities like Find.
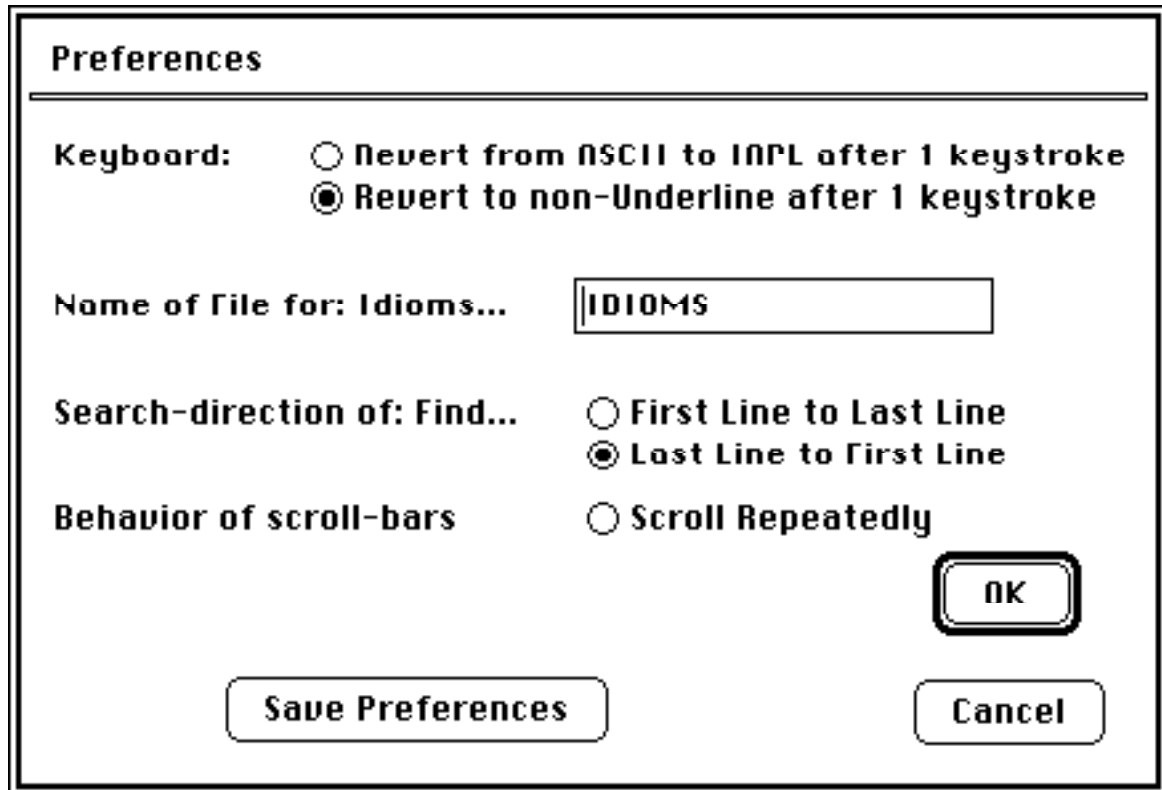
Figure 13. Dialog box which appears when Preferences... is selected.

APL Keyboard   ⌘K

This menu item can be checked and un-checked by selecting it. When checked (the usual mode), the keyboard is in "APL mode", suitable for typing APL program code. If you want ASCII characters (usually mixed case alphabetic characters) hit ⌘K and then type the desired characters.

The keyboard stays in ASCII mode until you hit ⌘K again. If you want to change this behaviour, viz. to make it stay in ASCII mode for only one keystroke, then revert to APL mode, choose Preferences... from this menu and click the appropriate radio button. See figure 13.

Underline A...Z   ⌘U

This menu item can be checked and un-checked by selecting it. When checked, the keyboard is in "Underline mode", suitable for typing uppercase and lowercase underlined characters. If you want an underlined character (whether uppercase or lowercase) hit ⌘U and then type the desired character as if in ASCII mode. See: APL Keyboard above. When the keyboard is set in Underline Mode, it is also automatically set in APL Mode, because underlined characters are strictly APL characters, not ASCII.

Usually you'll want only one underlined character, not a whole string. Thus the keyboard stays in Underline mode for only one keystroke, then it reverts to APL mode. If you want to change this behaviour, viz. to make it stay in Underline mode until you hit ⌘U again, choose Preferences... from this menu and click the appropriate radio button. See figure 13.

Note that underlined characters, e.g. ¡¬…⁄ are a distinct set of characters as far as APL is concerned, having nothing to do with the standard Macintosh "style" for any

given font called <u>Underline</u>. Typically such characters are used in identifiers employed by proprietary packages of APL utilities, to minimise accidental clashes with user-chosen identifiers.

Menu: **Search**

Find...   ⌘F

Allows you to set up a search string and look for it in the Session window or Edit window. This is a conventional Macintosh menu item. It allows lines containing given strings to be located in the session log. IAPL/Mac "obtains" the required search-string in the same way as it does for the facilities Copy Item and Edit Function. See above under these headings to see what "obtains" means.

The typical way of using this facility, e.g. to find the last time you re-assigned the value of variable Z, is to type the desired search string, e.g. z¢ into the Command line, then hit ⌘F. IAPL will search *backwards* through the Session log, (since this is the usual thing you'll need) and will highlight the entire line containing the search string (in the Edit Window it will highlight only the actual string z¢).

If you want to change this behaviour, viz. to make IAPL/Mac search forwards (from beginning to end) instead of backwards, choose Preferences... from the Edit menu and click the appropriate radio button. See figure 13.

Find Again   ⌘A

Finds the next occurrence of the search string without you having to set up the search string again. This is also a conventional Macintosh facililty, frequently invoked by hitting its keyboard equivalent: ⌘A. The searching begins from the currently selected line. When it reaches the top it beeps, then further keystrokes ⌘A resume searching from the end of the Session log again.

If you want to change this behaviour, viz. to make IAPL/Mac search forwards (from beginning to end) instead of backwards, choose Preferences... from the Edit menu and click the appropriate radio button. See figure 13.

Idiom...   ⌘I

Displays the text-only file called IDIOMS in the Session window. This file is identical in nature to a saved Session log. Expressions can be copied and pasted into the Command line window, just as with any other saved and reloaded Session log. The IDIOMS file can be edited by the user (e.g. the tutor, to include his or her own list of idioms) or composed upon a suitable session log. Remember to save the existing Session log if you want to refer back to it, since the IDIOMS file replaces it in the Session Window.

You can change the name, IDIOMS, of the file which is loaded into the Session Window when you select this menu item. Choose Preferences... from the Edit menu and overtype the contents of the appropriate field. See figure 13.

Symbol...   ⌘J

APL symbols are classified in a variety of forms, which can be displayed and "typed" by clicking, once the appropriate symbol has been found. See figures 6 and 7 and the section below on finding and identifying APL symbols. Once you see the symbol you want, click it. The symbol you click gets "typed" for you into the Command line at the cursor position. See figure 8.
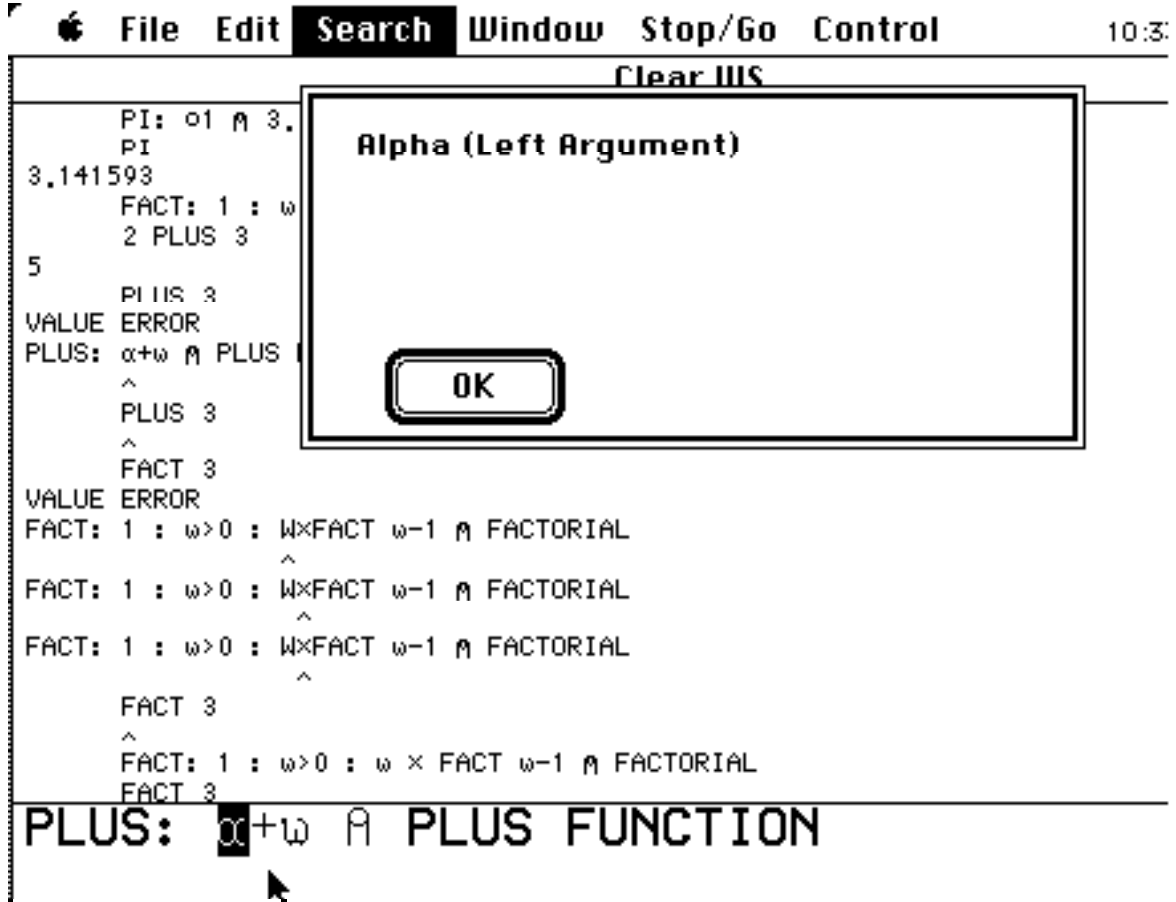
```
 🍎   File   Edit   Search   Window   Stop/Go   Control          10:3:
                                   Clear IIS
     PI: o1 ⍙ 3.
     PI
3.141593
     FACT: 1 : ω          ┌─────────────────────────────────────┐
     2 PLUS 3             │  Alpha (Left Argument)              │
5                         │                                     │
     PLUS 3               │                                     │
VALUE ERROR               │                                     │
PLUS: α+ω ⍙ PLUS          │                                     │
     ^                    │    ┌──────────┐                     │
     PLUS 3               │    │    OK    │                     │
     ^                    │    └──────────┘                     │
     FACT 3               └─────────────────────────────────────┘
VALUE ERROR
FACT: 1 : ω>0 : W×FACT ω-1 ⍙ FACTORIAL
                  ^
FACT: 1 : ω>0 : W×FACT ω-1 ⍙ FACTORIAL
                  ^
FACT: 1 : ω>0 : W×FACT ω-1 ⍙ FACTORIAL
                     ^
     FACT 3
     ^
     FACT: 1 : ω>0 : ω × FACT ω-1 ⍙ FACTORIAL
     FACT 3
```

PLUS: ▮+ω ⍙ PLUS FUNCTION

Figure 14. Dialog box which appears when What Is It? selected.

What Is It?   ⌘/

This causes a dialog window to appear with a message identifying the currently selected symbol. If you have made no selection as such, the symbol immediately *before* the cursor is taken to be the one indicated. If you select more than one symbol, only the first is identified. See figure 14 and the section below on finding and identifying APL symbols.

Mnemonic: '/' is on the same key as '?', but you cannot readily hit ⌘?.

Use this facility to help you read APL expressions in someone else's function definition if you are unfamiliar with APL notation. List the function in the session log, then double-click the line you want to read in order to bring it down into the Command line. Hit {up-arrow} (twice) to bring the cursor to the front. Then repeatedly hit the sequence:

        ⌘/, {return}, {right-arrow}

to see the names of successive characters.

Menu: **Window**

This makes windows re-appear which have been covered by other windows, or been made to disappear by clicking their "go-away boxes".

Help...   ⌘H

Displays a read-only window of on-line assistance information. Click anywhere on the window to see the next "page" of assistance. If you know how to move Macintosh resources between files, it will help you to know that all the pages of Help... are 'PICT' resources with resIDs lying between 300 and 399 inclusive. Simply attaching a

new 'PICT' resource having a resID lying in this range will cause IAPL/Mac to show it in its proper sequence.

Cmnd Line   ⌘M

Makes the Command line window reappear, but does not force IAPL to take note of it. Anything typed into it will be read when IAPL is ready to accept input. Conversely, whenever IAPL expects typed-in input it will cause the Command line window to appear. This item is useful for making the Command line come to the front whenever the Edit window is in use. Otherwise the best way to make the Command Line come forward is to start typing something.

Session Window   ⌘L

This window shows the history of the APL Session so far (see figure 10). Entire lines can be selected for pasting, but its contents cannot normally be altered retrospectively. The window can be emptied by clicking upon it, then choosing Clear Window in the File menu, or by hitting ⌘* (use the * on the numeric keypad if your keyboard has one). Simply clicking on a line of the session log selects it, and also performs an implicit Copy. You can then go right ahead and Paste. Double-clicking a line brings it down into the Command line. It is equivalent to Bring Down above.

TextEdit Window   ⌘T

This is the window used by the full-screen editor for functions and arrays. It allows full Macintosh Cut and Paste facilities, like any Macintosh text editor or word-processor. Normally it is activated by the Edit Function... etc. menu items, so you should not normally need to use this menu item. The library workspace EDITDEMO contains useful functions to access and control the Edit window, e.g. for real and simulated file I/O.

Graphics Window ⌘G

The graphics window normally appears automatically when graphic operations are performed. It can be made to reappear by choosing this menu item if its "go-away box" has been clicked, but any graphics written on it will have disappeared and the window will be blank.

Menu: **Stop/Go**

This menu generates a combination of ESCAPE calls to IAPL and the appropriate stack control command. They have been designed to fit the required task as a beginner might see it, not as APL expects it. The apparatus for suspended functions is a traditional one in APL and is provided because the standard demands it. Experience shows that suspended functions are more of a nuisance than a help to beginners. Fortunately "direct definition" (or "colon") functions cannot be suspended.

Abort   ⌘.

Simulates {esc} and then generates the APL command: £

Novices are advised to use Abort rather than Suspend and not to suspend execution of APL functions deliberately unless they are going to debug them straight away, and to clear out any suspended functions as soon as possible using Clear Stack (see below)

Suspend   ⌘,

Simulates {esc} only. Alternatively you can hit the {esc} key, which keeps its traditional APL function.

Novices are advised to use Abort rather than Suspend, i.e. not to suspend execution of APL functions deliberately unless they are going to debug them straight away, and to clear out any suspended functions as soon as possible using Clear Stack (see below).

Suspend will not escape from the "quad-prompt", i.e. when IAPL asks for numeric input in response to an expression like: Z¢π
Instead you must use Abort (which in this case, and this case only) *suspends* the calling function without aborting it).

Resume   ⌘R
Generates the APL expression: £πLC
This causes the topmost suspended function in the stack to resume execution.

Show Stack
Generates the APL command:)SI
This shows the stack of suspended functions, letting you see the different places in which suspending an expression calling nested functions has caused those functions to halt.

Clear Stack
Generates the APL command:)SIC
This empties the stack of suspended functions. You should do this before saving your workspace to keep things tidy. Otherwise your workspace will be saved, and will reload, with suspended functions waiting to resume execution. Whilst this is useful for debugging purposes, it is wasteful of space if you do not wish to perform any debugging.

Pop Stack
Generates the APL expression: £
This causes the topmost suspended function in the stack to quit.
Menu: **Control**
This menu contains 9 user-alterable menu items, ⌘1...⌘9, each of which can run an APL expression specified by the user. It is provided for the use of teachers to assist novices or others needing to run an IAPL application without typing APL expressions, since it allows any desired expression to be input by means of a menu choice or a corresponding ⌘-{digit} keystroke.

You can alter the contents and function of this menu by choosing Control... in menu: Edit and overtyping fields in the resulting dialog box (see figure 15). Each menu item has a user- (i.e. tutor-) assigned literal name plus a ⌘-digit key, e.g. ⌘1, ⌘2, ...

You can also set up the contents of this menu by means of an IAPL function. See under Control... above.

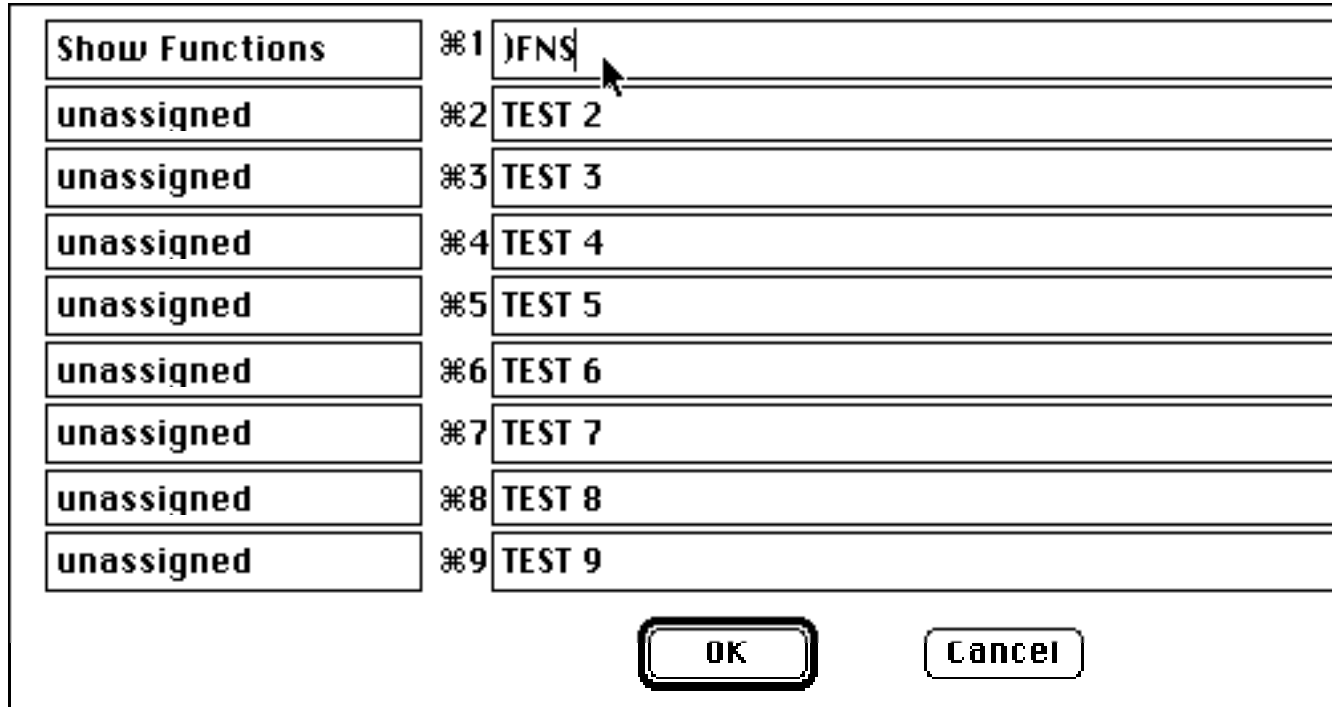| Show Functions	⌘1	)FNS
unassigned	⌘2	TEST 2
unassigned	⌘3	TEST 3
unassigned	⌘4	TEST 4
unassigned	⌘5	TEST 5
unassigned	⌘6	TEST 6
unassigned	⌘7	TEST 7
unassigned	⌘8	TEST 8
unassigned	⌘9	TEST 9

[OK]   [Cancel]

Figure 15. Dialog box which appears when Control... is selected from the Edit Menu.

Importing and exporting workspaces to I-APL on other types of computer

Workspaces run under IAPL/Mac are byte-for-byte compatible with those suitable for any other IAPL system. The πMC interface is, as you would expect, incompatible between different machines.

Workspace files can be imported to the Macintosh in a variety of ways, the Apple File Exchange utility perhaps being the easiest, since this can read 3.5" IBM format MS-DOS disks placed directly in the Mac drives. The resulting files however have the wrong icon (the so-called 'default' icon). They may also have a name like PLANETS.IWS, the .IWS part of which gets in the way and should be removed by renaming the file as PLANETS. Keep all workspace names to 8 characters or less. The Macintosh accepts names longer than 8 characters but IAPL does not.

Such a file will not initially be visible to Load... in the File menu, but it will be visible to Launder..., also in the File menu. Choosing Launder... will show the name of the file in a dialog box resembling figure 9, along with all the others in that folder. Selecti the file and press the Open button. This will cause the file to become to all intents and purposes an IAPL/Mac workspace, provided the filename conforms to the above requirements. A dialog box confirming this change will appear. It doesn't matter if you do it twice to the same file, but be sure to "launder" only files which you know to be IAPL workspaces. Thereafter the file will have the correct icon, be visible in the dialog box of Load... and should load correctly when double-clicked.

If you "launder" a non-workspace file by mistake, the author supplies a utility program to undo the damage at the cost of £12 (UK) or £15 (Overseas) which is inclusive of postage and packing, plus a humiliating sticker to put on your computer (or yourself).

Workspace files may be migrated to IAPL for PC by simply copying them back, appropriately renamed (e.g. from PLANETS to PLANETS.IWS), to an MS-DOS formatted

disk using the Apple File Exchange utility (the author has successfully used

KERMIT, too, in binary transfer mode). This makes IAPL/Mac a handy, productive way of developing IAPL workspaces for the PC.

Exploring an unknown workspace

A typical but forbidding task for the novice is to explore an unknown workspace, browsing the contents of variables and functions, running low-level functions with test-arguments to see what they give, opening the functions and copying out bits of the code to try and understand how it works.

Select WS Names... from the Edit menu. Figure 16 shows the appearance of the screen after doing this.
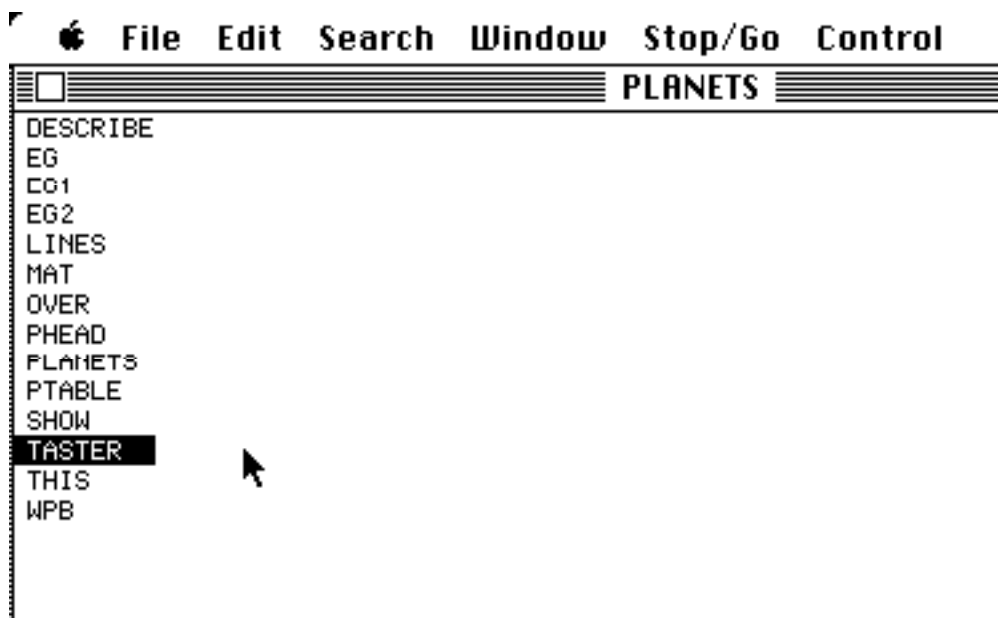
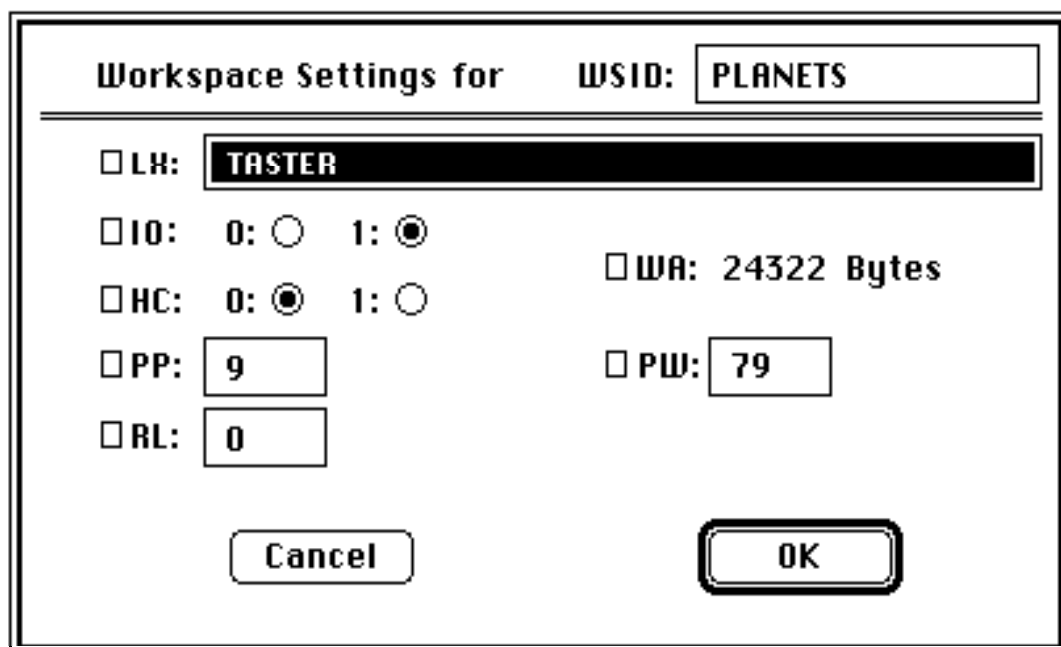Figure 16: The effect of choosing WS Names... from the Edit menu and selecting a line.

Figure 17: The effect of choosing WS Settings... from the Edit menu.

To see the current settings of the "quad"-variables, select WS Settings... from the Edit menu. A dialog box appears as in figure 17 (above). This is the most convenient way for novices to alter their values. These are updated only when you make the window disappear by clicking the OK button. If you click Cancel they all stay as they are.

Some tutors write their own APL utilities for inspecting strange workspaces (e.g. SHOW by the author). These tools sometimes have fixed expressions to run various operations. Instead of the student having to remember these each time, you can asign a name to them and put them in the Command menu. See above under Command... in menu Edit.
Finding and identifying APL symbols

IAPL/Mac closely follows the keyboard layout of IAPL for the IBM PC and compatibles. In APL mode (whenever item APL Keyboard in the Edit menu is checked), unshifted keys give Uppercase alpha and shifted keys give the corresponding APL symbol as assigned by the IAPL PC keyboard layout. Those symbols which need Alt in the PC keyboard layout can be typed by holding down the "Option" key in IAPL/Mac. Otherwise the Option key is not employed. In non-APL mode (whenever item APL Keyboard in the Edit menu is un-checked) ordinary keys have their usual values and uppercase and lowercase alpha can be typed.

APL in general, and IAPL in particular, offers a distinct set of underscored uppercase and lowercase alphabetic characters. These can be typed by selecting Underline A...Z in the Edit menu, or hitting ⌘U, which causes the next alphabetic character to be underlined. The keyboard then reverts after one keystroke to the APL keyboard. The Preferences... item in menu Edit lets you change this so that Underline A...Z stays checked until explicitly turned off (by selecting it again).

One of the most frustrating things for the novice (or expert coming from a different APL system) is to hunt for the special APL symbols on the keyboard. Even a marked keyboard doesn't help much since it is surprisingly difficult to scan any sort of complex layout for symbols which have no standardised ordering. Here is where the Symbol... item in menu Search helps.

Taking a leaf out of Gardiner's *Ancient Egyptian Grammar,* the symbols of APL are classified into: *Rounds, Nails, Arrows, Dots/Punctuation, Maths, Slashes, Barred, Squares, Greeks, Triangles, Angles, Logicals, Twiddles* and *Brackets.* Several symbols fall under more than one of these headings. You will normally find them recurring under each likely heading.

A collection of panels showing these different groupings may be browsed by button-pressing, until the required symbol is located and mouse-clicked. It is then pasted into the command line window at the current cursor position. Figures 6 and 7 illustrate two such panels which appear whenever Symbol... is selected.
Finding and re-inputting APL expressions and idioms
An idiom is a stock APL expression for achieving a given result, modifiable to employ the values in-hand. Novices don't find APL idioms particularly memorable. If you've used an idiom or a particular expression already, it's nice to be able to find it again quickly.

The Session window is scrollable back to the start of a session. If you've done some useful things you can save the contents of this window. You can reload an old session log, or somebody else's, although of course this won't put your workspace in a compatible state. Nevertheless this enables teachers to set up practice drills and expressions for the student to explore, allowing APL to be used as a docile vehicle for whatever topic is to be taught.

The session log window lets you select lines to re-input, but doesn't allow the full Macintosh facilities for editing text. It doesn't make much sense to rewrite history, and anyway you are supposed to use the Command line to edit old expressions. However the Idiom... item in menu Search helps here. It can load an old Session log (of which the IDIOMS file is a special case) into the Session window for you to select lines to re-execute. Alternatively, if you load the old Session log file into the Edit window (by selecting TextEdit Window in menu Windows and then Load Window... in menu File) then you can edit the file at will, saving it back in due course with Save Window.

You can temporarily substitute your own version of the file IDIOMS by selecting Preferences... in menu Edit to change the name by which IAPL/Mac knows the Idiom file.
The Direct Definition Facility
There is a fuller description of the Direct Definition facility in Garry Helzer's book "An Encyclopaedia of APL" available from I-APL Ltd. For those familiar with the concept of direct definition, details are given here about how it is implemented in I-APL.

There are two types of directly defined function in I-APL: simple and conditional.

Simple direct definition

A simple definition may be made in immediate execution mode by entering a name followed by a colon followed by an expression, e.g.
    <name> : <expr>
or:
    <name> : <expr> Ω <comment>
where <name> represents the name of the function being defined and <expr> is any I-APL expression which may optionally be followed by a comment symbol and a comment. When executed, the explicit result of the function is the value of <expr>, if any.

If either or both of the special identifiers † and ˇ appear in the expression outside single quotes, the function is ambivalent, and † and ˇ represent the left and right function arguments respectively. Ambivalent functions may be either monadic or dyadic, according to the arguments provided when it is invoked. Eg, in the simple case, the entry:

   PLUS: † ≠ ˇ Ω PLUS FUNCTION

defines a function called 'PLUS' which implements addition.

The function NULL, below, simply returns its right argument as the first example of its use shows, but if provided with a left argument it will ignore it as the second and third examples show.

   NULL: ˇ

```
   ∧ ≠ ℕULL 7
⊂0
   ∧ ≠ 5 ℕULL 7
⊂0
   ∧ ≠ ∪A∪ ℕULL 7
```

Thus NULL implements the Iverson and Sharp function 'Dex' (see A Dictionary of APL by K. E. Iverson). 'Lev' is equally easy:

```
   LEV: †
```

If neither † nor ˇ appears outside single quotes, the function is niladic. Eg, the entry

```
   PI: ©⊂
```

defines a function 'PI' which returns the value of π.

Conditional direct definition
A conditional definition may be made in immediate execution mode by entering a three-segment expression separated by colons, as follows:

```
   <name> :<expr0> : <cond> : <expr1>
```

When run, the function executes the <cond> expression first. It must evaluate to a boolean value which is a scalar or one-element vector otherwise an error is reported. If it evaluates to 0, the explicit result of the function is the evaluation of the left hand expression <expr0>. If 1, the explicit result is the execution of the right hand expression <expr1>.

For example, a recursive factorial function can be defined by entering:

```
   FACT: ⊂ : ˇ>0 : ˇ ∂ FACT ˇ-⊂
```

A directly defined function may be dynamically established by applying the system function πFX to a suitable character vector.

Editing and displaying directly defined functions
Any extant directly defined function may be displayed or edited by entering, in immediate execution mode, its name followed by a colon:

```
   <name>:
```

The definition is displayed and the cursor positioned at the right hand end of the line, inviting an edit. Pressing {return} redefines the function, signalling an attention aborts the edit.

The definition may also be extracted as a character vector by applying the system function πℂℝ to the name of the function (without the colon).

Invocation
All directly defined functions are either niladic or ambivalent. An ambivalent function can be invoked either monadically or dyadically, and if invoked monadically, the name A has no value. For example:

2 PLUS ∧

5

```
   PLUS ∧
VALUE ERROR
PLUS: †≠˘ Ω PLUS FUNCTION
   ß
   PLUS ∧
   ß

   FACT ∧
×
   2 FACT ∧
×
```

Miscellaneous facts

Any user name or system variable name to the immediate left of an assignment arrow in a directly defined function is localised. If you localise a system variable in this way, e.g. πIO, it starts off with the values of its global namesake.

The name class of a directly defined function is 3. In other words, if 'PLUS' is a directly-defined function:

```
   πNC ∪PLUS∪
∧
```

Directly defined function names are displayed with a colon suffix when they appear in the output from)FNS.

The del (∪≠∪)-editor cannot be used to define, edit or display directly defined functions.

πSTOP and πTRACE do not apply to directly defined functions.

Directly defined functions cannot be suspended. An error in a directly defined function causes the state indicator to be stripped back to where a suspension can occur. If the)SI stack was empty before the error, it is also empty afterwards.

πCR ∪FOO∪ is a vector when applied to a directly defined function 'FOO'.

Editing items in the workspace
Editing functions

There's nothing wrong with using the 'del' ∩≠) editor built-into IAPL. This is the original 1960's standard line-editor, so every APL interpreter has it. You can display a listing of a given function, e.g. MYFUN, by the expression:

```
   ≠MYFUN[π]
```

Then, by selecting lines of the listing of MYFUN from the Session log using Bring Down in menu Edit, overtyping and re-inputting them, you can alter MYFUN with hardly any more effort than with the screen editor. Eventually you finish editing the function by typing the symbol 'del' ∩≠) which is on the 'dollar' key (it's meant to be a mnemonic). This closes the function and IAPL/Mac accepts expressions once more.

However by typing MYFUN into the Command line and choosing Edit Function... you can edit the function by means of IAPL/Mac's screen editor. This allows you free use of the arrow keys, delete, cut, copy and paste, to modify the listing in whatever order you like. When you choose Finish Editing the contents of this window are fed back into IAPL. Note

that there can actually exist global variables ⍺ and ⍵, although these "variable names" are normally used only for function arguments.

However IAPL/Mac puts its own values in them, destroying any values they may already have. So don't store values in global † and ˇ which you want to keep.

If the function MYFUN happens to be a directly-defined (or "colon") function then you do not get the Edit window, but the Command line window, filled with the function definition, just as if you had double-clicked the function definition line in the Session log. By overtyping the contents of this, you simply re-input the entire function definition.

Editing character arrays
These are edited just like functions. Simply type in the name of the character array variable and select Edit Char Array... (or hit ⌘=). It doesn't matter if you select Edit Function... by mistake, IAPL/Mac will discover that it's really a variable and edit it accordingly. Incidentally, scalar or 1-dimensional character vectors will be converted to 2-dimensional arrays, so remember to ravel it afterwards (e.g. Z¢,Z) if you really find it valuable to use the screen editor on a scalar or vector.

Edit Char Array... will create the variable for you if it does not yet exist. Just edit it as if it were there. The empty Edit window will appear. Be sure to use Edit Char Array... this time and not Edit Function... (or ⌘=) or else the new item will be created as a function.

Editing character arrays is slightly faster if you do not change the line length or the number of lines. However, if you do, IAPL/Mac will square-off a ragged right edge. The width of the edited variable becomes that of the longest line. So be wary of leaving a lot of trailing blanks on the end of a line.

Editing numeric arrays
These are edited just like functions. Simply type in the name of the numeric array variable and select Edit Num Array... (or hit ⌘=). It doesn't matter if you select Edit Function... by mistake, IAPL/Mac will discover that it's really a numeric variable and edit it accordingly. Incidentally, scalar numbers will get converted into 1-vectors, and 1-dimensional numeric vectors will be converted to 2-dimensional arrays, so remember to ravel it afterwards (e.g. N¢,N) if you really find it valuable to use the screen editor on a scalar or vector.

Numeric variables get converted into character arrays and back to numeric arrays again. If you type a non-numeric character, the conversion process will fail and your variable may be corrupted. However usually it will remain unchanged from what it was. Incidentally, if the process does fail, ˇ will contain a ravelled form of the edited numeric variable, and † will contain the shape IAPL/Mac was trying to give it.

Editing numeric arrays is slightly faster if you do not change the number of elements, nor the shape of the array. If you do, IAPL/Mac recomputes a new shape based on the number of rows and the total number of elements, which it will pad with zeros until the number of rows divides it without remainder. Unlike Edit Char Array... it does not pad every row of the array to the width of the widest row. Thus if two rows contain different numbers of elements, then elements may spill from one row to another.

Edit Num Array... will create the variable for you if it does not yet exist. Just edit it as if it were there. The empty Edit window will appear. Be sure to use Edit Num Array... this time and not Edit Function... (or ⌘=) or Edit Char Array... or else the new item will be created as a function, or a character array.

Using Text-Only files

Any text-only file, even a LightspeedC listing, can be loaded into either the Session window or the Edit window, depending which is to the front as you select Load Window... from the File menu. Conversely if you save the contents of either the Session window or the Edit window by using Save Window or Save Window As... you create a text-only file, which is capable of being opened by the Apple Computer Inc. utility: Edit, or by LightspeedC, or by most word-processors or program development systems. If you do so, you will find that these text-editors show the file in a default font, which may be New York, Geneva or Monaco. Unless you are using ASCII characters only, the result will look like gibberish until you select a piece of text and change the font to St Albans.

Any text-only file created or saved back by IAPL/Mac carries the icon: which simply serves to denote "ownership" by IAPL/Mac and makes it double-clickable.

Such a file can also be moved to- and fro- between a Macintosh and an IBM-compatible PC running under MS-DOS, e.g. by using the Apple Computer Inc. utility: Apple File Exchange. Use the Text File conversion to do so, since MS-DOS and Apple have slightly different text-only file conventions. The difference is that Apple files terminate each line of the file by Return, i.e. the byte ASCII 13. MS-DOS files use a pair of bytes: Return,Line-Feed, i.e. the bytes ASCII 13, ASCII 10. MS-DOS text-only files are also apt to be terminated by EOF, i.e. the byte ASCII 26 (also known as {ctl-Z}).

You can manage text-only files by means of a collection of IAPL functions in the library workspace EDITDEMO. These allow you to write APL functions to do the following:
  — load a named text-only file into the Edit window
  — save the Edit window as a file of a given name (expressed as a character vector)
  — fetch the contents of the Edit window in the form of a 2D character matrix
  — place a 2D character array value into the Edit window
  — fetch line N of the Edit window in the form of a character vector
  — place a character vector into line N of the Edit window
  — show or hide the Edit window
  — get or set the selected text or the "i-beam" cursor position in the Edit window.
These facilities provide what is, in effect, text-file I/O using the Edit Window as a visible buffer.

Embedding samples of APL sessions in a desk-top published document

If you cut/copy a piece of APL text, wherever from, its corresponding ASCII codes (coded according to πAV) go into the so-called Clipboard. Unfortunately this will not preserve the font information, so if you now paste the clipboard contents (a standard Mac idiom) into a word-processed document you will see ASCII text in the default font, perhaps Geneva or New York.

Never despair. Just select the text you've just embedded and change it into font: St Albans and you will see recognisable APL text reappear.

St Albans exists both in 9 and 18-point sizes. Installing both fonts in your System File will allow you to print documents containing IAPL text on the ImageWriter in "Best" quality, since the printer driver will "scale the 18-point font down to 9-point in order to get the double-density of pixels that "Best" asks for.

In LaserWriter-printed documents, the quality of St Albans font as derived from its bitmap

image can be gauged from its use in this document.

Chapter 5: Extending IAPL/Mac with machine code and other resources
Writing Machine Code Resources to call from IAPL/Mac
All machine code on the Apple Macintosh is held in the form of resources. In a conventional application these resources are of type 'CODE', but many compilers allow you to generate code-resources having any desired type-code. This facility is used with HyperCard to create add-in commands and functions of type 'XCMD' and 'XFCN'. A comparable technique is used to attach code-resources of type 'APCC' to IAPL/Mac.

If the following APL expression is executed:

    ⊂2∧45 πMℂ VARℕAMES

IAPL/Mac looks for an 'APCC' resource attached to itself, having the Resource ID (ResID): 12345. If it finds it, it loads it into memory and branches to its entry-point. If there is no such resource, IAPL/Mac responds with an error message. If you receive a ready-written 'APCC' resource to attach to IAPL/Mac you need to know its ResID because this is the number by which you call it using πMℂ.

This machine-code facility differs from that provided in, say, the PC version of I-APL, inasmuch as the left-hand argument is the Resource ID of the add-in machine-code segment, given as a scalar integer. In the PC version, the left-hand argument of MC is an actual string of code-bytes in the form of a scalar integer or vector integers.

ResIDs 128 to 32767 are available for add-on resources. ResIDs 0 to 127 are reserved by I-APL Ltd to allow built-in facilities to be invoked via the πMℂ interface. ResID 11519 is reserved for future emulation of the built-in machine-code facility of IAPL for the PC. (The binary representation of 11519 just happens to be a PC machine-code instruction which branches to the required internal routines.)

Appendix C contains the source-code of a sample 'APCC' written in C for the LightspeedC or THINK C environments.

The built-in Machine Code Functions
There are a number of built-in machine code functions. They are called just like an attached 'APCC' code resource, but they all have ID numbers less than 128. Calling them is slightly tricky, so normally you will invoke them by means of cover-functions in the supplied workspaces. However, if you want to call them directly, you should consult the table below, using one of the expressions:

(i):   m πMℂ ∪˘∪
(ii):   m πMℂ ⊂ 2 º∪†˘∪
(iii):   m πMℂ ⊂ ⊂ ⊂ º∪˘∪
(iv):   m πMℂ ⊂ ⊂ 2 º∪†˘∪

The second and third columns are headed † and ˘. If there is a dash (-) in the †-column, use (i) above, else use (ii). An entry in the †-column (say) denotes the type of variable expected as †. See the **Key** at the bottom. If there are no asterisks (*) against the type-entries, you may optionally use (iii) instead of (i), or (iv) instead of (ii).

You must pass a table of *variable names* to πMℂ as the right-hand argument. You cannot use

numbers. Thus the following is wrong and will fail:

    m πMℂ ∪∧∪

The table of built-in quadMC functions is as follows:

m	†	ˇ	Description
10	-	N	specifies the target window of 11 to 20 below (N specifies window: 0=Command line, 1=Session, 3=Edit, 4=Work)
11	-	S	moves given string vector S into Target window
12	-	SS	moves given 2D ch-array SS into Target window
13	-	SS*	moves Target window contents into given char array SS (SS must be initialised to be an array of spaces of the correct shape, ℝºυ υ where ℝ[⊂] is the number of rows and ℝ[2] is the width of the widest row in the Target window. See 15 below)
14	N	S	moves S into Target Window to replace line N
15	-	N2*	sets N2 to be the required shape of Target window contents (mainly used in conjunction with 13)
16	-	N*	sets N to be the line-length of given line in Target window (mainly used in conjunction with 17)
17	N	S*	moves given line N of Target window into S
18	N	S	moves S into Target Window to become (precede) line N
19	-	S	moves contents of named file S into Target window
20	-	S	moves Target window into named file S
21	-	N	shows window having ID: N (ID for window: 0=Command line, 1=Session, 2=Help, 3=Edit, 4=Work, 5=Graphics)
22	-	N	hides window having ID: N (ID for window: see 21 above)
23	-	N*	waits for key struck, sets N to be the key code
24	-	N2*	waits for mouseDown, gets back coords in N2
25	-	N2*	sets N2 to (row,column) of start of selection in Edit Window
26	-	N*	sets N to width of selection in Edit Window
27	-	S*	sets S to the string value of selection in Edit Window
30	N	X	graph-drawing package, op-code N (Type of X depends on the value of N)
31	-	N	draws (Graphics window) picture having 'PICT' resource ID: N
32	N4*	N	sets N4 to the enclosing rectangle of picture having 'PICT' resource ID: N
33	N4	N	draws picture having 'PICT' resource ID: N in rectangle N4.
40	-	N/ NN	plays sound having 'snd ' resource ID: N, or vector NN of resource IDs.
50	-	N2	emulates a menu selection (N2[1] is menu number, N2[2] is item number)
51	-	N	sets hourglass cursor (N=0 thru 7)
60	N	S	sets Control menu expression N to string S (N=0 thru 9)
61	N	S	sets Control menu label N to string S (N=0 thru 9)

Key: S=char vector (1D), SS=char matrix (2D), X=unspecified here, N=scalar integer, NN=integer vector (1D), N2=integer 2-vector, N4=integer 4-vector.
If starred, e.g. N*, then the variable, e.g. N, is altered. Use expressions (i) or (ii) only.
Otherwise the variable is not altered. Expressions (iii) or (iv) may optionally be used

instead.

If the † column contains dash (-) then the function accepts the name of only one variable. In that case, either expressions (i) or (iii) must be used to call πMℂ.

Note that you can, in principle, use the names of other variables besides † and ˇ. You can also pass the names of up to 20 variables. But the easiest way to handle πMℂ calls is by means of a directly-defined or "colon" function, as illustrated by the library workspaces GRAFDEMO, CONTROL and EDITDEMO. However, do remember that ˇ must appear in the direct definition itself in order to make the function monadic (requiring one argument only), otherwise it will be niladic (requiring no arguments). If † appears, whether or not ˇ also appears, the function will become ambivalent (i.e. accepting both left and right arguments). Now the problem is that ∪ˇ∪ does not constitute an appearance of ˇ. In the eyes of APL, ∪ˇ∪ is only a string constant! So some finesse is necessary, for example:

        MYℂALL: m πMℂ ∪ˇ∪,0ºˇ

        LEV:†
        MYℂALL2: m πMℂ ⊂ 2 º∪†ˇ∪ LEV †

in order to "mention" † or ˇ without actually needing to use it.
Inspecting or Removing a Resource already attached to IAPL/Mac
There are a number of utilities, usually freeware applications, for copying a resource from one file to another. The most general way is to use the Apple Computer Inc. resource-editor, ResEdit. Any sort of resource can be attached to the IAPL/Mac application, but it is no use doing so unless it is a type which IAPL/Mac, or an attached machine-code resource, makes use of.

When you open IAPL/Mac with ResEdit, you will see a list of the following resource types (call it List A):

        'ALRT' 'APCC' 'APKB' 'BNDL' 'CURS' 'DITL' 'DLOG' 'FOND' 'FONT'
        'FREF' 'IAPL' 'ICN#' 'MENU' 'PICT' 'snd ' 'STR ' 'WIND'

All resource types are strings of four letters. 'STR ' and 'snd ' in fact have a trailing blank which is mandatory. The main types of interest to you, if you wish to customise IAPL/Mac yourself, are:

        'APCC' 'APKB' 'PICT' 'snd ' 'STR'

('APKB' is the keyboard layout) If you double-click on one of these names in list A, say 'PICT', a further window will appear (call it List B), showing what resources of type 'PICT' are present already. A line of the window stands for the resource, giving its name (if it has one) and its ResID.

Often you may wish to alter the ResID of some resource (but not of the ones that IAPL/Mac uses internally!). Here's how to do it for any resource type:

Choose a number between 1 and 32767 (ideally greater than 128) which does not appear in the window already (you cannot usefully have two resources of the same ResID number unless they are of different types). Select Get Info ⌘I from the File menu. A dialog window appears. The very first field on it (hilighted) is the ResID, which you overtype.

You may also wish to delete a resource, e.g. in order to make IAPL/Mac take up less room on disk (Resources take up no space in memory until they are called-for, then they can be erased from memory if the space is needed). IAPL comes with certain "demo" resources which are

not generally needed, e.g. the 'PICT' resources of the USA Map and World Map, also all the 'snd ' resources. Once you have List B, which

shows a line for each resource, click on the line belonging to the resource you want to get rid of. This highlights the line. The choose Clear from the Edit menu.

Adding pictures to your IAPL/Mac application

As well as using Clear on a selected resource, you can also Cut, Copy and Paste resources between different files. In particular you can open the Scrapbook File (the contents of your Scrapbook) and Copy a 'PICT' resource, then open IAPL/Mac (it is the file called Version *.*, e.g. 1.0) and Paste the 'PICT' resource into it. Notice you need only have List A showing in order to do this. Then you must change the ResID of the 'PICT' you have just Pasted, because it has a silly number like -32767. Change it to something sensible, like 1234.

If you attach a resource of type 'PICT' (the standard resource name for a "picture" resource as it is stored inside the standard System file called Scrapbook File) you can draw this picture in the Graphics window by means of either of the functions DRAWPICT or INDRAWPICT in the library workspace GRAFDEMO. Both functions take the ResID of the 'PICT' resource as the right argument. INDRAWPICT takes a left argument consisting of the enclosing rectangle to be used. DRAWPICT takes no left argument, but instead uses the "natural" enclosing rectangle (the one the 'PICT' resource was created with).

If the ResID of the 'PICT' resource you attach lies between 300 and 399, then it will be treated as one of the HELP pages, which are shown in their due numerical sequence whenever Help... is selected or ⌘H is hit.. Conversely you can use DRAWPICT to display a HELP page in the Graphics window if you know its ResID.

Adding sounds to your IAPL/Mac application

If you attach a resource of type 'snd ' (the standard resource name for a sound resource) you can play this resource by means of the PLAY function in the library workspace VOICE. This is something like the Hypercard facility, except that the sound is known by its resource identifier (ResID, an integer between 0 and 32767) and not by its resource name.

If N is the ResID of the 'snd ' resource, PLAY N will play its sound. N=1, 2, 3 or 4 plays the built-in system sounds which can be specified as variations on the plain warning "beep" of the Macintosh ("SysBeep"), viz. "Peep", "Clink-Klunk", "Boing" and "Monkey". N=10000+J for J=0 to 9 plays a recorded voice speaking the digit J.

Digitised sound is expensive in Macintosh storage space, but in IAPL/Mac sounds are not stored in the workspace itself. The waveforms are added to the Application file's "resource fork" at need. As far as IAPL is concerned they can be as large as your machine allows. If you don't want them and think they take up too much space on disk, then remove them using ResEdit as explained above.

Translating IAPL/Mac to another European Language

All English text (apart from words like)LOAD and)SAVE which are embedded in the IAPL nucleus itself) are contained in the resources of IAPL/Mac. They can therefore be altered using ResEdit, e.g. to translate them into another (Macintosh European alphabet) language.

If you want to try translating IAPL/Mac into a "National Language" (called "localizing" in MacSpeak) then these are the resource-types you must examine and translate:

'DITL', 'MENU', 'STR '.

In most cases it is simply a matter of double-clicking on each resource in turn, and overtyping the English text you see.

APPENDIX A
IAPL/Mac Standard-Conformance Document

This Appendix describes facilities the I-APL interpreter provides, and documents these facilities to the degree required for a conforming implementation by the Draft International Standard for the APL programming language, document ISO/DIS 8485.

At the time of writing the APL standard has not yet achieved the status of International Standard, but is still at the draft stage. The technical work on the document has been completed however, and no further changes to the draft are envisaged.

The term 'I-APL' refers to a family of APL interpreters designed for small computers. It is intended that all versions of I-APL conform to the standard, but it is possible that a specific port may not exactly conform in all details. This standard conformance document therefore only applies to IAPL/Mac for the Apple Macintosh, for English language users running on a black-and-white screen.

For the rest of this document the term 'IAPL/Mac' is used only where there is an expectation that another version of I-APL may differ in behaviour.

A conforming implementation of the APL language is required to provide all the defined facilities and implementation defined facilities described in the standard, exactly as specified therein. The I-APL implementation is designed to meet this requirement.

A conforming implementation may provide optional facilities. If provided, each such facility must behave as specified in the standard. The I-APL implementation provides exactly one such optional facility (TRACE AND STOP).

A conforming implementation may provide consistent extensions. The presence of consistent extensions is not permitted to affect the behaviour of a conforming program (ie a program that does not use them).

The I-APL implementation includes a number of 'minor' consistent extensions, none of which affects the behaviour of conforming programs.

A conforming implementation is required to use algorithms that produce the same results as those produced by the evaluation sequences presented in the standard. The I-APL implementation is designed to meet this requirement.

A conforming implementation of APL is required to produce documentation of its optional facilities, its implementation defined facilities and its consistent extensions. The following sub-sections contain this material for the I-APL implementation.
A-1 Optional Facilities
The status of each optional facility is as follows:

Shared Variable Protocol	*- absent*
Statement Separator Facility	*- absent*
Trace and Stop Control	*- present*

A-2 Implementation Defined Facilities

A-2.1 The Character Set

The character set consists of 256 unique characters, including all of the characters in the required character set. The following table shows the distribution of the characters in the I-APL atomic vector.

Dec:	0	⊂	2	∧	4	5	×	7	8	⊂	⊂0	⊂⊂	⊂2	⊂∧	⊂4	⊂5
Hex2:	*0	*⊂	*2	*∧	*4	*5	*×	*7	*8	*⊂	*A	*B	*ℂ	*D	*E	*F
Hex1 Dec.																
0* 0	ó	ò	ô	ö	õ	ú	ù	û	ò	ô	ö	õ	ú	nl.	û	ü
⊂* ⊂×	ó	ò	ô	ö	õ	ú	ù	û	ò	ô	ö	õ	ú	ù	û	ü
2* ∧2	sp.	!	∪	#	$	%	&	∪	∩	)	*	≠	,	-	.	/
∧* 48	0	⊂	2	∧	4	5	×	7	8	⊂	:	;	<	=	>	?
4* ×4	@	A	B	ℂ	D	E	F	Γ	H	I	J	K	L	M	ℕ	O
5* 80	P	Q	ℝ	S	T	U	V	W	X	Y	ℤ	[	\	]	^	_
×* ⊂×	`	a	b	ç	δ	e	f	g	h	i	j	k	l	m	n	o
7* ⊂⊂2	ρ	q	r	s	t	u	∨	w	x	y	z	{	|	}	~	
8* ⊂28	Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
⊂* ⊂44	ê	ë	í	ì	î	ï	ñ									
A* ⊂×0	†	°	¢	£	§	•	¶	ß	®	©	™	´	¨	≠	Æ	Ø
B* ⊂7×	∞	±	≤	≥	¥	µ	∂	∑	∏	π	∫	ª	º	Ω	æ	ø
ℂ* ⊂⊂2	¿	¡	¬	√	ƒ	≈	Δ	«	»	…		À	Ã	Õ	Œ	œ
D* 208	–	—	"	"	'	'	÷	◊	ÿ	Ÿ	/	€	‹	›	fi	fl
E* 224	‡	·	,	„	‰		Â	Ê	Á	Ë	È	Í	Î	Ï	Ì	Ó	Ô
F* 240	⌷	Ò	Ú	Û	Ù	ı	ˆ	˜	¯	˘	˙	°	¸	˝	˛	

A character's decimal number is obtained by adding the number above its column to the decimal number of its row. The letter ˙ is thus 10+240=250.

A character's zero-origin index in πAV is also given by the expression:

⊂× ⊂×€∩-πIO)≠∪0⊂2∧45×78⊂ABℂDEF∪≥ℝOW,COLUMℕ

where ℝOW and COLUMℕ are the (hex) row and column labels in the above table.

Unused entries in the above table are not assigned any symbol or meaning, and are displayed as squish-quads (π).

A-2.2 The Numbers

There are three distinct representations for the numbers: 1-bit booleans, 2-byte integers and 6-byte floating-point. A boolean number is represented by a single bit with value 0 or 1 only. Integers are represented by 16 bits in 2s complement format, yielding a range of -32768 to 32767 inclusive. The floating-point numbers are held in a Binary Coded Decimal representation consisting of a normalised 10 digit fraction, a 7-bit binary biased exponent and a 1-bit sign, with an implied decimal point to the left of the fraction. The floating-point number zero is represented by a zero fraction, zero biased exponent and zero sign bit.

A-2.3 Implementation Algorithms

An Implementation Algorithm is an algorithm used in the standard whose behaviour is implementation defined. A description of the characteristics of each implementation algorithm is required for a conforming implementation. The polynomials for the transcendental functions were obtained from "Approximations for Digital Computers" by Cecil Hastings Jr (Princeton University Press). The matrix divide algorithm is based on one kindly provided by I P Sharp Associates.

The implementation algorithms for the transcendental functions provide an accuracy of better than seven significant digits. The other arithmetic algorithms yield an accuracy of ten significant digits. The results are rounded by using a single guard digit. Where appropriate a fixed iteration Newton-Raphson square-root algorithm is used, accurate to nine significant digits.

A-2.3.1 Cosine

Calculated as ⊂ © ⊂.5707⊂×∧27-fl˘

A-2.3.2 Current Time

A nonnegative integer less than 32768 representing a number of seconds. This limits the right argument of πDL to less than 32768.

A-2.3.3 Deal

If 0<˘μ∧27×7 then πℝL¢next πℝL is πIO≠æ˘∂πℝL∞∧⊂⊂04

If ˘>∧27×7 then it is πIO≠æ˘∂∩⊂0000€∩πIO)≠?2º⊂0000)∞⊂E8

A-2.3.4 Display

The argument to Display is an APL array or an exception.

If the argument is an array then it is displayed as recommended in the Standard, except that an exponent field width of 3 is always employed where the output format is decimal-exponential.

If it is an exception then an error message line is displayed, followed by pairs of lines, each pair comprising a code line and a caret line.

The error message line will include one of the following error messages (these may be subject to translation in other I-APL ports).

AXIS ERROR	INTERRUPT	NOT SAVED
DEFN ERROR	LENGTH ERROR	RANK ERROR
DISK ERROR	LIMIT ERROR	SYNTAX ERROR
DOMAIN ERROR	NOT COPIED	VALUE ERROR
INCORRECT COMMAND	NOT ERASED	WS FULL
INDEX ERROR	NOT FOUND	

A code line includes the APL expression that was executing when the execution was signalled. A caret line contains a caret which indicates the approximate point at which execution was halted.

A code line may take one of three forms. These forms are as follows:

    Immediate/quad-input/execute: lin*e*
    Defined function execution: fnnam*e*[ln] lin*e*
    Direct function execution: fnnam*e*: lin*e*

Key: line   = any line
     fnname   = function name
     ln   = function line number

The last code/caret line pair will always display an immediate execution line, a quad-input line, or a function line. Any additional code/caret line pairs will always display an execute line or a direct definition function line.

If WS FULL or INTERRUPT is signalled during the lexical analysis phase of the immediate execution and quad-input modes then the caret line is not displayed.

The display produced by this algorithm is subject to the current value of the system variable πPW so that wide lines are folded, and each continuation line is indented by six spaces.

A-2.3.5 Divided by

Implements division. Division by zero signals a domain- error, overflow signals a domain-error and underflow returns a zero result.

A-2.3.6 Exponential

A polynomial is used to provide an accuracy of better than seven significant digits.

A-2.3.7 Function Display

The following forms are used to display traditional function lines:

    ≠ header
[ln] label:line
[ln] Ω any
[ln] line
    ≠

Key: header   = function header line
     ln   = function line number
     label   = label name
     line   = any line (including empty line)
     any   = any string of characters.

A-2.3.8 Gamma Function

For non-negative integer arguments multiplication is used. For other valid arguments a polynomial, followed by multiplication and/or division, is used to yield a result accurate to better than seven significant digits.

A-2.3.9 Hyperbolic Cosine

    .5∂∩*ˇ)≠∞*ˇ

A-2.3.10 Hyperbolic Sine

    .5∂∩∂ˇ)∂∩*flˇ)-∞*flˇ

A-2.3.11 Hyperbolic Tangent

    ∩∂ˇ)∂∩∏⊂≠*fl2∂ˇ)∞∞⊂≠*fl2∂ˇ

A-2.3.12 Inverse Cosine

    ⊂.5707⊂×∧27-∏⊂©ˇ

A-2.3.13 Inverse Sine

    If ⊂∑flˇ then ∩∂ˇ)∂∏∧©flˇ∞0©ˇ else ⊂.5707⊂×∧27

A-2.3.14 Inverse Tangent

A polynomial is used to provide an accuracy of greater than seven digits.

A-2.3.15 Inverse Hyperbolic Cosine

If Cμˇ<5E×2 _then_ ˇ≠∏4©ˇ
else if ˇ≤5E×2 _then_ ∩´2)≠ˇ

A-2.3.16 Hyperbolic Sine

If 5E×2≤flˇ _then_ ∩∂ˇ)∂´∩flˇ)≠4©ˇ
else if 5E×2μˇ _then_ ∩∂ˇ)∂∩´2)∂´flˇ

A-2.3.17 Inverse Hyperbolic Tangent

.5∂∩∂ˇ)∂´∩C≠flˇ)∞C-flˇ

A-2.3.18 Matrix Divide

Uses the Householder transformation. Based on an APL model supplied by I P Sharp Associates.

A-2.3.19 Minus

Implements subtraction. Underflow is ignored. Overflow signals a domain-error.

A-2.3.20 Modulo

In general P Modulo Q is equivalent to the APL expression P-Q∂æP∞Q executed with comparison tolerance set to zero, except when overflow or underflow would occur. When P∞Q would give an overflow the result is zero, and when an underflow the result is P.

A-2.3.21 Natural Logarithm

A polynomial is used to provide an accuracy of better than seven significant digits.

A-2.3.22 Next Definition Line

The following pseudo-APL algorithm calculates the Next Definition Line. Comparison tolerance is ZERO and N is the current definition line number:

If ℕ=7C _or_ ℕ=7C.C _or_ ℕ=7C.CC _then_ ℕ
else ℕ≠C0*-≠/0∑C .cflℕ

A-2.3.23 Numeric Input Conversion

This algorithm converts a list of characters representing a number in the decimal notation into a number. The result is either a number or an error. Numbers are rounded after the tenth significant digit, eg the input CCCCCCCCCC gives the number 1E12 internally.

A-2.3.24 Numeric Output Conversion

This algorithm converts a number into a list of characters that represent the number in the decimal notation. The algorithm suggested in the standard in used. The exponent field width is three.

A-2.3.25 Pi Times

Multiplies by 3.141592654

A-2.3.26 Plus

Implements addition. Underflow is ignored and overflow signals a domain-error.

A-2.3.27 Pseudo Random Number Generator

The following line indicates the algorithm used to calculate the next value of the random link.

π ℝL¢∧C C04fl×57C≠×25∂π ℝL

A-2.3.28 Read Keyboard

In IAPL/Mac an input line is buffered in a window called the Command line. Therefore characters can be typed-in at any time. As soon as a (typeable) character is hit, the Command line appears and becomes the active window. The cursor-control keys and mouse have their usual function. Thus the "i-beam" pointer places the character cursor wherever required in a string of characters, and{delete} deletes the character preceding the cursor. The Edit menu items (e.g. Cut, Copy and Paste) also function with the Command line active.

A-2.3.29 Sine

A polynomial is used to provide an accuracy of better than seven significant digits.

A-2.3.30 Tangent

∩ð˘)ð SIℕE ∞0© SIℕE ¢⊂©fl˘

A-2.3.31 Times

Implements multiplication. Underflow is ignored and overflow signals a domain-error.

A-2.3.32 Time Stamp

The result of Time Stamp depends on the accuracy of the system clock. On machines which do not have a system clock the result is 7ᵒ0.

A-2.3.33 To the Power

For integer right arguments repeated multiplication is used to calculate the result, except for ˘=.5 or ˘=∏.5 when square root is used. Otherwise the result is calculated by the APL expression:

*˘ə′†

A-2.3.34 Trace Display

The form of the Trace Display depends on the result of executing the line, as follows:

Result	Display
value or committed value	fnname[ln] d
value or committed value	fnname[ln] h
branch	fnname[ln] £n
nil	fnname[ln]
escape	No display

Key: fnname   = function name
     ln   = function line number
     d   = scalar or vector array
     h   = matrix or higher rank array
     n   = branch target line number.

A-2.4 Implementation Parameters

Implementation Parameters are quantities referred to in the standard whose values are implementation defined. The value of each implementation parameter is now presented.

Atomic Vector……………………………….	see above
Initial Comparison Tolerance ……………….	⊂E∏7
Initial Index Origin …………………………	⊂
Initial Latent Expression……………………	∪∪
Initial Print Precision………………………	7
Initial Random Link…………………………	0
Clear Workspace Identifier…………………	∪CLEAℝ WS∪
Positive Number Limit…………………….	⊂.⊂⊂⊂⊂⊂⊂⊂⊂E×2
Negative Number Limit…………………….	∏⊂.⊂⊂⊂⊂⊂⊂⊂⊂⊂E×2
Positive Counting Number Limit…………….	⊂⊂⊂⊂⊂⊂⊂⊂⊂
Negative Counting Number Limit…………	∏⊂⊂⊂⊂⊂⊂⊂⊂⊂⊂
Index Limit………………………………….	∧27×7
Count Limit………………………………….	∧27××
Rank Limit………………………………….	7
Workspace Name Length Limit…………….	8
Identifier Length Limit…………………….	∧2
Quote Quad Output Limit……………………	value of πPW
Comparison Tolerance Limit……………….	⊂E∏5
Integer Tolerance………………………….	⊂E∏7
Full Print Precision……………………….	⊂0
Print Precision Limit………………………	⊂0
Exponent Field Width………………………	∧
Session Identification Type…………………	character
User Identification Type……………………	character
Indent Prompt………………………………	×º∪ ∪
Quad Prompt……………………………….	∪π:∪,newline,×º∪ ∪
Function Definition Prompt…………………	∪[∪,line,∪]∪
Line Limit………………………………….	80
Definition Line Limit………………………	7⊂.⊂⊂
General Offer……………………………….	Not Applicable

Any action that would cause a limit specified by an implementation parameter to be exceeded will signal a limit- error.

A-2.5 Internal Value Sets

The system parameter which underlies each system variable may only be assigned values which belong to the internal value set for that system parameter. For example the internal value set for πLX is all character vectors; you can write πLX¢∪A∪ because scalar ∪A∪ is coercible to a vector of length 1. After such an assignment ºπLX will return a 1 which shows that the scalar ∪A∪ has indeed been coerced to a vector. Any attempt to assign a value not in the set will signal a limit-error.

The internal value sets are:

Comparison Tolerance:	all nonnegative numbers not greater than ⊂E∏5
Random Link:	all integers from 0 to 31103 inclusive
Print Precision:	all integers from 1 to 10 inclusive
Index Origin:	the integers 0 and 1

Latent Expression: all character vectors
Print Width: 19 to 254 inclusive.

A-3 Consistent Extensions

A consistent extension is any facility not specified in the ISO APL standard that, for a construct the standard specifies as producing an error, gives some effect other than signalling the specified error. This section documents the consistent extensions in I-APL.

Note that the use of any consistent extension in an I-APL program prevents that program from conforming to the standard.

In what follows, related consistent extensions have been grouped under a common heading. Every consistent extension is given a unique "CE" code for ease of reference. The code is followed by a short description of the extension, followed by the error that is replaced, in parentheses. After this line a fuller description of the extension is given.

Each example is executed in a freshly cleared I-APL workspace.

A-3.1 Direct Definition

CE1: Immediate execution direct function definition (syntax- error)
The following form is permitted in immediate execution mode:

>>---simple-identifier---b---colon---b---line--->>

A syntax error is replaced with a behaviour which defines or allows editing of a directly defined function whose name is simple-identifier.

If "line" is empty, the current definition is displayed on a new line and the keyboard unlocked for editing.

If "line" is non-empty, the function definition is determined by the content of line. The letter b represents any number of blanks. A definition-error is signalled if simple-identifier cannot be defined as a function.

Example:
```
  ASK: ∩oˇ)ºa,0ºa¢ˇ
  ASK:
ASK: ∩oˇ)ºa,0ºa¢ˇ
```

CE2: Direct function fixing (rank-error)
When the argument to the system function πFX is a character vector, then I-APL attempts to fix a directly defined function in the active workspace, rather than signalling a rank-error. If successful, the name of the fixed function is returned. If the character vector does not correctly define a directly defined function, a definition-error is signalled.

Example:
```
  πFX ∪PLUS:†≠ˇ Ω IMPLEMENTS ADDITION∪
PLUS
```

CE2.2 Canonical Representation
When the argument to the system function πCℝ is the name of a directly defined function, then a character vector representing the definition of that function is returned.

A-3.2 Extensions to primitive functions

CE3: Extension of multi-dimensional singles in primitive dyadic scalar functions (rank-error)
The behaviour of dyadic-scalar-extension is relaxed to allow a multi-dimensional array containing one element to conform with a non-scalar array. Instead of signalling a rank-error, the single is reshaped to match the shape of the other array prior to the application of the function. If both arguments are singles, the result has the rank of the argument of greater rank.

Example:

```
  ∩⊂ ⊂ᵒ⊂)≠⊂ 2 ∧
2 ∧ 4
  ᵒᵒ∩⊂ ⊂ᵒ⊂)≠⊂ ⊂ ⊂ ⊂ᵒ⊂
4
```

CE4: Extension of vector singles in primitive dyadic scalar functions (length-error)
The behaviour of dyadic-scalar-extension is relaxed to allow a one-element vector to conform with a non-scalar array. Instead of signalling a length-error, the single is reshaped to match the shape of the other array prior to the application of the function.

Example:
```
  ⊂ 2≠,⊂
2 ∧
```

CE5: Inner product axis extension (length-error)
The standard requires that the inner axes of the arguments to the derived Inner product function must be equal in length. I-APL relaxes this restriction by permitting either inner axis to have a unit length and still conform. Rather than signal a length-error, the unit axis is replicated (if necessary) to match the length of the other inner axis prior to application of the function. It is worth noting here that the equivalent extension is not made in the case of the Base Value function.

Example:
```
  ᵒ∩2 ∧ 4ᵒ0)≠.∂⊂ 5 ×ᵒ0
2 ∧ 5 ×
```

CE6: Grade up on character matrices (rank-error)

The Grade up function will return a result when applied to character matrices, rather than signal a rank-error. The result is an origin-dependent vector of indices which specify a monotone increasing lexical ordering of the rows of the character matrix. The atomic vector is used as the collating sequence.

Example:
```
  Ø4 5ᵒ∪YODA LElA LUKE DAℝTH∪
4 2 ∧ ⊂
```

CE7: Grade down on character matrices (rank-error)

The grade down function will return a result when applied to character matrices, rather than signal a rank-error. The result is an origin-dependent vector of indices which specify a

decreasing lexical ordering of the rows of the character matrix. The atomic vector is used as the collating sequence.

Example:
```
   Æ4 5º∪YO∆A LEⱢA LUKE ∆A℞TH∪
⊂∧24
```

CE8: Replicate (domain-error)

The standard requirement for the left argument of the Compress function to be near-boolean is relaxed to be non- negative near-integer. The effect is that for non-negative integer left arguments the specified number of copies of the corresponding element is returned, rather than signalling a domain-error. This extension is called Replicate.

The extension also applies to the first-axis and specified- axis forms.

Example:
```
   2 0 ∧/4 5×
4 4 × × ×
   2 ⊂/[⊂]2 ∧º∪⊂AT∆OℾU
⊂AT
⊂AT
∆Oℾ
```

CE9: Catenation of empties (domain-error)

The type of the array resulting from the catenation of two empty arrays is the same as the type of the left argument to the catenate function.

A-3.3 New system variables

CE10: πPW (syntax-error)

The current value of the system variable πPW determines the Printing Width, used to determine where folding occurs in the output of the display algorithm. The internal value set of πPW is the set of integers from 19 to 254 inclusive.

CE11: πHC (syntax-error)

The current value of the system variable πHC determines the status of Hard Copy control. The internal value set of πHC is the integers 0 and 1. When its value is 1 the display algorithm additionally routes all screen output to the system printer, if attached. When its value is 0 screen output will not be printed. The value of πHC in a clear workspace is 0.

A-3.4 New system functions

CE12: πWA (syntax-error)

The niladic system function πWA returns an integer scalar of the Workspace Available, a measure in bytes of the amount of unused workspace.

CE13: πID (syntax-error)

The niladic system function πID returns a 5 by 12 character matrix which identifies the characteristics of the operating environment, as follows:

Row	Meaning
1	Computer identification
2	CPU identification
3	Operating system identification
4	Display type identification

5   Port identification

See appendix B for the values in Macintosh, BBC, Archimedes and PC clones.

CE14: πTX (syntax-error)
The dyadic system function πTX provides a basic transmit facility in I-APL. The right argument may be a data array of characters or integers and the left argument may be one of the scalar integers 1,2 or 3. The data array is transmitted in ravel order to a destination determined by the left argument, as follows:

Left argument	Right argument	Destination
1	Raw codes	Screen
2	Raw codes	Printer
3	πAV elements or indices	Printer

The explicit result of πTX is 0 0⍴0.

CE15: πMC (syntax-error)
The dyadic system function πMC provides I-APL with a mechanism for executing Machine Code programs. The left argument is a vector of characters or integers that represent a program written in the machine language of the host computer. The right argument is a character array of maximum rank 3 containing the names of argument and result variables. The explicit result of πMC is 0 0⍴0.

A-3.5 System commands

CE16:)PCOPY (incorrect-command)
The system command)PCOPY provides I-APL with a Protected Copy facility. The behaviour is similar to that produced by)COPY, except that in the event of a name clash between objects in the library and active workspaces, the objects are NOT copied.

As with)COPY, two forms are permitted;)PCOPY WS will perform a protected copy on all defined functions and variables in the library workspace WS.)PCOPY WS OBJ will perform a protected copy on the object OBJ in the workspace WS. Multiple object names may not be specified.

CE17:)ERASE A B C (incorrect-command)
Multiple global objects in the active workspace may be erased by using the)ERASE system command with more than one object name. Notice that)ERASE is not atomic: if the operation fails before completing, some objects may have been successfully erased, whereas others may not. Objects are erased in the order specified.

CE18:) (incorrect-command)
A single right parenthesis entered in immediate execution mode causes the last APL statement entered in immediate execution mode at the current level of the state indicator, if any, to be re-displayed for editing.

A-3.6 Defined function

CE19: Ambivalent functions (syntax-error)
A dyadic defined function may be invoked either with two arguments or with a single argument on the right. In the latter case the name class of the left argument name is initially 0.

CE20: Leaky locals (implicit-error)
The initial value on entry to a defined function of a localised system variable is determined by its value immediately prior to localistion.

CE21: Duplication of local names (defn-error/domain-error)
Duplicates in the local-name-list of a defined function are permitted. Names are localised and labels assigned their values in the order in which they appear in the definition. As a final step the argument names, if any, are assigned their values from left to right.

CE22: Function line display: [nπ] (defn-error)
In function definition mode the form [nπ] may be used to display line n.

CE23: Function line edit: [nπm] (defn-error)
In function definition mode the form [nπm] may be used to edit line n. The number m is ignored.

CE24: Delete function line: [~n] (defn-error)
In function definition mode the form [~n] may be used to delete line n.

CE25: Create empty function line: [n] ESC (defn-error)
A blank line may be inserted into a function definition by signalling an attention inside function definition mode.

CE26: Tolerant imbalanced quotes (defn-error)
Imbalanced quotes are permitted in a function definition body-line.

A-3.7 Identifiers and ideograms

CE27: † and ˘ as identifiers (syntax-error)
The single character tokens † and ˘ may be used as identifiers.

CE28: Fifty-two additional letters (syntax-error)
The letter diagram is extended to include the lowercase alphabet and the underscored lowercase alphabet. This permits their use in I-APL identifiers.

CE29: Additional ideograms (syntax-error)
A number of additional ideograms is included in I-APL. These are given in the πAV table above.

APPENDIX B:
πID for all current ports

B.1 IBM Personal Computer and compatibles

Row	Contents
⊂	Pℂ
2	808X
∧	MSⅮOS
4	TEXT or ℂΓA or EΓA or HEℝCULES
5	PΓHℂB

B.2 BBC B and Master

Row	Contents
⊂	BBℂ B
2	×502
∧	OS ⊂.2
4	MODE <n>
5	Tony ℂheal

B.3 Archimedes

Row	Contents
⊂	Arℂhimeδes
2	AℝM
∧	ℝISℂOS
4	MODE <n>
5	Tony ℂheal

B.4 RC Piccoline

Row	Contents
⊂	ℝC PℐℂℂOLℐℕE
2	808X
∧	ℂℂP/M≠ℂ-DOS
4	TEKST
5	KSA7

B.5 Apple Macintosh (IAPL/Mac)

Row	Contents
⊂	MAℂℐNTOSH
2	×8000
∧	FℐNDEℝ
4	ℕ/A
5	ℐAℕ A ℂLAℝK

APPENDIX C:
Writing a Machine Code Resource using LightspeedC

Below is a simple program written in C, which can be compiled into a self-contained code resource using LightspeedC (or THINK C). Select 'Set Project Type' from the Project Menu. Give it the resource type 'APCC' and a resource ID like 1234. You can also give it a name, but IAPL/Mac won't use that.

Once compiled by 'Build Code Resource', it can be moved by means of ResEdit into the IAPL/Mac application. With IAPL/Mac running, you can call it by e.g.

  BEEP: ⊂2∧4 πMℂ ∪˘∪,0^{O˘}

Thereafter BEEP 5 will issue 5 system-beeps.

An 'APCC' resource can read from the workspace variables which the πM℃ facility passes to it, and write back into them, using the ptab pointer array. It can communicate interactively by means of calls to the Macintosh ROM routines, e.g. to employ its own windows and dialogs, although there is obviously the possibility of interfering with IAPL/Macs own windows and dialogs. In addition, IAPL/Mac passes it a pointer argument, msg_str, which points to an 80-byte buffer initialised to '\0' followed by blanks. A non-null string value (in C format) placed in this buffer will be output to the session log when the 'APCC' routine terminates.

The 'APCC' should return the integer value 1 on completion, to signal success, or 0 to signal failure. If 0 is returned, then IAPL will issue its DOMAIN ERROR message.

```
/*********************************************/
/* Sample quadMC code resource for IAPL/Mac   */
/* Beeps n times, given by value of arg(0)     */
/*              Written by Ian A. Clark             */
/*********************************************/

#include "MacTypes.h"

/*** These #defines might usefully be put in a header file, for use */
/*         by all 'APCC' code resources ... */
#define WSENTRY(m)      ((char *) ptab[m])
#define Type(n)          (WSENTRY(n)[0])/* type of APL VAR: 'n' or 'c' */
#define Esize(n)         (WSENTRY(n)[2])/* bytes per element (0=Booln) */
#define Rank(n)          (WSENTRY(n)[3])/* number of axes (0..7) */
#define Elements(n)    ((int)MINT( WSENTRY(n)+4))  /* tot.no.of elems */
#define Rows(n)        ((int)MINT( WSENTRY(n)+6))  /* no. of mx rows  */
#define Columns(n)     ((int)MINT( WSENTRY(n)+8))  /* no. of mx cols  */
#define ScalarInteger(n)   Rows(n)                        /* value of scalar */
#define ScalarContents(n) ((byte *)WSENTRY(n)+6)  /* element[1] (vc) */
#define VectorContents(n) ((byte *)WSENTRY(n)+8)  /* element[1] (vc) */
#define MatrixContents(n) ((byte *)WSENTRY(n)+10) /* element[1] (mx) */

#define byte  unsigned char
#define JMAX      10

main( n, ptab, msg_str)
    int              n;                      /* the number of entries in ptab table */
    unsigned char    *ptab[];        /* table of abs pointers into workspace */
    char             *msg_str;      /* buffer for returned msg to be output */
```

```
{
        int                     i,j;

        /* Expects just one integer scalar value, in *ptab[0]. Hence n==1 */
        if (n!=1) return 0;
                            /* 1 signals OK to IAPL, 0 forces DOMAIN ERROR */

        /* Verify that value(0) is really numeric, having just one element */
        if ( (Type(0)!='n') || (Elements(0)!=1) ) return 0;

        /* Pick up the value of the scalar integer in arg(0) */
        j = ScalarInteger(0);

        /* Use the integer value in j to issue between 1 and JMAX beeps */
        if (j<=0)  j=1;
        if (j>JMAX) j=JMAX;
        for (i=0; i<j; i++) SysBeep(10);

        return 1; /* 1 signals OK to IAPL, 0 forces DOMAIN ERROR */
}

int MINT(p)
        byte *p;
/* p points 2-byte lo/hi. MINT(p) returns int value with swapped bytes */
{
        return (*p)|((*(p+1))<<8);
}
```