

# **MacAPL Summary**

by Michael C. O'Connor  
Leptonic Systems Co.

This document summarizes the MacAPL language (version 2.21) from Leptonic Systems Co., 405 Tarrytown Rd. #145, White Plains, NY 10607 (914) 682-0377. It uses the New York 10 and 18 point, and the APL 12 point fonts. The APL font should be available from the same source this document was obtained from. To read this file successfully, the APL font should be installed in your system using the "Font/DA Mover" application.

This document is ©1987 Leptonic Systems Co. Unmodified copies may be distributed without charge. For a complete description of MacAPL refer to the MacAPL User Guide.

## **Getting Started**

Starting out in APL is as easy as  $2+2$ . In its simplest form, APL works like a desk calculator. You type in a mathematical expression in an algebra-like notation, and APL types out the answer for you when you press the Return key. Here are a few sample calculations:

2+3.14  
5.14

« You type this and press Return  
« APL types the answer

7-3+2  
2

« A second example

(7-3)+2  
6

« The second example with parenthesis

The computer indents six spaces as a visual signal to you each time it is your turn to type, but always displays the results at the left margin. This makes it a little easier to tell who typed what when you look at the screen, and you can also tell when the computer is ready to accept your input when the insertion point is indented.

The second example seems a little strange. Many people would say the answer to this calculation is 6 rather than 2. Take 7 and subtract 3, giving 4, and then add 2 to that and get the answer 6. Why did APL say the answer is 2?

APL executed the calculation right to left. It took 2 and added 3 to it, giving 5, and then subtracted 5 from 7 to get the answer of 2. Therefore the following is an important rule to remember:

**APL always executes expressions in strict right to left order, unless parenthesis are used to change the order of execution.**

Why is there such a rule in APL? Most other languages use a hierarchy scheme to establish an order of precedence. For example, a common rule would be to execute all multiplication and division before all additions and subtractions. In APL there are many more functions than in most other languages, and setting up precedence rules would be nearly impossible to remember. Instead, there is the consistent, right to left order of execution, and things become much easier overall.

APL can do a lot more than just add and subtract numbers. There are many different functions and each has a special symbol to identify it. There is no need to be overwhelmed by the large number of different functions, just like one does not have to be overwhelmed by the large number of books in a library. You can browse through everything, picking up only the specific tools you need to get the current job done. For example, the basic math operations:

2+3  
5

2-3  
TM1

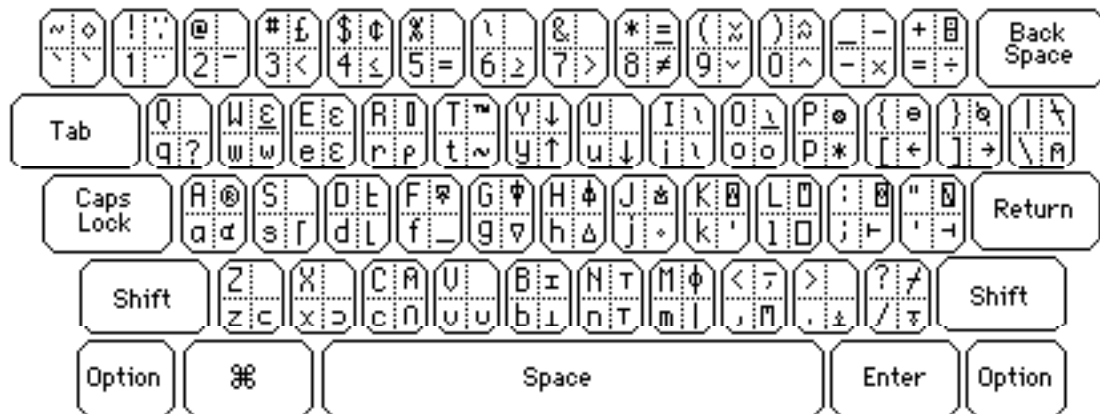
2-3  
6

2÷3  
0.6666666667

These are all you need to use APL for some quick calculations. Notice the "-" used for multiplying and the "÷" used for dividing which are taken from normal mathematical notation. The subtraction example shows the negative sign for numbers is raised up higher than the normal dash because it is useful to distinguish the subtraction function from the numeric negative sign.

## Keyboard Layout

The APL font is embedded inside the APL interpreter file itself, so the interpreter will work properly with any Macintosh system disk (but for higher quality printing, the APL 24 point font should be in your System File).



Upper left - Shift	Upper right - Option-Shift
Lower left - Unmodified	Lower right - Option

## The Workspace

When you click on the MacAPL icon to start the application, you are presented with a window named "Untitled", and it is empty except for the words "Clear Workspace" in the upper left corner. APL prompts you to enter an expression by indenting six spaces and displaying the blinking insertion point.

The window is a view into an APL workspace. A workspace holds a collection of data variables you create, and programs (known as functions) which you define. When you save a workspace in a file, the whole set of functions and variables are saved together. When a saved workspace file is opened, the entire collection becomes available to you in the state in which it was saved.

## Numbers and Characters

The simplest APL expression you can type is a single number. If you type one in, it just types it back at you. The computer accepts numbers with or without decimal points, and also in scientific notation:

5  
5

« You type this and press Return  
« APL types back the "answer"

TM5.654  
TM5.654

« Here is a negative number

5.634E75  
TM5.634E75

« This number is in scientific notation

3.14E<sup>TM</sup>20  
3.14E<sup>TM</sup>20

« This positive number is quite small

Negative numbers should be typed in using the negative sign (which is Option-2), not the dash (which means sign inversion). Scientific notation is most often used to represent very large or small numbers. In it, a decimal number is followed by the letter "e" or "E" and then another number holding a power of ten. APL may decide to use scientific notation to show very large or small numbers in a reasonable amount of digits.

MacAPL uses the Standard Apple Numeric Environment (SANE) for its calculations, and therefore some functions can generate "INF" and "NAN" results as values. "INF" represents "infinity", and "NAN" represents "not a number". Although these special values can be generated from calculations, you cannot type them in directly. See the MacAPL User Guide or the Apple Numerics Manual for more about SANE.

## Character Values

APL deals with text as easily as it deals with numbers. Text is made up of characters which are symbols typed on the keyboard. To enter a string of text characters in APL, type it in, enclosed with single quotes (the key next to the Return key, unshifted).

'x'	« A text character
x	« APL types it back

'Hello, world!'	« A "character vector"
Hello, world!	

One of the best features of APL is that it can deal with whole groups of numbers (or characters) as easily as it deals with single ones:

2 3 4	« Three numbers separated with
2 3 4	« spaces are treated as a unit

10 + 2 3 4	« Add 10 to all three numbers
12 13 14	

12 54 22 - 5 3 12	« Do 3 subtractions at once
7 51 10	

1 2 3 + 4 5 6 7	« Adding 3 numbers to 4 others
Length error	« causes an error message
1 2 3 + 4 5 6 7	
°	

You can give APL a string of numbers in a row, separated with spaces, as a group, and add another group of equal length to it; or give a single number and APL will add it to all the other numbers at once, as shown in the example above.

In the last example we tried to add three numbers to four numbers. APL only knows how to add one number to a group of numbers, or to add two groups of numbers of equal size. The computer reports a "Length error" and displays the offending statement, placing a pointer under the part of the statement where the error is detected.

A single number (or character) standing alone is called a **scalar**. A group of numbers separated by spaces (or a character string enclosed in quotes), is called a **vector**. A vector has a single **dimension** which is its length. We can also create multi-dimensional **arrays** of numbers or characters. It is also possible to mix numbers and characters together in a **nested array**.

If you have a vector, you can use a subscript or index to specify which elements in the vector you want to address:

12 44 395[2]	« The second element in this
44	« vector is the scalar 44
12 44 395[2 1 3 2 1]	« Index many elements at once
44 12 395 44 12	
'Hello there'[1 2 10 11]	« Index a character vector
Here	

## Dyadic and Monadic

Addition, subtraction, multiplication, and division are four of APL's **primitive functions**. A primitive function such as addition accepts **arguments** (the numbers on the left and right sides of the plus sign), and it returns a **result** (the sum which is displayed).

Primitive functions are built into the APL language and are called up by using the various APL symbols, + - ≠ = ^, etc. Soon we will describe user-defined functions, which are given symbolic names; and finally there are system functions, which have special symbolic names and are built into APL.

There are many primitive functions in APL and they are divided into classes, depending on the number of arguments they accept. So far we have only discussed **dyadic** functions, which accept two arguments. For example, in "2+3", the plus sign is a primitive function representing addition. Addition is a primitive dyadic function which accepts two arguments and returns a result. The scalar number 2 is the left argument, and the scalar number 3 is the right argument. The result is the sum of the two arguments, the scalar number 5.

Primitive functions are called **monadic** when they only take one argument, and dyadic when they take two. Here are some examples of monadic primitive functions:

-5	« The NEGATION function flips
TM5	« the sign of the argument
^5	« The IOTA function
1 2 3 4 5	« generates consecutive integers
2-^5	« Monadic IOTA, dyadic SUBTRACT
1 0 TM1 TM2 TM3	

In the first example we took the number five and applied the negation function to it, which resulted in the number negative five.

The third example shows the Iota function. This function generates a vector result containing all the counting

numbers up to the one you give in the argument.

In the last example we combined the monadic interval function with the dyadic subtraction function. APL figured out that the "-" means dyadic subtraction and not monadic negation by examining the context and syntax of the statement. Remembering the right to left order of execution, we can deduce that APL first calculated  $^5$ , which generated the five-element numeric vector 1 2 3 4 5. Following that, it performed dyadic subtraction. It took 2 and subtracted each number from it in turn, giving us the five numbers in the answer.

Many of the APL symbols have both monadic and dyadic functions attached to them, and the way they are used determines which function is performed. Usually the two functions do logically similar things.

## Using Variables

You can save the results of an expression by assigning a name to it. Later, you can use the result in another expression by referring to the name. The assignment arrow is used to do this, and it is typed as Option-[.

	X $^5$	« Create a new variable « (APL does not display a value)
5	X	« Type the variable name « The value is displayed
8	X+3	« Use it in a calculation « APL displays the answer
1 2 3 4 5 6	Variable $^6$ ^Variable	« Variable names can be any length « Use it in an expression « APL displays the answer
Value error	variable variable	« Upper and lowercase are different « APL generates an error « because there is no such variable
Hello	variable $^$ "Hello" variable	« Assign text to a variable « Ask for the value « APL displays it
1 2 3 4 5 6	Variable	« This other variable still « retains its value

In the above examples, we have some simple assignment statements. The first example creates a variable named X and assigns a value of 5 to it.



Variables can contain characters or numbers. They can contain a single character or number, or a whole multidimensional array of characters or numbers:

<pre> A“5 B“6 A+B 11 </pre>	<pre> « Set some variables « Add them together </pre>
<pre> A“1 2 3 4 5 6 7 8 9 B“0 1 2 3 4 5 6 7 8 A+B 1 3 5 7 9 11 13 </pre>	<pre> « Make them bigger arrays « Add them together </pre>

In the above example, we see that the expression A+B could cause a whole series of additions to occur in one step. This automatic extension of primitive functions, which operate element by element on large arrays at once, is one of the more powerful features of APL.

A while back we saw that you can index into an array in order to extract one or more of the values from it. A similar concept associated with the assignment function allows you to change one or more elements stored in a variable:

<pre> A“9 8 7 6 5 4 3 2 1 A[3] 7 </pre>	<pre> « Start with an array of numbers « Index an element from it </pre>
<pre> A[3 8 1] 7 2 9 </pre>	<pre> « Index several elements </pre>
<pre> A[3]“55 A 9 8 55 6 5 4 3 2 1 </pre>	<pre> « Change the 3rd element of A « Check the value of A </pre>
<pre> A[3 8 1]“77 88 99 A 99 8 77 6 5 4 3 88 1 </pre>	<pre> « Change several elements « Check the value again </pre>

## Multidimensional Arrays

As implied previously, data can take the form of large multidimensional arrays. In order to form them, we need to introduce a new primitive function:

$A \leftarrow 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$	« Start with a vector
$3\ 3 \textcircled{R} A$	« Reshape into a matrix
1 2 3 4 5 6 7 8 9	
$A \leftarrow 3\ 3 \textcircled{R} A$	« Save the new shape
$5 + A$	« Add 5 to each element
6 7 8 9 10 11 12 13 14	
$2\ 3\ 4 \textcircled{R} A$	« Make a 3-D array
1 2 3 4 5 6 7 8 9 1 2 3  4 5 6 7 8 9 1 2 3 4 5 6	

The " $\textcircled{R}$ " symbol invokes the reshape function. It takes a string of numbers or characters and molds it into an array which has the dimensions specified in the left argument. The expression " $3\ 3 \textcircled{R} A$ " requests the value in A to be displayed as a 3 (down) by 3 (across) array. The expression " $2\ 3\ 4 \textcircled{R} A$ " requests the values in A to be displayed as a 2 (deep) by 3 (down) by 4 (across) array. The reshape function repeated the elements in A as many times as necessary in order to complete the array. APL displays multidimensional arrays a plane at a time, with each plane separated by a blank line.

In order to index into multidimensional arrays, semicolons are used to separate the indexes for each dimension:

$A \leftarrow 3\ 4 \textcircled{R} ^{12}$	« Make a matrix
A	
1 2 3 4 5 6 7 8 9 10 11 12	
$A[2;3]$	« 2nd row, 3rd column

7

A[2;4 2]

« 2 elements from 2nd row

8 6

A[:,3]

« The whole 3rd column

3 7 11

```

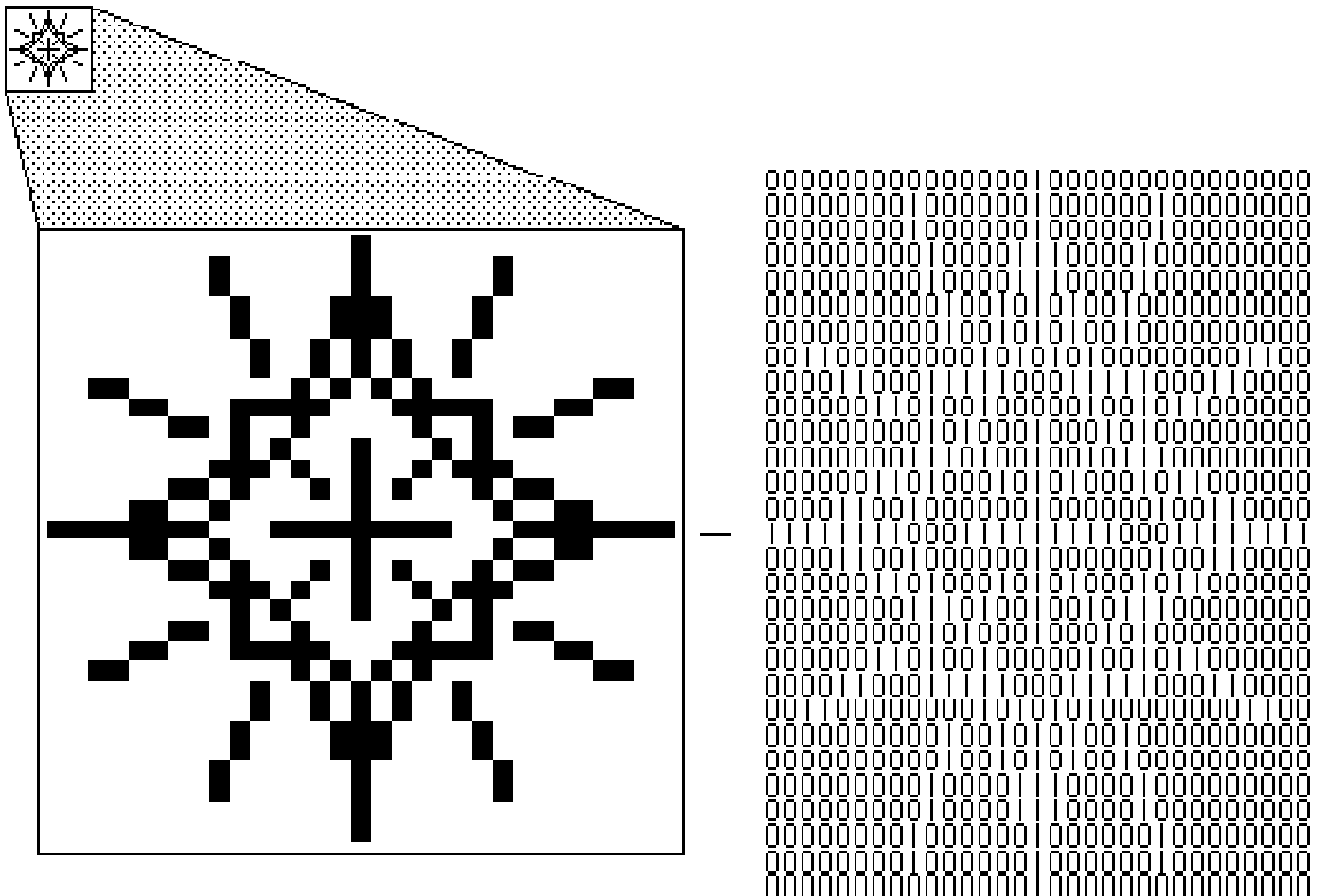
A[3;]"99
A[;3]"88
A
1 2 88 4
5 6 88 8
99 99 88 99
9 10 88 12

```

« These work in assignment too

## Picture Variables

The Macintosh computer is a graphics-oriented machine. A Macintosh picture consists of a rectangular array of dots which form the image. Each dot is either on (black) or off (white). If we represent each dot in the image as a number (1 for black and 0 for white), we can represent the picture as a rectangular array of 1s and 0s:



A numeric array such as this is a common data structure in APL, and is known as a boolean matrix. MacAPL allows you to bring in a Macintosh picture through the clipboard and assign the picture to an APL variable. The resulting picture variable can then be directly manipulated by all the built-in APL primitives exactly as though it

was a boolean matrix. For example, suppose we created a picture of an owl

with MacPaint, copied it onto the clipboard, exited MacPaint and started up MacAPL. We can then put the picture of the owl into an APL variable:

```
OWL←¬CLIPBOARD          « Put clipboard pic into a variable
```

Once you have the picture variable, you can manipulate it using APL primitives:

```
FLIPOWL←"OWL           « Make a "flip vertical" picture
¬CLIPBOARD←FLIPOWL      « Put flipped picture on clipboard
```

If you were to exit MacAPL and paste the clipboard into another application such as MacPaint or MacWrite, a picture of an upside-down owl would appear.

When you ask to see a picture variable in MacAPL, a picture window appears which holds the picture. The picture window is a separate Macintosh window. You can move it around the screen by clicking anywhere on the picture and dragging it around. You can click on other windows (such as the workspace window) to bring them to the front, then click on the picture window to activate it again. You can close the window if you want by first making it the active window and then using the "Close" command from the File menu. Note that in order to type more commands into the APL workspace, you must bring the workspace window to the front first by clicking on it.

If you ask to display a different picture variable, the image in the picture window will change to reflect the new picture. There is only one picture window, which is shared by all, even if there is more than one workspace running. Normally the picture window automatically takes on the shape of the picture it is displaying, but you can control the size and position of the window by adjusting the values in the ¬PICT variable (see its description in the system variables list).

Although picture variables display as a picture on the screen rather than as a matrix of 1s and 0s, it is treated as a matrix of 1s and 0s when used as an argument to an APL primitive.

Many functions return pictures when given pictures as arguments. Some functions can not logically return pictures however. For example the result of the expression "OWL+OWL" doesn't conform to the rules for a picture anymore (only 1s and 0s), and so APL returns a normal matrix of numbers having the values 0, 1, and 2, and displays them in the workspace window. The conversion of pictures into numbers is automatic whenever required.

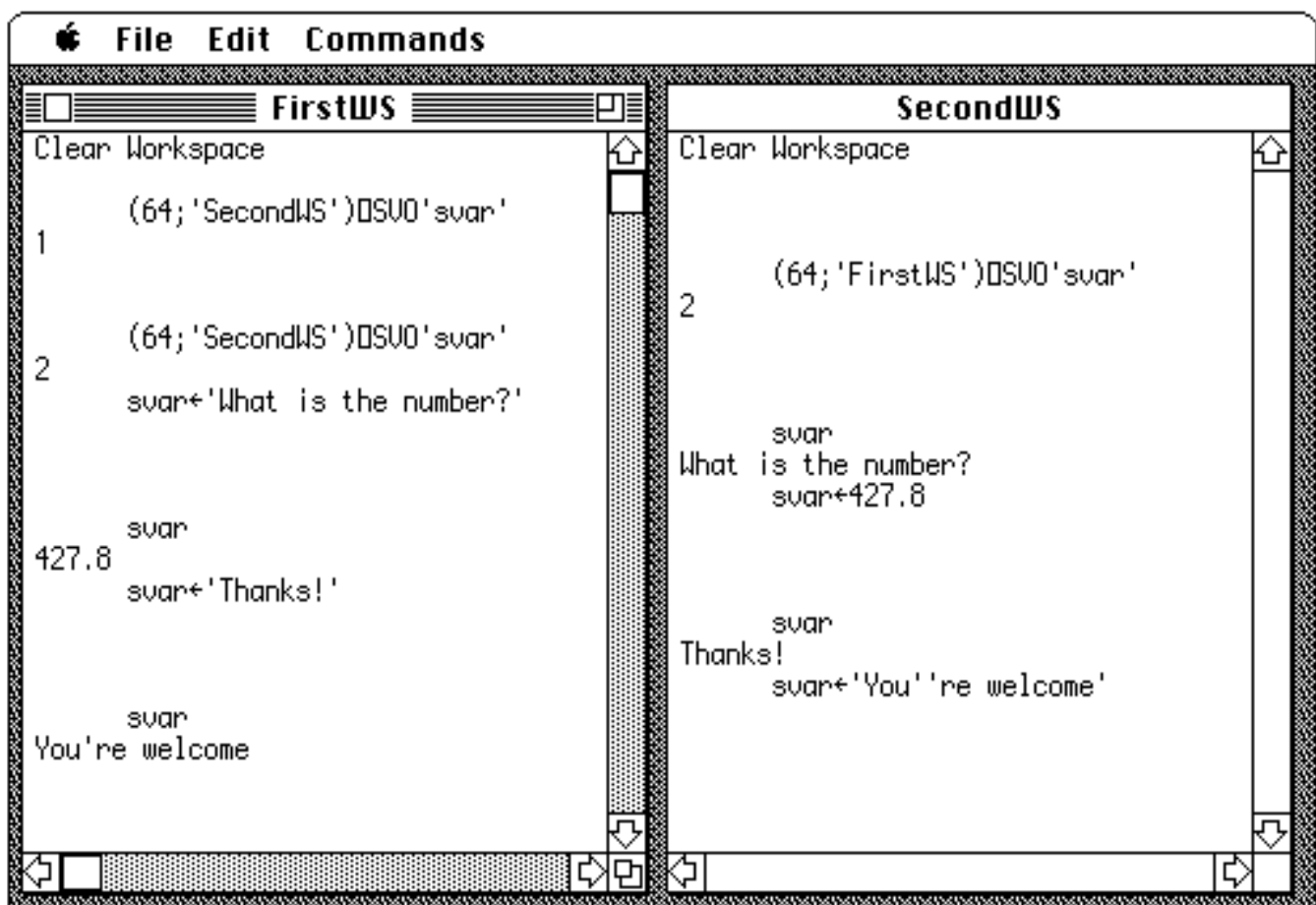
The only difference between a picture variable and a boolean matrix is the way it is displayed (picture variables in the picture window, boolean matrices as 1s and 0s in the workspace window). In fact, you can explicitly change pictures into booleans and back by using some MacAPL primitives. The monadic "+" when given a picture variable will return the identical data as a boolean matrix. The monadic "↑" when given a boolean matrix will return the identical data as a picture variable. For example you can create pictures from scratch by making the image first as a boolean matrix using APL, then using "↑" to make it into a picture, and finally you can export the new picture to another application by assigning the picture to the clipboard.

## Shared Variables

MacAPL provides shared variables as a way to communicate with things outside of the local workspace environment. Specifically, these are other workspaces, other users on the network, and disk files.

Facilities are provided to find out who is offering to share with you, to synchronize transfers of data between shared variables, and to allow safe updating of disk files shared by multiple network users. There are also ways to create, delete and append to files, and there is a way to write functions that bring up the standard system file dialog boxes, so a user can specify the file to share.

To see how this works we can try sharing a variable with some immediate mode statements. Below we have opened two workspaces with different names and will share a variable between them:



In this example we have the two workspaces called FirstWS and SecondWS. The first offer to share is made by FirstWS when (64;'SecondWS')DSVO 'svar' is executed. The number 64 is a code meaning that you want to share with another workspace on your own computer. The exact name of the other workspace is also specified.

The right argument specifies the name of the variable we want to share, 'svar'. The variable that we are offering does not need to exist in the workspace before we make the offer; if svar does not already exist, it will be created (and its contents will be set to an empty character vector).

The  $\neg$ SVO function returns a number indicating the degree of coupling the variable now has. The result of 1 means the offer was successful but the coupling is not complete because the other workspace has not yet accepted the offer.

Now we switch over to SecondWS and execute (64;'FirstWS') $\neg$ SVO 'svar' which offers a variable of the same exact name with FirstWS. In this case the result of the  $\neg$ SVO function is a 2, indicating that the variable is fully coupled. It can now be used to pass data back and forth.

Switching back to FirstWS, we execute the same sharing offer as we did before. This is not necessary, but it confirms to FirstWS that the offer has been accepted since the result of 2 is now produced. In fact it is common to write a function such that it loops on the  $\neg$ SVO offer until a 2 is returned.

Seeing that the offer has been accepted, FirstWS executes an assignment statement to place some data (the question "What is the number?") into the shared variable.

Switching over to SecondWS, we ask to display svar. It displays the question placed into it by FirstWS. Next, SecondWS assigns a number to the variable.

Finally, FirstWS takes another peek at svar and the number SecondWS has assigned is displayed. The conversation continues with the "Thanks!" and "You're welcome" messages.

The two workspaces are sharing a single variable between them, and it takes on the last value assigned by either party. In fact there is only one copy of the variable in memory, which both workspaces use in common.

When a variable is shared with another user on the network, the sequence is the same except for the initial  $\neg$ SVO offers. The statement (96;'username') $\neg$ SVO'svar' offers to share with another Macintosh running MacAPL whose user name (set with the Chooser desk accessory) is set to 'username'. The  $\neg$ PORTS variable will list the user names of other MacAPL users on the network.

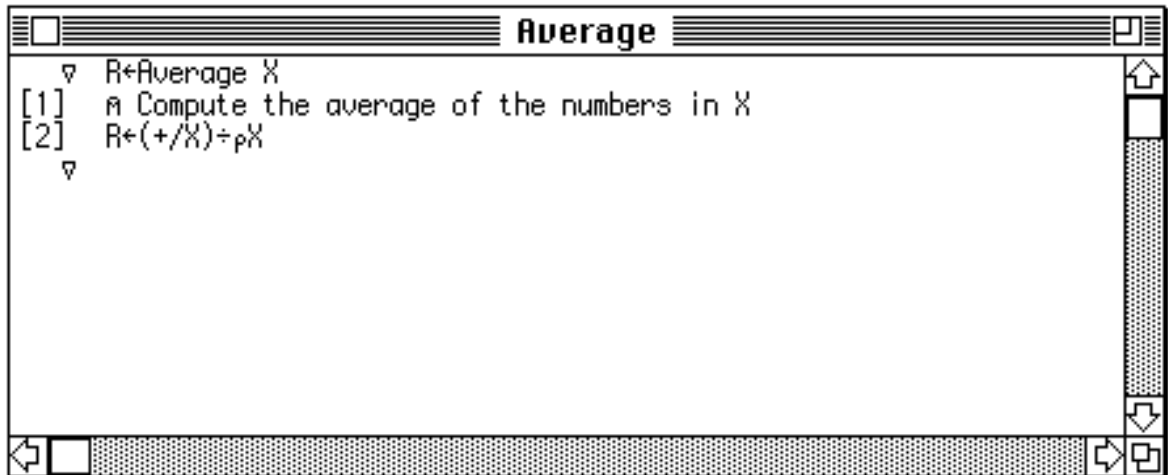
When a variable is shared with a file, the shared variable "becomes" the file by presenting itself as a character vector whose length is the same as the length of the data in the file. As you manipulate the shared variable, MacAPL reads and writes the file. For example if VAR was shared with a file, X"VAR[^10] would read the first 10 bytes of the file, and VAR[^10]"X would write the first 10 bytes.

More details about sharing variables and controlling the network are in the descriptions of  $\neg$ SVO,  $\neg$ SVR,  $\neg$ SVC,  $\neg$ SVQ,  $\neg$ ENQ,  $\neg$ DEQ,  $\neg$ PORTS, and  $\neg$ MSG.



## Creating Defined Functions

When you use the "Edit APL Object..." command, a dialog box appears which lets you specify the name of the function you wish to edit. When you do that, an editing window appears. Now we use normal Macintosh editing techniques to put the definition of the function into the editing window:



The first line of the text is the function header . It describes the name of the function, the arguments the function accepts, whether there is an explicit result to the function, and what the local variables are. The other lines make up the body of the function.

The little "©" symbols and the line numbers in the left margin automatically appear as you edit. The rest is what you type in. When you append or insert new lines in the function, APL automatically inserts new line numbers and adjusts them.

The function in the example will compute the average of a set of numbers. The first line, the function header, is a model describing the way the function will be used. The name of the function is "Average", and it takes a right argument represented by the "X". The "R←" portion of the header indicates that the result of the function can be used further by the rest of the expression which invokes the function. Some examples of the way the function could be used:

2.5	Average 1 2 3 4	« Average 4 numbers « APL displays the result
2.5	avg←Average 1 2 3 4 avg	« Assign the result « Check the value
8	2+(Average 1 2 3)+4	« Use it in an expression

Within the function, the variables R and X are called **dummy variables**. X represents the data passed to the function. R represents the data to be passed back as the result. In our example they have been named R and X, but they could have been given any names we wished.

Other variations are permitted in the function header. Often, a function could use additional variables within it for the purposes of holding temporary results. In that case, additional names could be specified at the end of the function header, each one preceded by a semicolon. Such variables are called **local variables**. You can define and use them in the body of the function. When the function ends, the local variables are automatically erased. When a local variable is used with the same name as a "real" variable in the workspace, the real variable is shadowed and is not available to the statements within the function. When the function ends, the local variable disappears and the real global variable reappears in an unchanged state.

Arguments, result, and local variables need not appear in a function header at all. A function can be defined that has no arguments at all. It would be invoked by simply typing the name alone on a line. All together, there are six basic types of function headers; monadic, dyadic, and **niladic** (the no-arguments case), and each of those with or without an explicit result. Finally, any function can have local variables.

The rest of the lines in the function perform the actual processing. Line one in the averaging example is a comment line which is there for documentation and is not executed. The second line performs the actual calculation of the average. It uses "+/" to add up all the numbers in X, and then performs division by "ⒺX", which computes the amount of numbers which were specified.

## Error Handling

When APL detects an error, it displays an error message, and also shows the expression which generated the error, along with a pointer to the approximate position of the error.

If the error was generated by an expression inside a defined function, the error report also includes the name of the function and the line which was executing:

Func	« Run a function called Func
Value error	
Func[3] value“nul+5	
o	

In the above example, the niladic function "Func" was executed, and everything went OK until line 3, when an attempt was made to use a variable "nul". Since the error is a "Value error", it seems "nul" was never defined (was never given a value).

APL remembers the state of execution when the error occurred in a place called the **State Indicator**. Once we fix the problem that created the error, we can instruct APL to resume execution where it left off.

To examine the State Indicator, you can use the menu command "Show State Indicator" in the "Commands" menu. This will bring up a window which displays the indicator.

Since our problem was caused by "nul" not being defined, let's fix that:

```
nul←0                                « Give nul a value
```

Now we can tell APL to try to resume the interrupted function. We do that by entering a branch statement. We want to tell APL to resume execution at line 3:

```
‘3                                    « Ask to resume at line 3  
                                     « (execution resumes successfully)
```

There is a way for your program to trap errors and handle them in your own way. See the description of the  $\neg$ ALX,  $\neg$ ELX,  $\neg$ DM, and  $\neg$ EN facilities.

## Menu Commands

### Apple Menu

**About MacAPL...** shows the version number and author credits.

**Reference...** opens a reference window containing on-line help.

The reference window contains text briefly describing the features of MacAPL. Usage instructions are explained in the window text. The text is read in from the "Reference" file, which you should have on the default directory on your disk. The "Reference" file is a text file which can be modified with any text editor. It should have been distributed to you along with the MacAPL application.

**Key Caps** is an accessory of particular interest in MacAPL. It allows you to easily determine how to type any of the special APL characters, most of which are typed in conjunction with the Option key.

Another important accessory is the **Chooser**. This accessory allows you to turn on the network, specify your network user name, and select the printer you wish to use.

## File Menu

**New** opens up a new, clear workspace for you to use.

**Open...** lets you open a previously saved workspace.

**Close** will close the frontmost window the same as if you clicked on its close box.

**Save...** will let you save the workspace (in the active window) in a file. The command is dimmed and not usable in the freely distributed version of MacAPL. See the full User Guide for information on how the Save commands are enabled in the purchased version.

**Save As...** will let you save the active workspace under a new name. The command is dimmed and unusable in the freely distributed version as described above.

**Page Setup...** brings up the standard dialog allowing you to set printer parameters.

**Page Format...** brings up a dialog box that lets you control the format of the pages printed by the Print Text... and Print APL Objects... commands. You can specify a header line for all pages, and control if there should be page numbers, the workspace name, and a timestamp on each page.

**Print Text...** allows you to print some or all of the text in the frontmost window. If some text in the window has been highlighted, only that text is printed. If no text is highlighted and there is only a flashing insertion point, all of the text is printed.

**Print Objects...** presents a list of the functions and variables in the workspace which may be printed. You select the items you want to print by clicking on the names in the list. The items are then printed in a formatted listing.

**Quit** closes all windows and ends MacAPL.

## The Edit Menu

**Undo** allows you to abandon changes you made in an editor window.

**Cut** moves the selected text to the clipboard.

**Copy** places a copy of the selected text on the clipboard.

**Paste** inserts text from the clipboard into MacAPL. In an editor window, it works in the usual manner. In a workspace window, the pasted text is "typed in" to the workspace and executed as input. Therefore the pasted text is always appended to the end of the text already in the workspace.

**Clear** deletes selected text without placing it in the clipboard.

**Show Clipboard** brings up a window containing the text in the clipboard.

**Send to Back** will move the front window behind all the others.

## Commands Menu

**Show State Indicator** opens a window which shows the state indicator of the workspace in the active window. As the workspace executes, the window changes to show the state indicator dynamically (this slows down execution considerably). There is a scroll bar in the state indicator window. It controls the speed of execution of the workspace.

**Interrupt Execution** halts the active workspace. You can then examine the state of variables, etc. and then continue execution by branching back into the interrupted function.

**Edit Object...** will display a dialog box showing the names of the functions or character vector variables in the workspace you may edit. You then can select a name, and an editing window will be opened so you can either edit the function or variable.

**Copy Objects...** lets you copy functions and variables from a saved workspace on disk into your active workspace. First a dialog box appears allowing you to select a workspace file. Then a list of all the objects in that workspace appears. Click on the objects you wish to copy, then click OK.

**Lock Objects...** allows you to lock objects in the active workspace. When you lock an object, its value cannot be changed. If it is a function, you will not be able to edit it, or look at its definition with  $\neg$ CR, etc. If it is a variable and you attempt to assign a new value, the statement will execute without error, but the new value will not be assigned. Locked objects can still be expunged (erased).

**Hide Objects...** allows you to hide objects in the active workspace. A hidden object will continue to exist, but will not be listed in any dialog boxes or by such facilities as  $\neg$ NL, etc. Once they are hidden, a user must simply "know" the object exists in order to use it. It is kind of like having the password for the object being the name of the object itself.

**Expunge Objects...** allows you to expunge (erase) objects from the active workspace.

# Primitive Functions and Operators

## Scalar Dyadic Functions

These functions accept numeric arguments and return numeric results. The arguments can be scalar numbers, in which case the result is also a scalar number. If one argument is an array and the other is a scalar, the scalar argument is applied to each element of the array argument. If both arguments are arrays of the same shape, the result is an array whose elements are produced by operating on the corresponding elements of each argument. Arguments which are arrays of differing shapes are not permitted and an error results.

$A+B$	Add A to B.
$A-B$	Subtract B from A.
$A\cdot B$	Multiply A by B.
$A\div B$	Divide A by B.
$A^B$	Raise A to the B power.
$A\mu B$	The remainder of B divided by A.
$A\beta B$	The larger of A or B.
$A\partial B$	The smaller of A or B.
$A\lceil B$	The base-A logarithm of B.
$A!B$	The number of combinations of B things taken A at a time.
$A\circ B$	Circular functions: Operate on B according to the value in A. For example to get SIN(B) use $1\circ B$ . (radians)
-7	arctanh
-6	arccosh
-5	arcsinh
-4	$(-1+B^2)^{.5}$
-3	arctan
-2	arccos
-1	arcsin
0	$(1-B^2)^{.5}$
1	sin
2	cos
3	tan
4	$(1+B^2)^{.5}$
5	sinh
6	cosh
7	tanh

## Scalar Monadic Functions

These functions accept a numeric argument of any shape and return a result of the same shape. Each element in the result is created by operating on the corresponding element in the argument.

$-B$	Reverse the sign of B.
$\neq B$	The reciprocal of B.
$-B$	-1 if $B < 0$ , or 0 if $B = 0$ , or 1 if $B > 0$ .
$*B$	The number "e" raised to the B power.
$\mu B$	The absolute value of B.
$?B$	A random number selected from the first B integers.
$\beta B$	The smallest integer greater than or equal to B.
$\partial B$	The largest integer less than or equal to B.
$\prod B$	The base "e" logarithm of B.
$!B$	Factorial of B (positive integers only).
$\emptyset B$	Pi times B.

## Relational Functions

These functions perform logical operations and return **boolean** results. If the relation holds, the number 1 is returned. If it does not, the number 0 is returned. Except for " $\neq$ ", these are all scalar functions which have the same syntactic rules as the scalar functions above. The functions " $=$ ", " $\bullet$ ", and " $\neq$ " accept any character or numeric arguments, but they must be both character or both numeric. The functions " $<$ ", " $\leq$ ", " $>$ " and " $\geq$ " accept any numeric arguments. The other functions perform boolean logic and only accept boolean values as arguments (numeric values of 0 or 1). Note that picture variables work well with these functions, since pictures are treated as boolean arrays.

$A=B$	A is equal to B.
$A\bullet B$	A is not equal to B.
$A<B$	A is less than B.
$A\leq B$	A is less than or equal to B.
$A>B$	A is greater than B.
$A\geq B$	A is greater than or equal to B.
$A^a B$	A Or B. Returns 1 if either argument is 1.
$A^o B$	A And B. Returns 1 if both arguments are 1.
$A\cdot B$	A Nor B. Returns 1 if both arguments are 0.
$A,B$	A Nand B. Returns 1 unless both arguments are 1.
$\dagger B$	Not B. Returns 1 if the argument is 0.
$A\text{' }B$	Membership. The result is the same shape as B. An element of the result is 1 if it appears anywhere in argument A.

These functions each have different syntactic rules which are described below. In the examples, "====" means "the same as".

Examples: 5@6 “ 6 6 6 6 6 and 2 4@'ThisText' “ This

Text

<code>A%B</code>	Take - For scalar A and vector B, returns the first A elements of B. If A is negative, return the last A elements. If A is larger than the number of elements, pad with additional 0s or spaces. Extends to higher order B arrays by specifying a vector A with as many elements as the dimensions in B.
------------------	--

**A~B** Drop - For scalar A and vector B returns all but the first A elements of B. If A is negative, return all but the last A elements. If A is larger than the number of elements, return an empty vector. Extends to higher order B arrays same as the Take function.

**A^B**      Iota - Return a result the same shape as B with integers specifying the result of a search of A. A must be a vector of the same type as B (characters or numbers). Each element of the result is determined by searching A for an element with the same value as the corresponding element of B. If one is found, the index of the first match in A is returned. If none is found, an index one higher than the last element in A is returned. Note the results are  $\neg$ IO sensitive.

**A?B** Deal - A and B must be scalar integers, and A must be  $\leq$  B. Return an vector of length A containing values from  $1:B$  without duplication.

**A/B** Compression - A must be a boolean vector the same length as the last dimension in B. The result is the same as B, except elements or columns corresponding to 0s in A are squeezed out. Example  
1 0 1/'ABC' → 'AC'

**A\B** Expansion - A must be a boolean vector which contains the same number of 1s as the last dimension in B. The result is the same as B, except additional elements or columns (spaces or 0s) are inserted in the places corresponding to the 0s in A. Example 1 0 1\5 6 5 0 6

**A~B** Rotate - A must be integer scalar or vector with the same number of elements as the dimensions of B. For scalar A and vector B, A elements are rotated from the end of B to the beginning. If A is negative, rotation is in the other direction. Example 3~1 2 3 4 5 ““ 4 5 1 2 3



## Mixed Monadic Functions

These functions accept one argument and return a result. Each has individual rules of use described below.

- ⒶB**      Shape of - Returns an integer vector containing the dimensions of B. Example  $\text{Ⓐ}1\ 5\ 7 \leftarrow 3$
- ,B**      Ravel - Returns B reshaped into a vector.
- ^B**      Interval - B must be a nonnegative integer scalar. Returns a vector of B consecutive integers starting with the value in  $\neg\text{IO}$  (usually 1). Example  $\text{^}5 \leftarrow 1\ 2\ 3\ 4\ 5$
- ~B**      Reverse - Returns B in with elements or columns reversed. Example  $\text{~}^5 \leftarrow 5\ 4\ 3\ 2\ 1$
- 'B**      Monadic transpose - For vector B, returns B as a one-column matrix. For higher order array B, reorganizes B along the main diagonal.
- ÓB**      Grade up - B must be a vector. Returns an integer vector with the values in  $\text{^}B$  permuted such that if used as an index into B, the values in B would appear in ascending order (for characters, the sort order is that which appears in  $\neg\text{AV}$ ). Example: If  $A \leftarrow 5\ 3\ 7$ , then  $\text{Ó}A \leftarrow 2\ 1\ 3$  and  $A[\text{Ó}A] \leftarrow 3\ 5\ 7$  assuming  $\neg\text{IO} \leftarrow 1$ .
- ÌB**      Grade down - Same as Grade Up but computes the descending order.
- ÷B**      Monadic Format - Returns a character representation of B. Often used to convert a numeric value or array to characters for formatting purposes. Example  $\text{÷}2+3\ 4 \leftarrow '5\ 6'$  (a 1 by 3 character matrix)
- ≥B**      Execute - B must be a character vector containing a valid APL expression. The expression is executed and the final value if any is returned. Example  $2+\text{≥}'6\neq 2' \leftarrow 5$

## Operators

In APL an operator accepts a function as an argument, which produces a new function which is then applied to the arguments in the expression. The character "~" below represents any one of the following: + - ≠ \* = • °

~/B      Reduction - For vector B, the result is a scalar computed by taking the ~ function and placing it "between" each adjacent element of B and computing the resulting expression. For higher order array B, the result is the same shape as B minus the last dimension, and each element of the result is computed across the rows in B. Examples:

+/1 2 3 4	« Add up a vector
10	
X←4 5⍲^20	« Make a matrix
X	
1 2 3 4 5	
6 7 8 9 10	
11 12 13 14 15	
16 17 18 19 20	
+/X	« Add each row
15 40 65 90	
°/'This'='This'	« 1 if arg is all 1s
1	
=/2 2 2 2	« 1 if all elements equal
1	

## Special Operations

A←B      Assignment - Assign the symbol name A to the same value as B. Used to create or change the value of a variable. Variable names can be any length, upper and lower case are different, can't start with a digit but can contain digits, alphabetic characters, \_, and `.

⍋B      Branch - Give control to the function line specified by B. B can be a scalar line number, a line label, or the empty vector. If a scalar line number, control passes to that line. If there is no such line or the number is 0, the function is exited without error and control passes back to the calling function if any.. If B is a numeric vector only the first element is considered. If a line label, control passes to the named line. Lines are labeled by prefixing a symbol name and a colon to the line. In a function, line labels work like local scalar integer variables containing the line number. If B is the empty vector, no branch is performed and control falls through to the next line. You can enter a branch in immediate mode in order to resume a function which was suspended because of error or a manual interrupt. Example ⍋(A<B)/Lab will branch to the line with label Lab: only if A<B.

- ‘ Bare Branch - Exit the function and all functions that led up to it, returning you to immediate execution mode. May be entered in immediate mode in order to clear the State Indicator of suspended functions.
- « Comment - All characters to the right of this symbol up to the end of the line or the next « are ignored as a comment.
- A“¬ Quad input - Prompt the user for input by displaying "¬:" (or whatever is in ¬SF). Accept an APL expression terminated by the Return key. The expression is evaluated and the resulting value is returned to the expression with the Quad. If the entered expression is in error or does not return a value, a message is displayed and the user is prompted to enter another line. Quad can be used anywhere a variable can, it does not have to be part of an assignment statement.
- ¬“B Quad output - When a value is assigned to quad, it is displayed on the screen. This is useful to insert in the middle of an expression to observe intermediate results. Example 2+¬“3+4 will display 7 and then 9.
- A“≤ Quote Quad input - APL waits for the user to enter a line of text terminated by the Return key. The line of text (not including the Return) is returned to the expression with the Quote Quad as a character vector.
- ≤“B Quote Quad output - Like Quad output, but the display is not terminated by a newline. It is useful for sending a prompt to the user before asking for Quote Quad input.

## System Variables

These are special variables available in every workspace whose purpose is communication between you and the workspace environment. The names all start with a ¬ (quad) character. The ones marked "(read only)" ignore any attempts at being set by you; the values are always determined by APL itself. The others can be changed as specified.

- ¬ALX Attention Latent Expression. Normally an empty character vector, but if you set it to a character vector containing an APL expression, the expression will be executed instead of the normal "Attention Signalled" error that occurs when you manually interrupt execution.
- ¬AV Atomic Vector. A 256 character vector which contains all the APL characters in order. (read only)
- ¬CLIPBOARD Clipboard. If you read this variable, it will contain a character vector containing the current contents of the clipboard text, or a picture variable, if the clipboard holds a picture. If you assign a character vector or picture variable to it, the clipboard will be set to the text or picture.
- ¬CT Comparison Tolerance. Scalar real number which is usually very small, but you can set it to a value between 0 and 1. Numbers whose values differ by less than this value are considered equal for comparison purposes.

¬DM	Diagnostic Message. Character vector containing the most recent error message. (read only)
¬ELX	Exception Latent Expression. Normally an empty character vector, but if you set it to a character vector containing an APL expression, it will be executed instead of the normal error reporting actions. Used to implement your own error trapping. On manual interrupt, ¬ALX is executed unless it is empty in which case ¬ELX is checked.
¬EN	Error Number. Scalar integer code number for the most recent error. (read only)
¬IO	Index Origin. Scalar integer normally 1, but can be set to 0. Specifies the first number in the set of integers used for indexing.
¬LC	Line Counter. Integer vector containing line numbers of functions currently in execution (read only)
¬LX	Latent Expression. Normally an empty character vector, but if you set it to an APL expression and save the workspace, the expression will be executed when the workspace is loaded.
¬NLT	National Language Translation. The character vector 'ENGLISH'. (read only)
¬PORTS	Ports on the Network. Your user name is set with the Chooser desk accessory. All users currently running MacAPL on the AppleTalk network will be listed (except yourself).
¬PP	Printing Precision. Scalar integer normally 10, but can be set in the range of 3 to 19. Specifies the number of significant digits for numbers displayed using default output.
¬PR	Prompt Replacement. Normally the empty character vector but can be set to a character scalar specifying the character to be returned as the prompt portion of returned quote quad input.
¬PW	Page Width. Scalar integer in the range of 12 to 255. Lines longer than this are wrapped to the next line(s) when displayed on the screen.
¬RL	Random Link. Scalar integer in the range 0 to 65535 which is used as the base for random number generation.
¬SF	Evaluated Input Prompt. Character vector containing the prompt displayed for quad input.
¬SI	State Indicator. Character vector showing the current state indicator. (read only)
¬TS	Time Stamp. Integer vector containing the current year, month, day, hour, minute, second, and day of week. (read only)

- ¬TT      Terminal Type. Scalar integer 60 indicating Macintosh. (read only)
- ¬UL      User Load. Scalar integer indicating the number of workspaces currently open. (read only)
- ¬VERSION      Interpreter Version. Integer vector containing version, edit, and serial numbers.  
(read only)
- ¬WA      Workspace Available. Maximum number of free bytes in the workspace. Since memory is allocated in sections, a single chunk of memory this size is probably not available. The value also varies dynamically as different sections of the interpreter are moved in and out of memory. (read only)

### System Functions

- R“¬CR B      Canonical Representation. B is a character vector with the name of an unlocked function. Returns a character matrix containing the function header and lines in the function. If B is invalid, returns an empty matrix.
- R“¬DEQ B      Dequeue. B is a character vector with the name of a variable shared with a file. It removes any prior ¬ENQ lock placed on the file.
- R“¬DL B      Delay Execution. B is a scalar number indicating the number of seconds to delay. Returns B.
- R“A ¬EDIT B      Edit. A and B are character vectors. A new editor window is opened which has the title specified in A, and the text in B. Further execution is suspended while you edit the text. After editing the text, close the editor window and execution will resume with ¬EDIT returning the edited text as a character vector. You can cancel editing with the Undo menu command in which case the original text is returned.
- R“¬ENQ B      Enqueue. B is a character vector with the name of a variable shared with a file. It locks the file so another user's conflicting ¬ENQ will wait until we perform a ¬DEQ.
- R“A ¬ENQ B      Enqueue range. B is a character vector with the name of a variable shared with a file. A is a 2 element numeric vector specifying the position and size of the area in the file to lock.
- R“¬EX B      Expunge. B is a character vector with the name of a variable or defined function. The named object is expunged (erased) from the workspace.
- R“32 ¬EX B      Delete a file. B is a character vector with the name of the file.

R“¬FX B	Fix Function. B must be the representation of a function in the format returned by ¬CR. A new function is defined, or the definition of an existing function is changed according to B. The name of the defined function is returned as a character vector. If B is in improper format, a scalar integer is returned which contains the line number of a line which is in error, and the workspace is unchanged.
R“¬HIDE B	Hide Object. B is a character vector with a function or variable name. The object is hidden and 1 is returned. If B is invalid, 0 is returned. See the Hide Object menu command.
R“¬LOAD B	Load Workspace. B is a character vector containing the file specification of a workspace file. The workspace is loaded in another window and started by executing it's ¬LX if any. B can be an empty vector in which case a standard open file dialog is presented for the user to select a workspace (if the Cancel button is clicked 0 is returned). If all went well, 1 is returned. Workspace not found or trying to open too many workspaces at once causes an error.
R“¬LOCK B	Lock Object. B is a character vector with a function or variable name. The object is locked and 1 is returned. If B is invalid, 0 is returned. See the Lock Object menu command.
R“¬MAC B	Machine Language Interface. Lets you execute arbitrary 68000 machine code routines. B can be a character vector. Each byte in B is a byte of the machine code to be executed. Index into ¬AV to obtain the proper character for each byte. The code is loaded into memory and locked. MacAPL executes it with a JSR instruction. A0 will hold the address of the code. Your code must end with RTS and preserve A6 and A7. The signed 32 bit value in D0 at the time of the RTS is returned as a scalar integer by ¬MAC. For example to do a SysBeep(1) you could execute ¬MAC ¬AV[63 60 0 1 169 200 78 117+¬IO]
A ¬MSG B	Message another user. B is a character vector with a one-line message. A is the network user name of another MacAPL user on the network. Useful for sending quick messages to friends.
R“¬NC B	Name Classification. B is a character vector with a symbol name. A scalar integer is returned which depends on the current use of the symbol: 0=available for use, 1=used as a line label, 2=used as a variable, 3=used as a function, 4=illegal or unavailable.
R“¬NL B	Name List. B is an integer scalar or vector. Returns a matrix with a list of names of type(s) specified by B. 1=line labels, 2=variables, 3=functions.
¬RESET	Reset. Clears the State Indicator of any suspended functions.
R“A ¬STOP B	Set Stop Vector. A is a vector of line numbers, and B is a character vector name of an unlocked function. Execution will be interrupted just before execution of the line(s) you specify. This can be useful for debugging. While execution is suspended, variables can be

examined or changed. To resume, enter a branch statement specifying the line at which execution should be resumed. Entering `←LC` is an easy way to resume where it left off. To remove all stops, specify the empty vector as the left argument.

**R←SVC B** Get Access Control Vector. Return the 4-element boolean vector representing the current access of the shared variable named in B.

**R←A←SVC B** Apply Access Control Vector. Add access restrictions to the shared variable named in B. Access control bits: [1] Our set needs their prior access [2] Their set needs our prior access [3] Our use needs their prior set [4] Their use needs our prior set.

**R←A←SVO B** Shared Variable Offer. A specifies the partner to share with and B names the variable to share. Result is 0=share failed, 1=share succeeded but not yet accepted, 2=fully shared. Both partners must offer the same named variable to each other for a full share. Generally the offer is repeated until 2 is returned.

**R←(32;'file')←SVO B** Offer to share with a file. Empty vector 'file' brings up file selection dialog. If file opened, returns a full share (2).

**R←(36;'file')←SVO B** Offer to share with a file, creating if not found. Empty vector 'file' brings up file specification dialog.

**R←(64;'wsid')←SVO B** Offer to share with another open workspace on your own computer. Requires a matching offer for a full share.

**R←(96;'user')←SVO B** Offer to share with another MacAPL user on the network. Requires a matching offer from the user for a full share.

**R←SVR B** Shared Variable Retract. Cancel the offer to share the variable named in B. Retraction is automatic when a shared variable is erased or the workspace is closed.

**R←SVQ B** Shared Variable Query. Return a matrix of variable names which the network user named in B has offered but you have not accepted.

**R←VR B** Vector Representation. Similar to `←CR` except the returned value is a character vector with a "prettier" display of the function including line numbers, etc.

**R←XLOAD B** Load Without Executing Latent Expression. Same as `←LOAD` except the `←LX` of the loaded workspace is not automatically executed.

## Error Messages and Condition Codes

1	Operating system code -999 - The Macintosh returned an O/S error
2	Workspace not found - wrong name or directory given
3	Workspace full - not enough RAM to do this operation
4	Function definition error - invalid header or line syntax
5	Label error - improper use of a colon in a function definition
6	Syntax error - improperly formed expression
7	Index error - subscript out of range
8	Rank error - wrong number of dimensions in array
9	Length error - wrong number of elements in an array dimension
10	Value error - undefined variable or missing function argument
11	Range error - value out of permitted bounds
12	Domain error - wrong type of data for this function
13	Attention signaled - execution interrupted by user
14	End of file encountered - attempt to read past end of data file
15	File not found - wrong name or directory for data file
16	Timeout - waited too long for data
17	Feature not available in this version of MacAPL
18	Resumed after a Macintosh system error
19	Can't open another workspace - close one first
20	Can't open another edit window - close one first
21	Internal confusion (stack) - internal MacAPL interpreter error
22	Illegal character in line - only permitted within quoted string
23	Symbol name error - character can not appear in a symbol name
24	File is not in proper format - saved workspace file is corrupted
25	Object is locked - attempt to change value of a locked function
26	Not with suspended functions - do <code>↵RESET</code> before retrying
27	The disk is locked - write protect tab is set on the disk
28	The file is locked - unlock with Finder get info command
29	The disk is full - no more room on the disk
30	Disk I/O error - physical media error
31	Feature not available in the runtime version of MacAPL
32	Stop condition - set with <code>↵STOP</code> function
33	Trace condition - set with <code>↵TRACE</code> function
34	Network is unavailable - AppleTalk off, or serial port already in use
35	Network link limit exceeded - Can't share another variable
36	Network link lost - Can't link with the user sharing the variable
37	Network user not found - No MacAPL user on the net with that name