Honeywell

APL USER'S GUIDE

MULTICS

SUBJECT:

Introductory User's Guide to Multics APL.

SOFTWARE SUPPORTED:

Multics Software Release 1.0

DATE:

January 1974

ORDER NUMBER:

AK95, Rev. 0

PREFACE

The organization of the manual follows the organization of the APL processor. Section I presents an introduction to APL. Section II discusses the invocation of the processor and the character set conventions necessary to communicate with it. Section III discusses the APL language. Section IV explains the APL system commands. Section V discusses the use of the APL editor to create and manipulate stored programs.

The remainder of the manual contains material of value to certain classes of users. For new users, Section VI presents an annotated reproduction of a sample demonstration APL session, containing the most basic features of the language. For users already familiar with APL/360, Section VII identifies the differences between APL/360 and Multics APL.

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

TABLES

SECTION I

INTRODUCTION

HISTORY OF APL

A Programming Language (APL) originated as a mathematical notation for the discussion of the theory of algorithms. It was invented by Dr. Kenneth E. Iverson and was described by him in his book, A Programming Language *. The value of the notation as a practical means for expressing an algorithm to a computer was soon noticed. An interpreter which realized a subset of the notation was developed by IBM on its 7090 computer. The success of this pilot interpreter led to a second and more powerful implementation, known as APL/360, on the IBM 360.

The success of APL can be attributed to some characteristics which distinguish it from more conventional programming languages. First, it is interactive by design rather than by decree--it is fast, succinct, forgiving, informative, and even fun to use. Next, it is at once both simple and powerful--it is easy to learn, transparent, yet it attacks abstruse problems with ease.

Multics APL is designed to behave as much like APL/360 as possible, to minimize the learning effort required of those already familiar with APL/360, and to promote compatibility at the source language level with other APL installations.

*John Wiley and Sons, 1962

CHARACTERISTICS OF APL

APL can be characterized as a line-at-a-time desk calculator with many sophisticated operators and a stored-program capability. The user needs little or no prior acquaintance with digital computers to use it. After invoking APL, the user types an expression to be evaluated. The APL interpreter performs the calculations, prints the result, and awaits a new input line. The result of an expression evaluation can also be assigned to a variable and remembered from line to line. In addition, there is a capability for storing input lines by an assigned name, so that a later mention of the name causes the lines to be recalled and interpreted as if they had been entered from the terminal at the time. Finally, there is the ability to save the entire state of an APL session, complete with all variable values and stored programs, so that the user may continue at a subsequent APL session.

The APL language uses its own specially designed character set, in which each operator is represented by a single character. The most convenient access to APL is via a Selectric-type terminal with the APL typing element (type ball) mounted. Multics APL is also usable from any ASCII terminal as well, although the user must be aware of the typing conventions used to represent some of the APL characters within the framework of the available ASCII graphics.

The Multics APL processor consists of three principal components: the interpreter for the mathematical expressions of the APL language, a system command processor which provides bookkeeping aids and an environment within which the language runs, and an editor which is used to create and modify stored APL programs.

SECTION II

COMMUNICATING WITH MULTICS APL

CALLING THE APL COMMAND

To call Multics APL, issue the command line:

```
apl
```

APL responds by typing six spaces before awaiting input from the user. This informs the user that APL would like to hear from him and improves the readability of the terminal listing. All of the user-typed lines will appear indented by six positions while all of the APL-generated responses will begin at the left margin.

Before typing any input, however, the user must determine how the APL character set is represented on his terminal. Since the APL character set differs significantly from the Multics standard character set, normal Multics typing conventions do not apply to communication with APL.

APL CHARACTER SET

In contrast to the 94 graphics of the Multics standard character set, the APL character set has 130 graphics. APL graphics are shown in Table 2-1, together with their internal codes.

The internal code assigned to each character is not normally of significance to the APL user. There is no way within the APL language to discover or make use of the internal representation of a character--for example, there is no collating sequence. However, there are occasions in Multics such that lines originating in APL are transferred out to the rest of Multics, or vice versa; in these instances the exact internal codes used by APL become significant. In this connection, the established code assignments agree with the Multics standard code assignments wherever any correspondence of graphics between the two character sets can be found.

Table 2-1. APL Character Set

Graphic	Code	ASCII	APL Selectric	Graphic	Code	ASCII	APL Selectric
!	041	!	(os) ' .	B	142	b	B
(es)	042	"	¨	C	143	c	C
(er) ω	043	#	ω		...		
⍳	044	$	⍳	X	170	x	X
⍴	045	%	⍴	Y	171	y	Y
×	046	&	×	Z	172	z	Z
'	047	'	'	←	173	{	←
(	050	(	(	|	174	¦	|
)	051	)	)	→	175	}	→
*	052	*	*	~	176	~	~
+	053	+	+	≤	200	(os) < -	≤
,	054	,	,	≥	201	(os) > -	≥
-	055	-	-	≠	202	(os) = /	≠
.	056	.	.	∨	203	"or	∨
/	057	/	/	∧	204	"an	∧
0	060	0	0	÷	205	(os) - :	÷
1	061	1	1	∊	206	"ep	∊
2	062	2	2	↑	207	"up	↑
3	063	3	3	↓	210	"do	↓
4	064	4	4	○	211	"ci	○
5	065	5	5	⌈	212	"ce	⌈
6	066	6	6	⌊	213	"fl	⌊
7	067	7	7	∆	214	"de	∆
8	070	8	8	∘	215	"cc	∘
9	071	9	9	⎕	216	"qu	⎕
:	072	:	:	∩	217	"ca	∩
;	073	;	;	⊥	220	"ev	⊥
<	074	<	<	⊤	221	"en	⊤
=	075	=	=	⊂	222	"in	⊂
>	076	>	>	⊃	223	"co	⊃
?	077	?	?	∪	224	"cu	∪
(ki) α	100	@	α	⍱	225	"no	(os) ∨ ~
A̲	101	A	(os) A _	⍲	226	"na	(os) ∧ ~
B̲	102	B	(os) B _	⊖	227	"rf	(os) ○ -
C̲	103	C	(os) C _	⌿	230	(os) / -	(os) / -
	...			⍫	231	"lf	(os) ∇ ~
X̲	130	X	(os) X _	⍟	232	"lo	(os) ○ *
Y̲	131	Y	(os) Y _	⌽	233	"rr	(os) ○ |
Z̲	132	Z	(os) Z _	⍉	234	"tr	(os) ○ \
[	133	[	[	⍒	236	"gd	(os) ∇ |
\	134	\	\	⍋	237	"gu	(os) ∆ |
]	135	]	]	⍝	240	"la	(os) ∩ ∘
_	136		_	⍞	241	"qq	(os) ⎕ '
	137			⌶	242	"ib	(os) ⊤ ⊥
∇̄	140	\̄	∇̄	⍀	243	(os) \ -	(os) \ -
A	141	a	A	⌹	244	"mi	(os) ⎕ ÷

Table 2-1 (cont). APL Character Set

NOTE: The following abbreviations are used in the table:

(es) - escape character	(os) - backspace and overstrike each following character
(er) - erase character	
(ki) - kill character	

In addition to the above graphics, all the Multics control characters are members of the APL character set with their usual octal codes.

010	backspace
011	horizontal tabulate
012	newline (carriage return & linefeed)
040	blank

The problem arises of representing this character set on various terminals. Multics APL is designed to be usable from two general classes of terminals: Selectric-type terminals with interchangeable typing elements (type balls), like the IBM 1050 and 2741; and ASCII terminals which do not have interchangeable graphics, such as Teletype Model 37 and GE TermiNet 300 or Honeywell SRT301.

Selectric-type Terminals

In the case of Selectric-type terminals, APL assumes that the user has mounted the APL type ball (IBM part number 1167988). This is the most convenient way to access APL, as the entire APL character set is made available with a minimum of special typing conventions.

The fourth and eighth columns of Table 2-1 show how each character of the APL set is represented on a Selectric-type terminal. Most characters can be typed with one keystroke; however, several characters must be produced by backspacing and overstriking other graphics. The abbreviation "(os)" in Table 2-1 means that the subsequent graphics must be overstruck to produce the desired character. Characters generated by overstriking graphics are nevertheless considered to be single characters internally.

ASCII Terminals

Using ASCII terminals is somewhat more complicated. The third and seventh columns of Table 2-1 show how each APL character is represented on an ASCII terminal. While most characters can be typed with one keystroke, a few are represented by more complicated correspondences: some are produced by backspaces and overstrikes, and many have defined for them mnemonic escape sequences.

As in the case of Selectric-type terminals, characters produced by backspaces and overstrikes or escape sequences are considered single characters.

Canonicalization

As soon as backspaces are allowed in any typed line, it becomes evident that there are many different ways to type a given line. That is, there are many different sequences of keystrokes that produce visually identical results. To reduce confusion and allow greater freedom to the typist, APL canonicalizes each input line as it is read. This means that the characters typed by the user are sorted into their visual order on the page, independently of the temporal order in which they were typed. Hence, the user need not bother to type overstrikes in any specified order.

A more complete explanation of APL canonicalization process is given under "Details of APL Input Line Processing," later in this section.

Erase and Kill Processing

Typing errors in Multics APL are corrected through the mechanism of erase and kill characters rather than by backspacing and hitting the ATTN button as in APL/360. The kill character is the alpha ⍺ (which is represented as commercial at (@) on ASCII terminals), and the erase character is the omega ⍵ (which is represented as the number sign # on ASCII terminals).

The kill character removes the entire line preceding it. That is, the kill character deletes itself, anything overstruck with it, and all characters to the left. Characters to the right of a kill character are not deleted.

The definition of erase is a little more complicated. If the erase character is overstruck with anything, then only that print position is removed. But if the erase character appears alone in a print position, then the character in the preceding print position is removed as well. If there is no character in the preceding position (i.e., it is white space), then the entire white space or carriage motion preceding the erase character is deleted.

Since erase and kill are performed <u>after</u> canonicalization, the spatial positions of the characters on the line determine which characters are removed; i.e., the order in which the characters were typed is not significant.

In Multics APL, kill is performed before erase; it is not possible to erase a kill character.

Table 2-2. Mnemonic Escape Sequences

Escape	Code	APL Meaning	Escape	Code	APL Meaning
"al	100	⍺ alpha	"in	222	⊂ included
"an	204	∧ and	"la	240	⍝ lamp
"ba	174	| bar	"lf	231	⍫ locked function
"ca	217	∩ cap	"lo	232	⍟ logarithm
"cc	215	∘ center-circle	"mi	244	⌹ matrix inverse
"ce	212	⌈ ceiling	"na	226	⍲ nand
"ci	211	○ circle	"no	225	⍱ nor
"co	223	⊃ contains	"om	043	⍵ omega
"cu	224	∪ cup	"or	203	∨ or
"de	214	∆ delta	"qq	241	⍞ quote-quad
"do	210	↓ down arrow	"qu	216	⎕ quad
"en	221	⊤ encode	"rf	227	⊖ reverse-first
"ep	206	∊ epsilon	"rr	223	⌽ reverse/rotate
"es	042	¨ escape	"ti	176	~ tilde
"ev	220	⊥ evaluate	"tr	234	⍉ transpose
"fl	213	⌊ floor	"up	207	↑ up arrow
"gd	236	⍒ grade down			
"gu	237	⍋ grade up			
"ib	242	⌶ I-beam			

<u>Escape Processing</u>

An escape mechanism is provided in order to allow any arbitrary character or sequence of characters to be entered in spite of canonicalization, erase, and kill. The escape character is the diaeresis (¨) (represented as double quotation marks (") on ASCII terminals). The escape character is followed by (1) another escape character, which represents a diaeresis as data in the input line without performing an escape function; or, (2) a one-, two-, or three-digit octal number, which represents a single character of precisely that internal code; or (3) one of the two-character mnemonic escape sequences listed in Table 2-2. While escapes are necessary primarily for users of ASCII terminals, the same escapes are accepted from Selectric-type terminals.

Since erase and kill are performed before escape processing, it is possible to erase and kill the various characters of an escape sequence. For example, an escape followed by an erase results in an erasure of the escape character rather than the input of an erase character (an erase character can be input either with an octal escape or with its mnemonic escape ¨ *OM*).

Details of APL Input Line Processing

The following paragraphs contain a more detailed explanation of input line processing.

Initially, APL obtains its input from the Multics stream "user_input". Upon the occurrence of any error, APL reverts to reading the stream "user_i/o". In either case, the following initialization is performed. APL makes an "info" order call in an attempt to determine the type of terminal to which the stream is attached. If there is no terminal, an unknown type of terminal, or if the "info" order is rejected, then APL assumes that there is no terminal connected. In this case, the input is used directly as read, and no code conversion, canonicalization, erase, kill, escape, or overstrike processing is performed. In other words, the input stream in this case must already consist of correct Multics APL internal character codes. (The APL function editor and its *W* request is a convenient way to generate a file suitable for input to APL.)

If it is recognized that the input stream is connected to a Selectric-type or ASCII terminal, then processing appropriate to the device is selected. In order that APL be able to do its own line processing, a "modes" order call is issued to the Multics ring-zero Teletype device interface module to turn off standard canonicalization, erase, kill, and escape processing. The former mode is remembered for subsequent restoration.

Whenever APL is left and re-entered by the)*E* system command or the *E* editor request, the remembered mode is restored before the callout. Upon return, the attachment of the input stream is again inspected for possible changes and the appropriate handling for it is selected. If the input stream attachment is disturbed without notifying APL (e.g., when the user quits out of APL), then APL continues to apply the same line processing as was selected for the old attachment, leading to possibly confusing or inconvenient results with the new device.

Assuming that the terminal was properly identified, the first processing of the input line (a _line_ is a sequence of characters ending with a new-line character) is conversion of each input character into the APL internal character code. Each graphic is then flagged with its color (it can be red or black) and its position on the printed page. The nongraphic (carriage-motion) characters are ignored except for their contribution to the positioning of the graphic characters. Each position consists of the following: subline, tab stop, and print position within tab stop (sublines are separated by vertical-tab and form-feed characters; tab stops are separated by horizontal-tab characters). The graphics are then canonicalized (sorted into ascending spatial order, i.e., the order in which they are seen on the printed page). Within a single print position, overstruck graphics are sorted into ascending order according to the APL internal character code.

Kill processing is performed next. If any print position contains a kill character, it and all positions to the left of it are eliminated. Since this elimination is done after canonicalization, some of the characters eliminated may have been typed before the kill character and some may have been typed after it; what counts is the position of each character on the printed page.

Erase processing is performed next. If any print position contains an erase character and any other graphic, then that print position is eliminated. If the erase character is alone in its print position, then the preceding print position or carriage motion is also eliminated. Nothing on a previous subline can be erased by erase characters. Since kill is performed before erase, kill characters cannot be erased.

Escape and overstrike processing are performed next, both at the same time. Since canonicalization, erase, and kill have already been performed, carriage motions, erase, and kill can be used to properly format escapes and overstrikes. For example, erase can be used to correct individual characters in escape sequences, or to erase the escape character itself. Escaped-in erase, kill, or escape characters are treated as data since no further erase, kill, or escape processing follows. Also, backspaces and other carriage motions generated by the escape feature are treated as data and do not influence either the overstrike processing or the canonicalization.

Finally, the last step consists of inserting new carriage-motion and color-shift characters to make all the graphics print with proper color in their correct print positions.

LINE LENGTH 256

In Multics APL, the longest input line that can be processed is 256 characters. When a line to be interpreted consists of several typed lines because of newline characters imbedded in character constants, the total interpreted line must fall within this limit.

PROGRAM INTERRUPT IS ENABLED

Whenever Multics APL is running, it provides the user a handler for the "program_interrupt" condition. If the user desires to interrupt APL, he presses the QUIT button and types the "pi" command. APL then sets a timer to give the current calculation a short grace time in which to complete. If the end of a line to be interpreted is encountered before the timer expires, APL suspends operation, types six spaces, and awaits input from the terminal. If the timer expires before the end of a line is reached, the state of the computation reverts to the previous console suspension point. APL then types six spaces, and awaits input. In either case, the user can use the)*SI* or)*SIV* system commands to determine where his computation was suspended. The jump operator → can be used to restart the computation.

USING APL

When APL has been invoked and awaits input the user may enter one of three types of input: an expression to be evaluated immediately, a system command, or an invocation of the function editor.

An expression to be evaluated immediately is the most common response. This entry initiates computations. This class of input is discussed in Section III.

A system command interrogates or adjusts the environment in which the computations are performed. Most system commands are attendant to bookkeeping functions such as listing names of current variables and functions, erasing variables or causing APL to return to Multics. System commands are discussed in Section IV.

Finally, the function editor is used to store functions, or programs of APL lines, for later execution, rather than to execute each line as it is typed. The editor provides a means to create, modify, and replace stored function definitions. The function editor is discussed in Section V.

SECTION III

THE APL LANGUAGE

WORKSPACES

A workspace is that area of computer memory set aside for the APL interpreter to remember everything it must keep track of during a session. An APL workspace contains: (1) a name table listing the names of all variables, functions, and groups that have been defined; (2) the values of all variables and the definitions of all functions and groups; (3) a state indicator, which records the instantaneous state of functions currently in execution; and (4) a few occasionally referenced static parameters, such as page width and index origin.

When the APL command is issued, the active workspace is initially clear; that is, it contains no variables, functions, or groups, and it contains some default values for the static parameters.

APL system commands exist for saving the contents of the active workspace as a Multics segment, or for reloading the workspace from a previously saved copy. This permits an APL session to be interrupted and saved, and then taken up again on a later date with no loss of information. This also permits a user to maintain several saved workspaces, each applicable to some separate task, and to take them up in turn as desired.

VALUES

In APL, the value of any variable or expression is a rectangular array of elements, each of which is a single character or a single number. The array can have any number of dimensions from zero up, and the extent of each dimension can be anything from zero up. The number of elements in the array is equal to the product of the dimension extents. Character and numeric elements cannot be mixed within the same value.

Type

Three salient characteristics of an APL value are its type, its rank, and its shape. The type of a value is either character or numeric, depending upon whether its elements are characters or numbers. Remember that characters and numbers cannot both appear in one value. Type is important in that some operations demand operands of a specific type.

Internally, the APL interpreter further classifies numeric values into three subtypes: bit, integer, and double-precision floating-point. However, these internal subtypes are invisible to the user, and conversions between them are made automatically by the interpreter.

An array with no elements at all (the so-called null value) is acceptable to operations demanding either character or numeric operands. It may be thought of as having either type, as the occasion demands.

Rank

The rank of a value is the number of its dimensions. A scalar has rank zero, and consists of but a single element. A vector has rank one, and consists of a number of linearly ordered elements. Matrices of rank two and arrays of higher rank are also permitted.

Shape

The shape of a value is its set of dimension extents, expressed as a vector. A scalar, having no dimensions, has a null shape vector; i.e., a vector with no elements. A vector has a shape vector that consists of a single number, the number of elements in the vector value. Matrices of rank two have shape vectors with two elements. Arrays of higher rank have shape vectors of length equal to their rank.

Since the shape of a value is a numeric vector, it itself is an APL value. The shape operator ⍴ provides a way to extract the shape of a value. For example if *A* is a 5 by 2 by 4 character array, then ⍴*A* is a numeric vector of three elements with value 5 2 4. The shape of the shape is again an APL value; in fact, it is the rank of the original array. Thus, ⍴⍴*A* would be 3 in the current example.

Output of Values

Two workspace parameters directly affect the output of values. They are the page width and number of digits of precision. The page width is the maximum number of character positions per line that the interpreter will fill when typing output. It is set by default to 80 characters, but it can be changed by using the *)WIDTH* system command or the *WIDTH* library function. The number of digits of precision is the number of significant decimal digits that are to be displayed when numbers are output. Numbers are rounded to this precision before printing. This precision does not affect the accuracy with which internal calculations are carried out; it affects only the final printing of answers. The initial precision is 10 decimal digits, but it can be changed by the *)DIGITS* system command or the *DIGITS* library function.

A character element is output simply as the single character which it is; it is not placed within quotes or otherwise altered.

A numeric element is output in the simplest representation possible in the decimal notation. Positive signs are omitted, negative signs are printed as the upper minus, ¯. Magnitudes are displayed rounded to the current workspace precision, with trailing zeroes suppressed. If the magnitude is very large or very small, the interpreter may choose to output the number in scientific notation, which consists of the digit string, the letter *E* (for "exponent"), and an integer which is the power of ten by which to multiply the digit string to obtain the true number being represented. For example, Boltzmann's constant in joules per degree Kelvin, which is 1.38 times 10 to the -23 power, would be printed by APL as 1.38*E*¯23.

A vector of character elements is output as a character string, with no extra blanks or other separators intervening between the elements. Of course, the elements themselves may have as values the characters blank, tabulate, newline, etc. If a character vector is longer than the page width, as many elements as possible are printed on the first line and then the excess elements overflow to a following line, indented by six spaces. As many overflow lines as necessary are inserted to print all the elements of a very long vector.

In a vector of numeric elements, each element is set off from the preceding one by two blanks. As with character vectors, if one line is insufficient in width to accommodate all the elements, the excess elements are placed on a succeeding line or lines, each indented by six spaces.

Matrices and higher-rank values are printed in rectangular planes, with decimal points lined up in rows and columns. Each plane is preceded by a blank line. As many planes as necessary are printed to output the entire array. For example, the output of a 5 by 2 by 4 by 3 character array consists of ten planes, each consisting of a blank line and four lines of three characters each. If the page width is insufficient to hold even one line of the output, then the excess elements from each line are moved to inserted lines, indented as usual. An array with no elements prints simply as a blank line.

A consequence of this output format is that it is impossible to distinguish a scalar from a one-element vector by means of their printed values. It is also impossible to distinguish the exact rank of higher-dimensional arrays when they consist of a single element, or have no elements at all. In cases where the shape of a value must be known precisely, the shape operator ρ can be used to explicitly extract its shape.

Some examples of the output of values will be found following the discussion of value input below.

Input of Values

A scalar character element is input by typing the desired character between a pair of quote marks. Between quote marks, if it is desired to represent a quote mark itself, the quote mark must be typed as two quote marks. Thus, the input of a character element whose value is to be a single quote mark consists of four quote marks, two to bound the element value and two to represent the single quote being entered. Blanks, tabulates, newlines, and any other legal APL character, including the ones constructed from overstrikes or escape sequences, can be entered between quote marks. A character produced by overstrikes or escapes is considered a single element internally.

A scalar numeric element is input by typing the upper minus sign ¯ if negative, or no sign if positive, and a string of decimal digits optionally containing a decimal point. Scientific notation can also be used for input, in which case the digit string is followed by the letter *E* and the desired decimal exponent expressed as an integer. No blanks are permitted within the representation of a single numeric element. In the final value input to the workspace, no record is retained of the way in which a number was typed; for example, all the following inputs result in the very same internal value: `1 0001.00 0.01E2 1000E¯3`.

Note that APL distinguishes the minus sign (¯) from the subtraction operator (-). The subtraction operator is not permitted within a constant.

A vector of numeric elements is input by typing its members, separated by one or more spaces or tabs.

A vector of character elements can be input in either of two ways. The individual character elements, each enclosed by a pair of quote marks, can be typed separated by one or more spaces or tabs. Alternatively, the entire string of elements can be typed between one pair of quote marks. Within such a string, any legal APL character can appear as an element. To represent the quote character as one of the elements of the string, it must be typed as two quote characters.

Arrays of rank higher than one cannot be input directly. Such values must be constructed by entering their elements as vectors and then using the dyadic reshape operator ρ to reshape them to the desired dimensions, filling in the supplied elements in row-major order. For example, the input `2 3ρ1 2 3 4 5 6` is an expression whose value is a 2 by 3 array of numbers from one to six.

```
      13.49                    Examples of input and output.
 13.49
      39.2 ¯14                 Use of the upper minus sign to
 39.2  ¯14                     enter a negative element.
      39.2 -14                 The minus sign is not the same
 25.2                          as the subtraction operator.
      1 0001.00 0.01E2 1000E¯3
 1  1  1  1
      'DON''T'
 DON'T
      'D' 'O' 'N' '''' 'T'
 DON'T
      2 3ρ'WHYNOT'              A 2-by-3 matrix built
                                from a vector constant by use
 WHY                            of the reshape operator.  A
 NOT                            blank line precedes each plane
      2 2ρ1003 32 ¯28E¯1 416   of array output.

  1003      32
   ¯2.8    416
      0.000000000000000000000138
 1.38E¯23
```

NAMES

Names are used within APL for naming variables, functions, and groups. A name consists of an alphabetic character followed by any number of alphabetic or numeric characters. For the purposes of this rule, the alphabetic characters are considered to be

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

and

$\underline{A}$ $\underline{B}$ $\underline{C}$ $\underline{D}$ $\underline{E}$ $\underline{F}$ $\underline{G}$ $\underline{H}$ $\underline{I}$ $\underline{J}$ $\underline{K}$ $\underline{L}$ $\underline{M}$ $\underline{N}$ $\underline{O}$ $\underline{P}$ $\underline{Q}$ $\underline{R}$ $\underline{S}$ $\underline{T}$ $\underline{U}$ $\underline{V}$ $\underline{W}$ $\underline{X}$ $\underline{Y}$ $\underline{Z}$

and also _ itself. The numeric characters are 0 1 2 3 4 5 6 7 8 and 9. All characters of a name are significant. Names can be of any length. In APL statements, at least one space or tabulate must appear between consecutive names or numbers in order to separate them. Spaces and tabulates are optional between any other two constituents of statements.

A variable is simply an APL value that has been given a name. Unlike most programming languages, APL requires no declarations of names. A variable is created by merely assigning a value to a name (values are assigned with the assignment operator ←) as discussed under "Other Operators and Symbols", later in this section. Variables are not restricted to values of specific type or dimensions; any variable can take on any APL value. When a new value is assigned to a variable, any previous value possessed by that variable is discarded. An attempt to assign a value to a group or function name is an error, as a name can refer to only one object at a time. A reference to the value of a variable before it has ever been assigned one is also an error.

A function is a stored APL program. It consists of any number of APL lines to be interpreted, plus a header line which specifies some important properties of the function. A function is created and altered with the APL function editor, as described in Section V. When the name of a function is included in an expression being interpreted, the stored lines are brought forth and executed, much as if they had been typed in place of the name.

A group is a list of names of other objects. Grouping the objects allows them to be copied and erased as a unit, without repetitively typing their individual names. Groups have no significance other than in the copy and erase system commands.

The names of all objects in a workspace can be listed with the)*VARS*,)*FNS*, and)*GRPS* system commands. Any variable, function, or group can be deleted from a workspace with the)*ERASE* system command. System commands are discussed in Section IV.

OPERATORS

Values can be built up into complicated expressions by operating upon them with operators. APL has a great number of operators, most of them borrowed from the language of everyday arithmetic, algebra, and related areas of mathematics.

Some operators operate on only one operand, (e.g., negation, absolute-value, reciprocal), while others operate upon two (e.g., addition, subtraction). An operator taking one operand is said to be monadic. In APL, a monadic operator is always written before (to the left of) its operand, which becomes its right, and only, operand. An operator taking two arguments is said to be dyadic. A dyadic operator is always written between its left operand and its right operand.

Some APL operators are inherently monadic or inherently dyadic, but others can be used either way with a corresponding slight change in the definition of the operator. For example, the subtraction sign represents the subtraction operation when used dyadically, as *A-B* is the value of A minus the value of B; but it represents the negation operation when used monadically, as *-B* is the value of B negated (algebraically changed in sign).

Scalar Operators

A scalar operator is one defined as acting on a single scalar element as operand (when used monadically) or on a single pair of scalar elements (when used dyadically). As discussed below, it is possible by observing certain restrictions to provide an array as an operand of a scalar operator, but, in this case, the scalar operation is still defined only in terms of its action on the individual elements of the array, independently of each other. In other words, a scalar operator applied to an array merely extends its action to each individual element of the array. This is in distinction to a mixed operator, which accepts an entire array at once as its operand, and which performs some action on the whole array at once, an action in which the elements of the array cannot be considered independently of one another.

Examples of scalar operations include addition, subtraction, absolute-value, logical AND, and logical OR.

Examples of mixed operations include matrix transposition, matrix inversion, reshaping, and sorting.

The sense in which scalar operators extend to operate upon the elements of arrays will now be clarified. First, monadic operators present no difficulty: if a monadic operator is applied to an array, the result of the operation is an array of identical rank and dimensions, and each element of the answer is the result of applying the scalar operation independently to the corresponding element of the operand. For example, if the value of *A* is a numeric vector of six elements, then the value of -*A* is again a numeric vector of six elements, each element having had its sign changed from that of the operand vector.

The case for dyadic scalar operators is slightly more complicated because now there are two operands to be considered. If a scalar operator is applied to two operands of identical rank and dimensions, then the answer is again an array of the same rank and dimensions, and each element of the answer is generated by applying the operation to the two corresponding elements of the operands. For example, two identical length vectors may be added, element by element; `6 2 3+1 4 7` gives the result `7 6 10`.

If a scalar operator is applied to two arrays that fail to match in rank and dimensions, and one of the arrays consists of only a single element, then the single element is considered to be replicated to the rank and dimensions of the other operand, and the operation proceeds element-by-element as before. In other words, the single element participates with each element of the other operand in turn, producing a result identical in rank and dimensions to the other operand. For example, if the value of *A* is a 2 by 3 matrix of integers, then the value of 6+*A* is a 2 by 3 matrix of integers each six greater, because the single element 6 is applied independently to each of the elements of *A*.

If both operands consist of only a single element, then the rank of the result is arbitrarily taken as the rank of the right operand. For example, if *A* has the value of a 1 by 1 array consisting of the number 103 and *B* has the value of a scalar (rank 0) number 88, then *A*-*B* has the value scalar 15. In contrast, the value of *B*-*A* would be the 1 by 1 matrix ¯15.

The final case which must be considered is that of a scalar operator applied to two operands which do not match in rank and dimensions, and in which neither operand consists of only a single element. In this case, the operation is in error, and the APL interpreter suspends operation and issues a diagnostic message. The message is *RANK ERROR* if the two operands do not match in rank, or else *LENGTH ERROR* if they match in rank but some corresponding pair of dimension extents do not match. Error reporting and possible recovery actions are discussed later in this section.

ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION + - × ÷

When used dyadically, the operators + - × and ÷ represent the arithmetic operations of addition, subtraction, multiplication, and division. Unlike some programming languages which truncate quotients of integers to an integer, APL retains the fractional part of a quotient as accurately as the hardware permits (approximately 19 decimal digits).

A *DOMAIN ERROR* occurs when an attempt is made to divide by zero or when the result of an operation exceeds the capacity of the hardware to represent numbers (the largest magnitude representable is 1.7014118346046923l7E38).

```
      6.3+21.4                    Examples of dyadic + - × and
 27.7                             ÷.
      6 2 ¯7-¯1 2 ¯3              The minus operator as distin-
 7  0  ¯4                         guished from the negative
      3×13.9 ¯1 0.11E4            sign.
 41.7  ¯3  3300
      360000÷0.00003
 1.2E10
      2E20×3E30                   6E50 is beyond the capacity of
 DOMAIN ERROR.                    hardware.
      2E20×3E30
          |
      15÷7                        Quotients are not truncated to
 2.142857143                      integers.
      1 2×3 4 5                   The vectors are not the same
 LENGTH ERROR.                    length.
      1 2×3 4 5
         |
```

PLUS, NEGATION + -

The monadic + operator leaves its numeric operand unchanged.

The monadic - operator represents negation; that is, algebraic change of sign of its operand.

```
      +¯1 2 ¯3
¯1  2  ¯3
      -¯1 2 ¯3
 1  ¯2  3
```

SIGNUM ×

The monadic × operator represents the mathematical signum operation; that is, 1 if its operand is greater than zero, 0 if its operand is 0, and ¯1 if its operand is less than 0.

A number is considered equal to 0 if it is within a certain tolerance of 0. This tolerance is called fuzz. Fuzz is discussed in Section IV.

```
      ×0 ¯1 2 ¯3
 0  ¯1  1  ¯1
```

RECIPROCAL ÷

The monadic ÷ operator finds the reciprocal of its right operand. An attempt to extract the reciprocal of 0 causes a *DOMAIN ERROR*.

```
      ÷¯1 2 ¯3
 ¯1  0.5  ¯0.3333333333
```

EXPONENTIAL, LOGARITHM * ⍟

A raised to the B power is expressed in APL as *A*B*. The logarithm of B to the base A is expressed as *A⍟B*. Note that the base is the left operand in the definition of both operations. If the base is omitted (monadic usage of the operators), the natural logarithm base 2.718281828... is used. Thus, *1 is the natural logarithm base itself.

There are no square-root or cube-root operators in APL, so the exponential operator is used to perform these operations. For example, the square root of A can be expressed as *A*0.5* or as *A*÷2*. Since APL does not handle complex numbers, any attempt to extract an even root of a negative number results in a *DOMAIN ERROR*.

```
      10*1 2 3 4
 10  100  1000  10000
      2 5 10⍟10
```

```
3.321928095  1.430676558  1
      ⍟10
2.302585093
      *1 2.302585093
2.718281828  10
      ¯8*÷3
¯2
```

MODULO |

The vertical bar | used dyadically represents the modulo operation: *A|B* is the remainder left when A is divided an integral number of times into B (note the order of the operands: the left operand is the divisor and the right operand is the dividend). More precisely, if A is not 0, then an integer quotient Q is chosen so that the remainder *B-(Q×A)* is the smallest possible non-negative remainder (i.e., greater than or equal to 0 but strictly less than the absolute value of A), and this remainder is the value of *A|B*. If A is 0 and B is not negative, then B itself is the value of *A|B*. If A is 0 and B is negative, then *A|B* causes a *DOMAIN ERROR*.

```
      10|27
7
      3.5|¯8.3                       Because
2.2                                  0 ≤ 2.2 = ¯8.3-¯3×3.5 < 3.5.
      1|2.718281828 3.141592654
0.718281828  0.141592654
```

ABSOLUTE VALUE |

The vertical bar | used monadically represents the absolute value of its right operand; that is, the algebraic sign of the operand is changed to positive if it was negative.

```
      |¯2 ¯1 0 1 2
2  1  0  1  2
```

FACTORIAL AND BINOMIAL COEFFICIENT !

The exclamation point ! , when used monadically, represents the factorial of its right operand (note that it is written before its operand, as are all monadic operators in APL, as opposed to following it as in conventional mathematical notation). For non-integer operands, *!A* represents the gamma function of A plus 1. A *DOMAIN ERROR* results if A is a negative integer (the gamma function is singular for all negative integers).

The dyadic exclamation point *A!B* represents the A-th binomial coefficient of degree B, or the number of combinations of B things taken A at a time. More precisely, *A!B* is B factorial divided by the product of the factorials of A and (B minus A). If A or B is not integral, then the gamma function is used to interpolate, as in monadic factorial.

```
      !1 2 3 4 5
1  2  6  24  120
      !1.5
1.329340388
      0 1 2 3 4 5 6!6
1  6  15  20  15  6  1
      2!3.3
3.795
```

MAXIMUM AND MINIMUM ⌈ ⌊

The dyadic operators ⌈ and ⌊ represent the maximum and minimum operations respectively. They are defined only for numeric operands; characters have no collating sequence in APL.

```
      3⌈1 2 3 4 5
3  3  3  4  5
      3⌊1 2 3 4 5
1  2  3  3  3
      ¯73.1⌈¯29.88
¯29.88
```

CEILING AND FLOOR ⌈ ⌊

The monadic forms of ⌈ and ⌊ represent the ceiling and floor operations respectively. Ceiling is defined as the algebraically smallest integer greater than or equal to its operand; floor is defined as the algebraically largest integer less than or equal to its operand.

A number is considered equal to an integer if it is within a certain tolerance of that integer. This tolerance is called fuzz. Fuzz is discussed in Section IV.

```
      ⌊0.6 1 1.4 1.8 2.2
0  1  1  1  2
```

```
      ⌈0.6 1 1.4 1.8 2.2
1  1  2  2  3
      ⌈¯0.6 ¯1 ¯1.4 ¯1.8 ¯2.2
0  ¯1  ¯1  ¯1  ¯2
      V←¯2.1 ¯3 ¯3.9 ¯4.8 16.99
      ⌊V
¯3  ¯3  ¯4  ¯5  16
      1|V
0.9  0  0.1  0.2  0.99
      (⌊V)+(1|V)
¯2.1  ¯3  ¯3.9  ¯4.8  16.99
```

RANDOM NUMBER ?

While the dyadic form of the ? operator is a mixed operator (the deal operator), monadic ? is a scalar operator, the random number generator. The operand of the random number operator must be a positive integer, say A, and the result of the operator is an integer chosen randomly and uniformly from the set of integers ⍳*A*. As explained under the ⍳ (index generator) operator, the set ⍳*A* is a vector of A integers, either 1,2,3,...,A or else 0,1,2,...,A-1, depending upon whether the workspace index origin is set to 1 or to 0 respectively. The index origin can be changed with the)*ORIGIN* system command or the *ORIGIN* library function.

The random number algorithm used by Multics APL is a multiplicative congruential generator with period 34359738368. In this algorithm, the seed used to produce each random number is a function of the seed used to produce the previous one. In a clear workspace, the starting seed is derived from the calendar clock, so that the sequences of random numbers generated are ordinarily unpredictable from session to session. If it is desirable to work with a reproducible sequence of random numbers, the user should specifically initialize the seed with the)*SETLINK* system command or the *SETLINK* library function. The seed can be set to any integral value from 1 to 34359738367. The seed is properly remembered and restored by the)*SAVE* and)*LOAD* system commands.

```
      ?10
9
      ?10
5
      ?10
5
      ?10 10 10 10 10 10 10 10 10 10 10
8  5  4  9  2  9  3  9  3  8  6
      )SETLINK 666
```

```
WAS 17970104840
      ?10 10 10 10 10 10
9  3  6  7  2  4
      )SETLINK 666
WAS 10635721024
      ?10 10 10 10 10 10
9  3  6  7  2  4
      )ORIGIN 0
WAS 1
      )SETLINK 666
WAS 10635721024
      ?10 10 10 10 10 10
8  2  5  6  1  3
```

COMPARISON OPERATORS   < ≤ = ≠ ≥ >

The APL comparison operators are < ≤ = ≠ ≥ and >. They represent the mathematical relations of less than, less than or equal to, equal-to, not-equal-to, greater-than-or-equal-to, and greater than respectively. The comparison operators are all dyadic operators, and they all return the numerical value 1 to signify "true" and the numerical value 0 to signify "false".

Operands of < ≤ ≥ and > must be numeric, otherwise a *DOMAIN ERROR* occurs. Operands of = and ≠ can be numeric or character or both. A number is considered not equal to a character; hence, in a mixed-type comparison, = always returns 0 and ≠ always returns 1.

Two numbers are considered equal if they are within a certain tolerance of each other. This tolerance is called fuzz. Fuzz is discussed in Section IV.

```
      5.3>4 5 6 7
 1  1  0  0
      'NO QUOTES'≠'NOPQRSTUV'
 0  0  1  0  1  1  0  1  1
      5='5'
 0
      ¯4.3 ¯3.9 ¯3.5≤¯4
 1  0  0
      'IS'='ARE'
 LENGTH ERROR.
      'IS'='ARE'
          |
```

LOGICAL OPERATORS ~ ∧ ∨ ⍲ ⍱

The symbols ~ ∧ ∨ ⍲ and ⍱ represent the logical operations NOT, AND, OR, NAND, and NOR respectively. The NOT operator ~ is monadic; the other four are dyadic. Both the operands and the results of the logical operators are restricted to the two numeric values 1 and 0, which signify "true" and "false" respectively. *~A* is 1 if and only if A is 0. *A∧B* is 1 if and only if both A and B are 1. *A∨B* is 0 if and only if both A and B are 0. *A⍲B* is 0 if and only if both A and B are 1. *A⍱B* is 1 if and only if both A and B are 0.

By virtue of their actions on operands of 0 and 1, the six comparison operators introduced in the preceding section can also be used as dyadic logical operators, with = representing EQUIVALENCE, ≠ EXCLUSIVE OR, ≤ IMPLICATION, and ≥ is IMPLIED BY. This gives APL the complete set of all ten nontrivial dyadic logical operations. *A=B* is 1 if and only if A and B are both 0 or both 1. *A≤B* is 1 unless A is 1 and B is 0. *A>B* is 0 unless A is 1 and B is 0. *A≥B* is 1 unless A is 0 and B is 1. And finally, *A<B* is 0 unless A is 0 and B is 1.

```
      ~0 1
1 0
      0 0 1 1∧0 1 0 1
0 0 0 1
      0 0 1 1>0 1 0 1
0 0 1 0
      0 0 1 1<0 1 0 1
0 1 0 0
      0 0 1 1≠0 1 0 1
0 1 1 0
      0 0 1 1∨0 1 0 1
0 1 1 1
      0 0 1 1⍱0 1 0 1
1 0 0 0
      0 0 1 1=0 1 0 1
1 0 0 1
      0 0 1 1≥0 1 0 1
1 0 1 1
      0 0 1 1≤0 1 0 1
1 1 0 1
      0 0 1 1⍲0 1 0 1
1 1 1 0
      001.0∧0.001E3
1
```

CIRCULAR AND HYPERBOLIC OPERATOR ○

The circle operator ○ is used to generate the common circular and hyperbolic functions of its right operand. The left operand determines which function is generated. Angular values are expressed in radians.

¯7○*A*	is defined as	argtanh *A*;
¯6○*A*	is defined as	argcosh *A*;
¯5○*A*	is defined as	argsinh *A*;
¯4○*A*	is defined as	(¯1+*A*×*A*)⋆0.5;
¯3○*A*	is defined as	arctan *A*;
¯2○*A*	is defined as	arccos *A*;
¯1○*A*	is defined as	arcsin *A*;
0○*A*	is defined as	(1-*A*×*A*)⋆0.5;
1○*A*	is defined as	sin *A*;
2○*A*	is defined as	cos *A*;
3○*A*	is defined as	tan *A*;
4○*A*	is defined as	(1+*A*×*A*)⋆0.5;
5○*A*	is defined as	sinh *A*;
6○*A*	is defined as	cosh *A*;
7○*A*	is defined as	tanh *A*;

○*A* (monadic) is defined as *A*×3.14159265358979...

Any other left operand of ○ is a *DOMAIN ERROR*.

```
      ¯1○0 0.5 1
0  0.5235987756  1.570796327
      ○0.5 1 10
1.570796327  3.141592654  31.41592654
      5○0.5 1 10
0.5210953055  1.175201194  11013.23287
      ⋆0.5 1 10
1.648721271  2.718281828  22026.46579
      ○÷180                     Radians per degree.
0.01745329252
      ÷○÷180                    Degrees per radian.
57.29577951
      1○(0 10 20 30 40 50 60 70 80 90×○÷180)
0  0.1736481777  0.3420201433  0.5  0.6427876097
      0.7660444431  0.8660254038  0.9396926208
      0.984807753  1            Table of sines.
      (¯1○0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1)÷○÷180
0  5.739170477  11.53695903  17.45760312  23.57817848
      30  36.86989765  44.427004  53.13010235  64.15806724
      90                        Table of arc sines.
      3○(¯3○1 2 3 4 5)
1  2  3  4  5
```

Mixed Operators

A mixed operator is one that must consider an array operand as a whole, rather than acting independently on its constituent elements. Each individual mixed operator has its own rules about the rank and dimensions of operands it will accept. Like the scalar operators, some mixed operators can be used either monadically or dyadically, with some change in meaning of the operation performed.

A few of the operator descriptions in this section make some use of subscript notation before it is formally introduced. That is, *V[I]* is used to refer to the I-th element of the vector V, and *M[I;J]* is used to refer to the I,J-th element of the matrix M. The subscripting capability of APL is actually far more powerful than these simple uses suggest, and is discussed at length in "Other Operators and Symbols" in this section.

SHAPE ρ

Monadic ρ is an operator whose value is the dimension vector, or shape as it is frequently called in APL, of its right operand. The type and actual element values of the operand are ignored. The result of the shape operator is always a numeric vector.

The shape of a scalar (which has no dimensions) is a null vector (a vector with a length of 0; i.e., containing no elements). The shape of a vector is a vector of length 1 (because the operand has one dimension) whose single element value is the length of the operand. The shape of a matrix (2 dimensions) is a vector of length 2, whose element values are the extents of the two dimensions of the operand matrix. Similarly, the shape of any array is a vector of length equal to the rank, or number of dimensions, of the array.

Since the result of the shape operator is a vector, the shape operator can be applied to that result to produce the shape of the shape. As the length of the shape is the rank of the original operand, this is a way of obtaining the rank of any array.

```
      ρ1 2 3 4               The shape of a vector is its
4                            length (number of elements).
      ρ'2+A⍉B'               The shape of a character vec-
5                            tor.  Five characters.
      ρ''                    The shape of a null vector.
0                            It has no elements.
      ρ93                    The shape of a scalar is a
                             null vector.
      ρρ93
0
      ρρ2 3 4 5 6
1
```

The rank of a scalar is the shape of a null vector: zero. The rank of a vector is one.

RESHAPE ρ

The dyadic ρ is the reshape operator. It is used to form a sequence of element values into a specified shape. The left operand of the reshape operator must be a shape (a vector of non-negative integers). The element values of the right operand are used to fill up an array of the shape specified as the left operand. The shape of the right operand is ignored.

If the new array requires less elements than the old array provides, the excess elements are simply not used. If the new array requires more elements than the old array provides, then the elements of the old array are repeated over and over, as many times as are necessary to fill up the new shape.

The element values are extracted and packed in row-major order. That is, the first elements treated are those of the first row of the first plane, followed by the second row of the first and so on, through the last row of the last plane.

As a convenience, a single element specified as the left operand of a reshape is accepted as if it were a one-element vector, regardless of its true shape. If the new array is to have any elements at all, then the right operand of the reshape operator must have at least one element.

```
      5ρ1 2 3
 1  2  3  1  2
      M←2 3ρ1.1 2.2 3.3 ¯4.4
      M

  1.1  2.2  3.3
 ¯4.4  1.1  2.2
      3 2ρM

  1.1  2.2
  3.3 ¯4.4
  1.1  2.2
      6ρ10
 10  10  10  10  10  10
      0ρ3.14
```

Three element vector reshaped to a five-element vector.

A matrix formed by reshaping a vector.

The same elements values used to make a different shape.

A scalar reshaped to a vector.

A scalar reshaped to a null vector.

```
      ''ρM                          A matrix reshaped to a scalar.
1.1
      3 5ρ'ONE  TWO  THREE'

ONE
TWO
THREE
      3 2 4ρ'+-×÷+⋆=○'

+-×÷
⋆=○+

-×÷⋆
=○+-

×÷⋆=
○+-×
```

RAVEL   ,

The monadic comma , is the ravel operator. The ravel operator makes its operand into a vector, by retaining all of its elements but ignoring its shape.

```
      5
5
      ρ5

      ,5
5
      ρ,5
1
      M←2 3ρ'ABCD'
      M

ABC
DAB
      ,M
ABCDAB
```

CONCATENATE   ,

The dyadic comma , is the concatenate operator. Both its operands must be scalars or vectors, and the result is a vector in which the elements of the left operand are followed by the elements of the right operand. Since a single APL value cannot contain both character and numeric elements, the operands of the concatenate operator must be both character or both numeric.

```
      1 2 3,4
1  2  3  4
      'WHY','NOT'
WHYNOT
      (8+7),(21÷3)
15  7
      1.1 2.2 3.3,''
1.1  2.2  3.3
      1.1 2.2 3.3,'ABC'
DOMAIN ERROR.
      1.1 2.2 3.3,'ABC'
                 |
```

INDEX GENERATOR   ⍳

The monadic iota ⍳ is the index generator operator. Its operand must be a single non-negative integer value. The result of ⍳A is a vector of integers of length A, the first element of which is the workspace index origin (either 0 or 1), and succeeding elements of which are each one greater than the preceding element.

The workspace index origin can be changed with the)ORIGIN system command or the ORIGIN library function.

```
      )ORIGIN 1
WAS 0
      ⍳4
1  2  3  4
      ⍳10
1  2  3  4  5  6  7  8  9  10
      ⍳0

      ⍳1
1
      )ORIGIN 0
WAS 1
      ⍳4
0  1  2  3
      ⍳10
0  1  2  3  4  5  6  7  8  9
      ⍳0

      ⍳1
0
```

INDEX ⍳

The dyadic usage of iota as *A*⍳*B* represents the index of the first occurrence of B in the vector A. The left operand of index must be a vector. The right operand can be an array of any shape; its elements are considered independently of one another, producing an answer of shape identical to the right operand. As each element is selected from B, it is compared to the successive elements of A, starting with the first and proceeding until a match is found. If a match is found, then the answer is the number of the element that matched.

The numbers returned by index follow the workspace index origin. If the index origin is 0, then the first element of A is numbered 0, the next is numbered 1, and so on. If the index origin is 1, then the first element of A is numbered 1, the next 2, and so on.

Two numeric elements are considered equal if they are within a certain tolerance of each other. This tolerance is called fuzz, and is discussed in Section IV. If more than one element of A matches the element of B being considered, the index of the earliest is returned. If no element of A matches, then an index one greater than the last index of A is returned; e.g., if A has seven elements, then 7 will be returned in origin 0 (because the elements of A are numbered from 0 to 6), but 8 will be returned in origin 1 (because the elements are then numbered from 1 to 7).

```
      )ORIGIN 1
WAS 0
      6 7 9 3 4⍳9
3
      6 7 9 3 4⍳1 2 3 4
6  6  4  5
      'ABCDEF'⍳'FAT CAT'
6  1  7  7  3  1  7
      )ORIGIN 0
WAS 1
      '+-×÷XYZ'⍳'X+2×Z'
4  0  7  2  6
      'FEEBLE'⍳'BED'
3  1  6
      M←2 3⍴'FATCAT'
      M

FAT
CAT
      'ABCDEF'⍳M

 5  0  6
 2  0  6
```

TAKE AND DROP ↑ ↓

Take ↑ and drop ↓ are both dyadic operators that accept a vector of integers V as left operand and any array A as right operand. The length of the vector V must be equal to the rank of the array A (except that a scalar V is automatically replicated to the rank of A). The function of take *V↑A* is to take or retain, for each dimension I in the rank of A, the first (if *V*[*I*]>0) *V*[*I*] elements of that dimension, or the last (if *V*[*I*]<0) |*V*[*I*] elements of that dimension, discarding the other elements. The function of drop *V↓A* is to drop or discard the first (if *V*[*I*]>0) or last (if *V*[*I*]<0) |*V*[*I*] elements of each coordinate I, retaining the others. For both take and drop, |*V*[*I*] must not be greater than (⍴*A*)[*I*].

```
      A←?3 5⍴100                    A random 3-by-5 array.
      A

 98  52   9  26  58
 34  14  91  47  15
 97  33  66  85  26
      2 ¯3↑A                        Take the first 2 rows and
                                    the last 3 columns.
  9  26  58
 91  47  15
      ¯1↓A                          Same as ¯1 ¯1↓A, drop the last
                                    row and the last column.
 98  52   9  26
 34  14  91  47
      7↑'ABRAHAM LINCOLN'
 ABRAHAM
      7↓'ABRAHAM LINCOLN'
  LINCOLN
```

GRADE UP AND GRADE DOWN ⍋ ⍒

The grade up ⍋ and the grade down ⍒ operators are the APL sorting operators. They are both monadic, and accept any numeric array as operand (characters have no collating sequence in APL--hence they cannot be sorted). The result of ⍋ or ⍒ is a permutation array (or array of subscripts) identical in shape to *A* that orders the elements of *A* to be monotonically nondecreasing or nonincreasing along the last dimension of *A*. That is, when the result of the grade operator is used to subscript its operand, the resultant value is found to be sorted along the last dimension. The sort preserves the original order of equal elements. Elements are considered equal if they are within a certain numeric tolerance of each other, known as the <u>fuzz</u>; see Section IV.

If *A* is a vector, then *A*[⍋*A*] is the elements of *A* sorted into increasing order. If *A* is a matrix, then ⍋*A* is a permutation matrix each row of which orders each row of *A* into ascending order, so *A*[*I*;(⍋*A*)[*I*;]] is the I-th row of *A* sorted.

If an integer-valued expression in brackets follows the grade operator, as ⍋[*I*]*A*, then the value of that expression *I* is taken as the coordinate number upon which to sort instead of the last coordinate. The coordinate numbers as well as the subscripts returned by the grade operators follow the numbering of the index origin. Thus, in 1-origin indexing, if *A* is a matrix, then ⍋[2]*A* is the same as ⍋[*A*], while ⍋[1]*A* is the permutation matrix which orders the columns of *A* into increasing order.

```
        V←3 1 4 1 5 9                    A vector to sort.
        ⍋V                               Permutation vector, says big-
  2  4  1  3  5  6                       gest item is second, next is
        V[⍋V]                            fourth, etc.
  1  1  3  4  5  9                       V, sorted.
        ⍒V                               Permutation vector for descen-
  6  5  3  1  2  4                       ding sort, note equal items.
        A                                An array to sort.

   98  52   9  26  58
   34  14  91  47  15
   97  33  66  85  26
        ⍋A                               Sorting along the rows.

   3  4  2  5  1
   2  5  1  4  3
   5  2  3  4  1
        ⍒[1]A                            Sorting along the columns.

   1  1  2  3  1
   3  3  3  2  3
   2  2  1  1  2
        ALF←'ABCDEFGHIJKLMNOPQRSTUVWXYZ,. '
        M←ALF⍳'LETTERS TO BE SORTED.'
        ALF[M[⍋M]]                       Letters cannot be sorted, but
  BDEEEELOORRSSTTTT.                     they can be made into numbers.
```

REVERSE ⌽ ⊖

The reverse operator ⌽ is a monadic operator that reverses the elements of the last dimension of its operand array. Like the grade operators, the reverse operator also accepts the number of a dimension upon which to act in brackets; ⌽[*I*]*A* reverses the elements of A along the I-th dimension; where I can be specified by an integer valued expression. The dimension numbers follow the setting of the workspace index origin. The operator ⊖ is a shorthand for ⌽[1] in 1-origin or ⌽[0] in 0-origin; that is, ⊖ reverses along the first coordinate instead of the last.

```
       )ORIGIN 1
WAS 0
       V
3  1   4   1   5   9
       ⌽V
9  5   1   4   1   3                 V, reversed.
       ⍋V
2  4   1   3   5   6
       ⌽⍒V                           Not the same as ⍋V because of
4  2   1   3   5   6                 the two equal elements.
       ⌽[3](2 3 4⍴⍳24)               Same as ⌽(2 3 4⍴⍳24).  Rever-
                                     sal along the last coordinate
   4    3    2    1                  of a three-dimensional array.
   8    7    6    5
  12   11   10    9

  16   15   14   13
  20   19   18   17
  24   23   22   21
       ⌽[2](2 3 4⍴⍳24)               Reversal along the second di-
                                     mension.
   9   10   11   12
   5    6    7    8
   1    2    3    4

  21   22   23   24
  17   18   19   20
  13   14   15   16
       ⌽[1](2 3 4⍴⍳24)               And along the first.  Same as
                                     ⊖(2 3 4⍴⍳24).
  13   14   15   16
  17   18   19   20
  21   22   23   24

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

ROTATE ⌽ ⊖

The dyadic forms of ⌽ and ⊖ represent the rotate operators: ⌽ rotates the elements of its right operand along the last dimension, ⌽[*I*] rotates along the I-th dimension (the coordinate numbering following the index origin), and ⊖ rotates along the first dimension.

The left operand of ⌽ and ⊖ specifies the amount of rotation as follows: in the expression *A*⌽[*I*]*B*, A must be an array of integers one less in rank than the array B, each integer specifying the number of positions to the left that each corresponding vector of B along the I-th dimension is to be rotated. Elements rotated off the left end of a vector re-enter it on the right end. Zero is a valid rotation (which results in no change), as are negative numbers (which result in rotation to the right), as well as very large numbers (which may have the effect of rotating the vector through its starting position several times--the interpreter is clever enough to avoid performing the superfluous complete cycles). If a scalar is given for A, then it is replicated to the required shape; i.e., all vectors of B along the I-th coordinate are rotated by the same scalar amount.

```
      2⌽3 1 4 1 5 9
 4  1  5  9  3  1
      A

  98  52   9  26  58
  34  14  91  47  15
  97  33  66  85  26
      ¯1 0 523⌽A

  58  98  52   9  26
  34  14  91  47  15
  85  26  97  33  66
      0 1 2 3 4⊖A

  98  14  66  26  15
  34  33   9  47  26
  97  52  91  85  58
```

TRANSPOSE ⍉

The monadic ⍉ is the ordinary transpose operator. It interchanges the coordinate numbering of the last two coordinates of its array operand: the last coordinate becomes the next-to-last, and the next-to-last becomes the last. Obviously, monadic transpose requires an operand of rank at least 2.

```
      A

  98  52   9  26  58
  34  14  91  47  15
  97  33  66  85  26
      ⍉A

  98  34  97
  52  14  33
  9   91   66
 26   47   85
 58   15   26
       ⍉(2 3 4⍴⍳24)

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

 13   17   21
 14   18   22
 15   19   23
 16   20   24
```

In the dyadic transpose operation *A*⍉*B*, A must be a vector of integers of length equal to the rank of B, so that *A*[*I*] corresponds to the I-th dimension of B. Then, dimension I of B becomes dimension *A*[*I*] of the result. The numbering of dimensions in A follows the workspace index origin.

It is not necessary that all the integers in A be different. If two or more integers of A are equal, then that dimension of the result is composed of elements taken from the diagonal crossing the dimensions of B that map into it (if the several dimensions of B are not identical in extent, then the resultant dimension stops as soon as the shortest is exhausted). For example, 1 1⍉*B* is the ordinary major diagonal of the matrix B. It is required, however, that all dimensions that will finally appear in the result value be specified somewhere in the vector A. That is, the vector A must consist of the numbers from the index origin (which is the number of the first dimension) through the highest element of A, with some possibly repeated but none missing. Or, stated in APL, the elements of the vector A must be chosen from the set ⍳⌈/*A* and every member of ⍳⌈/*A* must be present at least once in A.

```
       3 1 2⍉(2 3 4⍴⍳24)

  1   13
  2   14
  3   15
  4   16

  5   17
  6   18
  7   19
  8   20

  9   21
 10   22
 11   23
 12   24
        3  2  1⍉(2  3  4⍴⍳24)

  1   13
  5   17
  9   21

  2   14
  6   18
 10   22

  3   15
  7   19
 11   23

  4   16
  8   20
 12   24
        1  1  2⍉(2  3  4⍴⍳24)

  1    2    3    4
 17   18   19   20
        1  2  1⍉(2  3  4⍴⍳24)

  1    5    9
 14   18   22
        1  2  2⍉(2  3  4⍴⍳24)

  1    6   11
 13   18   23
        1  1  1⍉(2  3  4⍴⍳24)
1   18
        2  1  2⍉(2  3  4⍴⍳24)

  1   14
  5   18
  9   22
```

COMPRESSION   /   ⌿

Compression is a dyadic operator. In the expression *A/B*, A must be a vector of ones and zeros, of length equal to the last dimension of the array B. The resultant value is obtained by selecting or retaining those elements along the last dimension of B that correspond to a 1 in the vector A, and omitting or dropping those elements that correspond to a 0. Thus, the result is of the same rank as B, and of the same dimensions except for the last, along which it has been compressed.

In the expression *A*/[*I*]*B*, where I is an integer-valued expression, the dimension along which the compression is to act is I (the numbering follows the index origin). The expression *A*⌿*B* is a shorthand for compression along the first dimension.

```
      1 1 0 1 0 0/3 1 4 1 5 9
3  1  1
      0 0 1 0 0 1 0 0 0 1 1 0 0/'JKLMNOPQRSTUV'
LOST
      A
 98  52   9  26  58
 34  14  91  47  15
 97  33  66  85  26
      1 0 1 1 0/A
 98   9  26
 34  91  47
 97  66  85
      0 1 0⌿A
 34  14  91  47  15
```

EXPANSION \ ⍀

Like compression, expansion is a dyadic operator requiring a vector of ones and zeros as left operand. However, in the case of expansion it is the number of ones in the left operand vector that must equal the last dimension of the right operand array. The resultant value of *A**B* is obtained by including each of the elements along the last dimension of B in the answer in positions corresponding to a 1 in the vector A, and filling in the answer positions corresponding to a 0 in A with either 0 (if B is numeric) or else blank (if B is character). Thus, the result is of the same rank as B, and also of the same dimensions, except for the last, along which it has been expanded.

In the expression *A*\[*I*]*B* the integer-valued expression I gives the number of the dimension to be expanded instead of the last (the numbering of the dimensions follows the workspace index origin). The expression *A*⍀*B* indicates expansion along the first dimension.

```
      1 1 0 1 0 1 1 0\3 1 4 1 5
3  1  0  4  0  1  5  0
      1 1 0 1 1 0 1 0 1 1 1 0 1 1 1 1\'ITISABIGONE.'
IT IS A BIG ONE.
```

```
      A
98  52   9  26  58
34  14  91  47  15
97  33  66  85  26
      1 0 0 1 1 0\A
98  52   9  26  58
 0   0   0   0   0
 0   0   0   0   0
34  14  91  47  15
97  33  66  85  26
 0   0   0   0   0
```

MEMBERSHIP   ϵ

In the expression *AϵB* the result is an array of ones and zeros identical in shape to the array A, with ones corresponding to those elements of A that are found to occur somewhere (anywhere) within array B, and zeros for those elements of A that are not found in B. The shape of B is irrelevant; the array B merely represents a collection of objects, and the ϵ operator determines which elements of a given array A are members of the collection and which are not.

```
      D←0 1 2 3 4 5 6 7 8 9  The digits from 0 to 9.
      Dϵ3 1 4 1 5 9          What digits are used in this
0 1 0 1 1 1 0 0 0 1          number? 1, 3, 4, 5, and 9.
      (Dϵ3 1 4 1 5 9)/D      Compression operator can list
1 3 4 5 9                    them by name.
     Dϵ2 7 1 8 2 8           What digits are used in this
0 1 1 0 0 0 0 1 1 0          number?
      (Dϵ3 1 4 1 5 9)∧(Dϵ2 7 1 8 2 8)
0 1 0 0 0 0 0 0 0 0          Digits in common (intersec-
      (Dϵ3 1 4 1 5 9)∨(Dϵ2 7 1 8 2 8)    tion of sets).
0 1 1 1 1 1 0 1 1 1          Digits in either (union).
      'ABCDEF'ϵ'THE FAT CAT'
1 0 1 0 1 1
      'THE FAT CAT'ϵ'ABCDEF'
0 0 1 0 1 1 0 0 1 1 0
```

ENCODE   ⊤

The encode operator *A⊤B* encodes a numeric scalar B into its positional representation in any number system, the radix being specified by the numeric vector A, each element of the vector A representing the radix applicable to the corresponding position. The result is a vector equal in length to the vector A, each element of which is the digit occupying the corresponding position of the representation of the value of B when expressed in the desired number system.

```
      10⊤1983                      Decimal representation of
3                                  1983, to one digit.
      10 10 10 10 10 10⊤1983       To enough digits.
0  0  1  9  8  3
      8 8 8⊤41                     Octal representation of 41.
0  5  1
      24 60 60⊤11642               A mixed radix representation:
3  14  2                           hours, minutes, and seconds.
```

EVALUATE ⊥

The evaluate operator *A⊥B* is the inverse of the encode operator. It accepts a numeric vector A defining the radices of the positions in a number system and a vector B of positions representing a number in that system. The result is a scalar, the value of the number. The vectors A and B must be the same length (except that a scalar on the left is replicated to match the length of the right operand).

```
      10 10 10 10⊥1 9 8 3          The number whose decimal
1983                               representation is 1983.
      10⊥1 9 8 3
1983
      8⊥0 5 1                      The number whose octal repre-
41                                 sentation is 51.
      24 60 60⊥3 14 2              Hours, minutes, and seconds
11642                              converted to seconds.
      2⊥1 0 1 1 0                  Binary 10110.
22
```

DEAL *?*

Though the monadic form of *?* is a scalar operator, the dyadic form *A?B* is a mixed operator, the deal operator. A and B must be numeric integer scalars with $0 \le A \le B$, and the result of *A?B* is a vector of A elements selected randomly and without replacement from the set *⍳B*. Thus, the effect is that of shuffling a deck of B cards, and then dealing A of them. The set *⍳B* consists of either the integers from 0 to B-1, or from 1 to B, depending on whether the workspace index origin is set to 0 or to 1 respectively. Details of the random number generation algorithm used by the deal operator can be found under

"Random Number" earlier in this section which uses the same generator.

```
      3?12                          Three things dealt out of 12.
 4  7  1
      6?8                           Six things dealt from eight.
 8  4  7  3  1  2
      10?10                         A random permutation of the
 9  8  2  3  6  1  5  7  10  4          numbers from 1 to 10.
      10?10                         The selection is without re-
 3  9  6  4  5  7  2  8  1  10          placement.
      ?10⍴10                        Here, in contrast, it is with
 8  6  7  1  3  2  9  1  10  4          replacement.
```

MATRIX INVERSE ⌹

Monadic ⌹ is the matrix inverse operator. In the expression ⌹*A* the right operand A must be a numeric array whose shape is such that the product of all dimensions of A except the last (i.e., ×/¯1↓⍴*A*) is greater than or equal to the last dimension (¯1↑⍴*A*). The result is a matrix of ¯1↑⍴*A* rows and ×/¯1↓⍴*A* columns whose elements are chosen to least-squares best-fit the matrix product of *A* and ⌹*A* to the identity matrix of order ×/¯1↓⍴*A*. That is, if *X* is the result of ⌹*A*, then the elements of *X* are are chosen to minimize +/,((*A*+.×*X*)-*I*)*2 where I is the identity matrix.

If A is a square matrix, then ⌹*A* is the ordinary matrix inverse of A. If A is over-square (more rows than columns), then A is not exactly invertible, and ⌹*A* is the least-squares best inverse. If A is under-square, then ⌹*A* results in a *DOMAIN ERROR*.

```
      A←?3 3⍴100                    Get a matrix to invert.
      A

 76  86  11
 46   2  80
 18  94  73
      X←⌹A                          Invert it.
      X

 0.01087283472    0.007732186776  ¯0.0101120017
 0.002828057635  ¯0.007888481932   0.008218766035
¯0.006322581406   0.008251204652   0.005608931826
      A+.×X                         Test the result, which should
                                    be the identity matrix.
 1               ¯8.673617380E¯19 ¯8.673617380E¯19
¯8.673617380E¯19  1               ¯2.168404345E¯19
¯6.234162492E¯19 ¯5.149960319E¯19  1
```

MATRIX DIVIDE ⌹

Dyadic ⌹ is the matrix divide operator. The result *X* of *A*⌹*B* is chosen to least-squares best-fit the matrix product of *B* and *X* to *A*. More precisely, the elements of *X* are chosen to minimize +/,((*B*+.×*X*)-*A*)⋆2.

The shape requirements of the matrix divide operator are as follows. The shape of B must be of the form *H*,*U* where U is a single dimension and H is a (possibly null) vector of preceding dimensions, and the product of the elements of H, or ×/*H*, must be at least U. The shape of A must be of the form *H*,*T* where H is identical to the H in the shape of B, and T is a (possibly null) vector of succeeding dimensions. The result X has the shape *U*,*T* and it is ×/*T* sets of least-squares best solutions in U unknowns to the ×/*T* sets of ×/*H* linear equations in U unknowns.

```
        A                A matrix to use as a divisor.


 76  86  11
 46   2  80
 18  94  73
      6 3 9⌹A                   Divide a vector by it.
¯0.002574446627  0.06727179433  0.03729851195
      A+.×(6 3 9⌹A)             Test the answer, which should
6  3  9                         produce 6 3 9 in this matrix
                                multiplication.
```

I-BEAM ⌶

The I-beam operator ⌶ is a monadic operator that accepts as operand a numeric integer scalar chosen from a small set of values. The result is the value of some system parameter, which one being chosen by the operand. The results versus the value of the operand are as follows:

⌶19	The real time in sixtieths of a second since this instance of APL was invoked (uses "clock_").
⌶20	The time of day, in sixtieths of a second since midnight (uses "clock_" and "sys_info$time_delta").
⌶21	The CPU time used so far since this instance of APL was invoked, in sixtieths of a second (uses "hcs_$get_usage_values").
⌶22	The size of workspace remaining available to be used,

in units of 9-bit characters (i.e., four times the number of words). This number will reflect the fact that a Multics APL workspace can be many segments in size. However, since any single APL value must fit wholly within one segment, it is possible for some APL expressions to cause *WORKSPACE FULL* errors even when I-beam 22 is returning large values. For example, it is impossible to create a 300,000 character item, even in a clear workspace with millions of characters of space available.

⌶23	Multics "nusers".
⌶24	The time of day, in sixtieths of a second since midnight, that this instance of APL was invoked.
⌶25	The date, as a 6-digit integer, MMDDYY.
⌶26	The first element of I-beam 27 (or 0 if I-beam 27 is null).
⌶27	The vector of statement numbers in the state indicator, most recent first. An element of 0 is returned corresponding to evaluated input (⎕) entries in the state.

All results returned by I-beam are scalar integers except for I-beam 27, which is a vector of integers.

```
      24 60 60 60⊤⌶20          The current time of day, in
9  14  27  53                  hours, minutes, seconds, and
      ⌶25                      sixtieths.
102772                         The date, 27 October 1972.
      ⌶23
40                             Forty Multics users.
```

Other Operators and Symbols

In addition to scalar operators and mixed operators, which fall neatly into their respective classes, there are a number of other miscellaneous operators that defy classification. Also, there are a number of symbols that can appear in APL statements that are not operators but that still require explanation.

ASSIGNMENT ←

The left arrow ← is the assignment operator in APL. A variable name must appear immediately to the left of the assignment operator; any expression can be the right operand. The purpose of the assignment operator is to set the value of the variable to that of the expression. If a variable of the given name does not already exist in the workspace, then one is created (if the name is already in use for naming a function or a group, then it is an error to make an assignment to the name--a name can refer to only one object at a time). If the variable already exists, then its previous value is discarded.

Note that any variable can take on any APL value, regardless of its type or shape. There are no declarations of variables in APL.

In addition to being assigned to the variable name mentioned as left operand, the value of the expression on the right is also considered the result of the assignment operator in the APL statement in which it is found. This value may then be further operated upon. Thus, assignments, like other operators, can be used freely within statements.

In an APL statement consisting of a single expression, when the very last operation executed in that expression is the assignment operation, then the APL processor does not print the final value of the expression. On the other hand, if the last operation is not an assignment, then the value of the expression is printed. This automatic printing of values not assigned to any variable is the most common method of performing output in APL (the other two methods are to use the input/output symbol ⎕, which is discussed below, or to write a <u>list</u> for the APL statement, which is discussed later in this section.

```
      Q←⍳3                          Assign a value to Q.
      Q                             Now print it.
1  2  3
      Q←2 5⍴'NEW  VALUE'            Change its value.  Old value
      Q                             is discarded, only new value
                                    remains.
NEW
VALUE
      S←6×P←⍴Q←2↑Q                  Multiple assignments.
      Q                             Print the new values.

NE
VA
      P
2  2
      S
12  12
      )ERASE P Q S                  Variables can be erased.
      Q                             Now there is no Q.
VALUE ERROR.
      Q
      |
```

In addition to assigning a complete new value to a variable, it is also possible to assign values to specific elements of an array value, leaving the remaining elements undisturbed. This is done by writing a subscripted variable name to the left of the assignment operator, and is discussed below in "Subscripts".

OUTPUT ⎕

The quad symbol ⎕ is the output symbol in APL. It is not an operator; instead it behaves much like a variable. If a value is assigned to the output symbol (with the assignment operator ← discussed previously), then that value is printed. In addition, the value is also available for further operations within the statement, just as in any assignment operation.

```
       ⎕←Q←6 3 29                ⎕ can be used to force print-
 6  3  29                        ing of expressions ending in
       Q[⎕←⍋Q]                   assignments, or to print
 2  1  3                         intermediate results.
       1+⎕←1+⎕←1+⎕←1
 1                               These three lines printed by
 2                               the ⎕ symbol.
 3                               ..
 4                               This line is final value.
```

EVALUATED INPUT ⎕

The same symbol ⎕ is also the evaluated input symbol. We have already seen how it behaves as the output symbol when a value is assigned to it. In contrast, if a ⎕ is encountered that is not to the left of an assignment arrow, then that ⎕ is being used for input. The interpreter temporarily sets aside evaluation of the statement containing the ⎕, types the go-ahead string ⎕:, and then awaits the input of an APL expression from the user. Whatever expression is typed (it can be any APL expression) is evaluated and the result taken as the value of the ⎕symbol in the original statement. Interpretation of that statement then resumes.

Input typed in response to the ⎕symbol is called "evaluated input", because it can consist of a fully general APL expression to be evaluated.

If the expression typed in response to the ⎕ symbol has no value or results in an error report, then the interpreter again requests input by typing ⎕ as usual. Execution of the statement containing the ⎕ does not proceed until valid input is received.

If a system command or function editor invocation is typed when the interpreter is expecting evaluated input, the command or edit is performed; when the function is finished, the interpreter again requests the evaluated input.

If the user wishes to abandon execution of a statement or program requesting input instead of supplying it, a right arrow alone is typed. As discussed later, this has the effect of returning the interpreter back to its state at the last time it was accepting regular immediate input from the terminal.

```
        3×(⎕+2)                     A line requiring evaluated in-
⎕:                                  put.  Interpreter calls for
        ⍳3                          it.  Any expression is OK.
9   12   15                         Result of original line.
        'DID YOU SAY ',⎕,'?'
⎕:
        'HELLO'
DID YOU SAY HELLO?
        ,⌽⎕
⎕:
        2 4⍴⎕                       Any expression is legal, even
⎕:                                  ones with further ⎕ symbols.
        'WHY NOT?'
 YHW?TON
        ⍳⍴⎕
⎕:
        THIS IS JUNK.
VALUE ERROR.
        THIS IS JUNK.
        |
⎕:                                  Input requested again.
        6 3 29 8
1   2   3   4                       This time it is good.
```

CHARACTER INPUT ⍞

Another form of input, known as "character input", can be invoked by using ⍞ as the input symbol rather than ⎕. When the character input symbol is encountered, the interpreter reads one line from the user's terminal and takes the characters typed as the value of the ⍞ symbol. No go-ahead signal is typed for this form of input; the APL program employing the character input symbol has already notified the user that input is expected. Also, the line input is not evaluated in any way--its characters are simply made available verbatim as the character value of the ⍞ symbol.

The newline character that terminates the input line is not included in the value. Also, a line consisting of exactly one character is considered a scalar; any other line length results in a vector. A newline alone is legal and it results in a null vector.

No errors are possible in character input, since no evaluation is performed. Entry of system commands or function editor calls or the right arrow during character input is not possible; these characters, as any others, are simply made available as the value of the ⍞ symbol. To escape from character input and to restore the interpreter to the point of the last console suspension, issue a Multics QUIT followed by a "program_interrupt". This has the same effect on the interpreter state as typing a right arrow alone to evaluated input or following a function suspension. See "Clearing the State Indicator" in this section.

```
      'DID YOU SAY ',⍞,'?'
HELLO
DID YOU SAY HELLO?
      'ABCDEFGHIJKLMNOPQRSTUVWXYZ,. '⍳⍞
SECRET MESSAGE.
19  5  3  18  5  20  29  13  5  19  19  1  7  5  28
      ⍞                        Call for input.
)ORIGIN 0                      Type the input.
)ORIGIN 0                      Value of resultant expression.
      ⎕
⎕:                             In evaluated input, the system
      )ORIGIN 0                command is acted upon.
WAS 1
⎕:                             And then the input is again
      ')ORIGIN 0'              requested.
)ORIGIN 0
      ⍞
3+2
3+2                            No evaluation.
      ⎕
      3+2
5                              3+2 evaluated.
```

JUMP →

The jump operator → is normally valid only in APL statements stored in a function definition, not in lines typed from the terminal for immediate execution. It is a monadic operator expecting an integer operand, the number of the statement to which to jump (in the absence of a jump operator, statements in a function definition are executed sequentially).

A precise description of jump usage is found under "Branching" in this section.

The jump operator is validly typed from a terminal when restarting or abandoning a function execution that was suspended due to an error. (See "Restarting Suspended Functions" and "Clearing the State Indicator" in this section) or when abandoning evaluated input (see the discussion of "Evaluated Input" above).

`→3`	Jump to statement 3.
`→I`	Jump to statement I. The value of I must be an integer statement number.
`→(N>12)/9`	Jump to statement 9 if N is greater than 12.
`→`	Unwind the state indicator stack back to the previous console suspension.

PARENTHESES ( )

As in ordinary mathematics and conventional programming languages, parentheses are not operators but symbols of grouping. Within any expression, if a subexpression is placed within parentheses, then that subexpression is evaluated by itself and the result of the evaluation participates as a unit in the evaluation of the remainder of the expression. Parenthesized subexpressions can be nested to any level, and redundant parentheses are not harmful.

In the absence of any parentheses, there is a rule that establishes what grouping APL assumes about the constituents of an expression. The rule is carefully explained under "The Right-to-Left Rule", in this section.

```
      (3+7)×2                     Parentheses needed because
20                                3+(7×2) is not intended.
      (○1)*÷2                     Sqrt(pi).  Without parens,
1.772453851                       gives pi times sqrt(1), or pi.
      ÷(÷1)+(÷2)+(÷3)+(÷4)        Parallel resistance.  With no
0.48                              parens, gives incorrect 43÷30.
      (5+3)⍴2                     Gives 7 7 7 when the parens
2  2  2  2  2  2  2  2            are deleted.
      (((((4+8)))))               Extra parentheses are not
12                                harmful.
```

LISTS ;

A list is a series of expressions separated by semicolons. Thus, the semicolon is not an operator, but serves to separate expressions. There are two uses for lists in APL.

First, an APL statement can be a list of expressions instead of only one expression. The meaning of a list as a statement is fully discussed later, under "Statements", in this section, but briefly, the effect is simply to print the values of each of the expressions, one after the other, with no intervening spaces.

```
      )ORIGIN 1
WAS 0
      1;2;ι3;4                  A list of expressions.  Values
121  2  34                      are printed side-by-side.
      'LOG PI = ';⍟○1           Lists can be used to label or
LOG PI = 1.144729886            identify results.
      'THE MATRIX IS';2 3ρι6;', YOU SEE.'
THE MATRIX IS                   The newlines and spaces
 1  2  3                        inherent in the values
 4  5  6, YOU SEE.              themselves are unchanged.
```

The second use for lists in APL is as array subscripts. Subscripts are described next.

SUBSCRIPTS [;]

Any nonscalar APL value, a constant, a variable, or a parenthesized subexpression, can be followed by a subscript list, which is a list of expressions separated by semicolons and placed within brackets. The number of expressions in the list must be equal to the rank of the value being subscripted, and the result of the subscripting operation is a new array formed from the elements of the original value that are selected by the subscript list.

The shape of the result of a subscripting operation is determined by the shapes of the subscripts, not by the shape of the value being subscripted. However, the shape of the value being subscripted does determine how many expressions must be in the subscript list and the ranges of their allowed values.

If A is an array value of rank R, and it is subscripted by a list of expressions E1, E2, ..., ER, then the result of `A[E1;E2;...;ER]` is an array of shape `(ρE1),(ρE2),...,(ρER)` consisting of the elements chosen from A whose first subscripts are equal to elements of E1, whose second subscripts are equal to elements of E2, and so on.

Thus, if all subscripting expressions E1, ...,ER are scalars, then the result of the subscript operation is also a scalar, namely the single element of the array A having precisely those subscripts. If some subscript expression is a vector or higher-ranked array, then the result is a similarly shaped array, built up from the various elements selected out of A by the various elements of the subscripting array. Finally, if many or all of the subscript expressions are arrays, the result has rank equal to the sum of their ranks, each subscript expression contributing its dimensions to the rank of the result.

The elemental subscript values themselves are interpreted according to the current workspace index origin. For example, in 0-origin indexing, the subscripts for a 5 by 7 matrix are allowed to take on the values 0 to 4 for the first dimension and 0 to 6 for the second. In 1-origin indexing, the corresponding allowed ranges are 1 to 5 and 1 to 7, respectively. Any subscript value that is not numeric, not integral, or outside the dimensions of the subscripted array value causes an *INDEX ERROR* to occur, with the error marker placed under the left bracket opening the subscript list.

A vacuous expression in any subscript position (nothing but the semicolon alone) is interpreted as selecting all possible values for that subscript as if ⍳*N* had been supplied where *N* is the corresponding dimension extent. For example, if M is a matrix, then *M*[1;] is the first row of M, while *M*[*N*] is the same as M.

```
      A                                An array to experiment with.

  98  52   9  26  58
  34  14  91  47  15
  97  33  66  85  26
      A[3;4]                           A sub (3,4), the fourth ele-
 85                                    ment of the third row.
      A[3;4 2 5 5]                     Select several elements from
 85  33  26  26                        the third row.
      A[;3 5]                          Columns 3 and 5.

   9  58
  91  15
  66  26
      A[1 3;2 3⍴2 3 1 5 1 3] A three-dimensional array
                                       obtained by subscripting a
  52   9  98                           two-dimensional one.
  58  98   9
  33  66  97
  26  97  66
      'ABCDEF '[1 7 2 1 4 7 6 1 4]
 A BAD FAD
```

```
      )ORIGIN 0                    The meaning of subscripts
WAS 1                              changes with the index origin.
      'ABCDEF '[5 0 3 4 3 6 2 0 1]
FADED CAB
      A[2 1;3 4 0]

 85  26  97
 47  15  34
      )ORIGIN 1
WAS 0
```

A subscripted variable can be placed to the left of the assignment operator. When this is done, the variable must already possess a value of shape acceptable to the subscripting operation, and the new value, which is the right operand of the assignment operator, must have a shape precisely equivalent to that of the result of the subscript selection. Then the elements of the variable's value selected by the subscripts are assigned the new values from the corresponding elements of the assigning expression. Elements of the variable's value not selected by the subscripting operation remain unchanged.

Two important restrictions apply to assignment to a subscripted variable. First, neither the rank nor the dimensions of the value of a variable can be extended in this way; only elements that already exist can have their values changed. Second, the assignments of the various elements within a single assignment operation take place in an unspecified order; hence, if some particular element is selected several times and assigned different values, which of the values it will ultimately be found with is unspecified.

It is an error to assign character values to elements of a numeric array, or numeric values to elements of a character array. Characters and numbers cannot be mixed in any APL array.

```
      A                            Our favorite array.

 98  52   9  26  58
 34  14  91  47  15
 97  33  66  85  26
      A[2;]←⍳5                     Replace the second row.
      A[3;5]←¯10                   And the very last element.
      A                            See what we have now.

 98  52   9  26  58
  1   2   3   4   5
 97  33  66  85 ¯10
      A[2 3;⌽⍳5]←A[1 2;]           Copy the first and second rows
      A                            of A reversed into the second
 98   52    9   26   58
 58   26    9   52   98
  5    4    3    2    1
```

and third rows.
Only the first row remains unchanged.

```
      A[1;3 3 3]←6 7 8
```

Here it is unspecified whether the final value of A[1;3] is 6, 7, or 8.

REDUCTION OPERATOR ◎/ ◎⌿

The reduction operator is a composite operator consisting of a slash / preceded by any standard APL scalar operator for which a dyadic meaning is defined. For example, plus-reduction is +/ and maximum reduction is ⌈/. The scalar operators are all enumerated under "Scalar Operators", in this section. When it is necessary to discuss the reduction operator in general, it will be shown as ◎/ with the understanding that the ◎ symbol, which has no APL meaning, stands for some particular scalar operator in every actual instance.

Operator reduction behaves as a monadic operator accepting an array of any rank as operand. When applied to a vector operand, the result ◎/*V* is defined to be the same as placing the scalar operator ◎ between each of the elements of V. For example, if V is a four-element vector, then +/*V* is the same as *V*[1]+*V*[2]+*V*[3]+*V*[4] (where the subscripts have been expressed in 1-origin indexing). Thus, the plus-reduction of V is seen to be the sum of the elements of V.

When the operand of the reduction operator is a null vector, the answer is the identity element for the scalar operator involved, if it has one; otherwise, it is a domain error. The identity elements for the various scalar operators that can be used in operator reduction are shown in Table 3-1.

Table 3-1. Scalar Operator Identity Elements

operator	⊙	⊙/⍳0	operator	⊙	⊙/⍳0
addition	+	0	modulo	|	0
subtraction	+	1	binomial coefficient	!	1
multiplication	×	1	and	∧	1
division	÷	1	or	∨	0
exponential	⋆	1	less	<	0
logarithm	⍟	domain error	less-equal	≤	1
circular	○	domain error	equal	=	1
maximum	⌈	¯1.701411835*E*38	not equal	≠	0
minimum	⌊	1.701411835*E*38	greater-equal	≥	1
nand	⍲	domain error	greater	>	0
nor	⍱	domain error			

When the operand of the reduction operator is a scalar, it is treated as a one-element vector. The result of reducing a one-element vector is always simply the single element itself--even if this is an element of type or value not normally returned by the scalar operator involved in the reduction.

When the operand of the reduction operator is not a vector but a higher rank array, then the reduction is along the vectors that form the last dimension of the array. The result is an array of rank one less than the original, with shape equivalent to the shape of the original except for the disappearance of the reduced-over last dimension.

Operator reduction can also be performed along other dimensions of an array than the last. The expression ⊙⌿*A* signifies reduction by the ⊙ operator along the first dimension of the array A. The expression ⊙/[*I*]*A* signifies reduction along the I-th dimension of the array A, where I is an integer-valued expression. The dimension numbering follows the setting of the workspace index origin; i.e., the first dimension is numbered 0 in 0-origin and 1 in 1-origin. In any case, the result of reduction is an array of one lower rank with the corresponding dimension absent from the shape of the result.

The order in which the repeated scalar operations of a reduction are performed is sometimes of consequence--it is not in the case of plus-reduction--but is in the case of minus-reduction. For example, if V is a four-element vector, -/*V* gives *V*[1]-(*V*[2]-(*V*[3]-*V*[4])) which is different in value from ((*V*[1]-*V*[2])-*V*[3])-*V*[4]. The rule is that the operations are performed in right-to-left order; i.e., the first operation performed is the rightmost one, and the result of that operation becomes the right operand of the next operation to the left, and so on. As will be seen in "Scope of Operators" in this section, this is the same interpretation given to the expression *V*[1]-*V*[2]-*V*[3]-*V*[4] if it were to be typed directly.

```
      ×/6 3 ¯2 4                     Times reduction.  The product
¯144                                 of the vector elements.
      ⌊/6 3 ¯2 4                     Minimum reduction.  Finds the
¯2                                   smallest element of a vector.
      ×/⍳6                           ×/⍳N is the same as ! for in-
720                                  teger N in 1-origin.
      =/'A'                          Reduction of a single element
A                                    always yields that element.
      ⌊/''                           Reduction of a null vector
1.701411835E38                       yields operator's identity.
      +/('E'='THEN AS EVER')         Count the 'E's.
3                                    There were three of them.
      ⎕←A←?3 4⍴100                   Get an array to fool around
                                     with.
 56  23  31  83
 97  15  54  55
 70  80  88  49
      +/A                            Add up the rows.
193  221  287
      ×⌿A                            Multiply out the columns.
380240  27600  147312  223685
```

INNER PRODUCT ⊙.⊕

The inner product operator is a composite operator built up out of any standard dyadic scalar operator, followed by a period, followed by another dyadic scalar operator. When the general inner product operator is being discussed, it will be shown as ⊙.⊕ with the understanding that the ⊙ and ⊕ symbols, which have no APL meaning, are replaced by particular scalar operators in every actual instance. For example, ordinary matrix multiplication is performed in APL by the +.× inner product.

Inner product behaves as a dyadic operator. In the inner product *A*⊙.⊕*B*, the last dimension of the array A must be identical in length to the first dimension of array B. If A and B are both vectors, then the result of *A*⊙.⊕*B* is a scalar whose value is ⊙/(*A*⊕*B*). That is, the elements of A and B are pair-wise combined with the ⊕ dyadic scalar operator, and the resulting vector of answers is reduced with the ⊙ operator to form a single-element result. The name "inner product" is suggested by the fact that *A*+.×*B* is the inner product of the vectors A and B in ordinary vector algebra; that is, the components are pair-wise multiplied and the products added to give the scalar result.

More generally, if A and B are of higher rank, then each vector forming the last dimension of A (there will be many of them, as determined by the preceding dimensions of A) is paired with each vector forming the first dimension of B (again, there will be many of them, as determined by the remaining dimensions of B) to form a single element of the answer just as in the vector-vector case. (The elements of the two vectors are pair-wise combined with the ⊕ operator, and then reduced with the ⊙ operator.) The elements thus formed from all the reductions acquire a shape whose earlier dimensions are those of the array A except its last, and whose later dimensions are those of the array B except its first. The last dimension of A and the first dimension of B are lost in the reduction process. More formally then, in *A*⊙.⊕*B*, ¯1↑ρ*A* must be identical to 1↑ρ*B*, and the result *A*⊙.⊕*B* has shape (¯1↓ρ*A*),(1↓ρ*B*).

```
      ⎕←A←?3 4ρ10                A 3 by 4 matrix to ex-
                                 periment with.
  3   9  10  10
  3   8   6   1
  7   8   8   2
      A+.×1 2 3 4                Postmultiply by a four-vector
 91  41  55                      giving a three-vector.
      6 0 3+.×A                  Premultiply by a three-vector
 39  78  84  66                  giving a four-vector.
      A+.×(⍉A)                   Matrix multiplication by its
                                 own transpose.
 290  151  193
 151  110  135
 193  135  181
      7 3 14⌈.+A                 Add elements and then maximum
 21  22  22  17                  reduce.
```

OUTER PRODUCT ∘.⊕

The outer product operator is a composite operator consisting of the symbols ∘ and . followed by any standard dyadic scalar operator. When the outer product operator is discussed in general, it will be shown as ∘.⊕

with the understanding that the ⊕ symbol, which has no APL meaning, is replaced by some particular scalar operator in every actual instance.

In an outer product *A*∘.⊕*B*, each element of A is combined with each element of B using the ⊕ operator, producing (×/⍴*A*)×(×/⍴*B*) results arranged in an array of shape (⍴*A*),(⍴*B*). That is, the result of *A*∘.⊕*B* is an array with all the dimensions of A in addition to all the dimensions of B. The elements of the array are all possible pair-wise ⊕ operations on an element from A and an element from B.

```
      1 2 3 4∘.×6 3 ¯2                   Ordinary outer product.

 6   3  ¯2
12   6  ¯4
18   9  ¯6
24  12  ¯8
      3 1 4 1 5 9∘.≥(⍳10)                 A histogram of zeros and
                                                    ones.
1  1  1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0  0  0
1  1  1  1  0  0  0  0  0  0
1  0  0  0  0  0  0  0  0  0
1  1  1  1  1  0  0  0  0  0
1  1  1  1  1  1  1  1  1  0
      0 1 2 3 4 5∘.!0 1 2 3 4 5Pascal's triangle.

 1   1   1   1   1   1
 0   1   2   3   4   5
 0   0   1   3   6  10
 0   0   0   1   4  10
 0   0   0   0   1   5
 0   0   0   0   0   1
```

COMMENTS ⍝

The comments indicator ⍝ causes the interpreter to ignore it and everything following it on the line. Thus, arbitrary comments can be entered into APL statements for documentation purposes. During execution of the line, the interpreter behaves as if the comments were not there. Comments can be placed on lines typed on the terminal as well as lines within function definitions.

```
      ⍝ THIS LINE IS IGNORED.
      ⍝ SO IS THIS ONE.
      3+2 ⍝ BUT THIS ONE SHOULD GIVE US A FIVE.
5
      '->->',⍞,'<-<-'
⍝ WHAT ABOUT THIS ONE?
->⍝ WHAT ABOUT THIS ONE?<-<-
```

EXPRESSIONS, LISTS, AND STATEMENTS

Now that we have studied values and operators, we are ready to start putting them together to make expressions and finally complete APL statements. A statement in APL is either an expression or a list of expressions separated by semicolons. An expression is, generally speaking, either a value alone (a variable, a constant, or a function taking no operands), or else a value composed of other values by means of operators operating upon them. The values operated upon can themselves be expressions, of course, which is what allows expressions to become arbitrarily long. The remainder of this section more precisely defines the concepts of expressions and their evaluation.

Scope of Operators: The Right-to-Left Rule

In any expression containing more than one operator, it is important to know in what order the operations are performed; in other words one must know how much of the remainder of the expression is considered as an operand of any particular operator. For example, it makes a great deal of difference whether the APL expression *X+!Y×Z* is taken as meaning (1) X added to the factorial of the product of Y and Z, or (2) X added to the product of Z and the factorial of Y, or (3) Z multiplied by the sum of X and the factorial of Y.

In conventional algebra and some programming languages, the order of execution is determined by a fixed hierarchical ordering of operators (e.g., all exponentiations are performed before all products, which are performed before all sums). In APL, this would be unworkable due to the enormous number of different operators involved--no one could possibly remember the necessarily arbitrary ordering of so many unrelated operators. Hence, the rule is:

> In any APL expression, each operator takes for its operand everything to the right of it as written.

Thus, in any APL expression, the first operation to be performed is the right-most one, and then the result of that operation becomes the operand for the next operator to the left, and the result of that operation is fed to the next operator to the left, and so on. This rule is often referred to as the right-to-left rule.

Seemingly, the above rule specifies the scope of only right operands while saying nothing about left operands. Actually, by implication the rule specifies that the left operand of any dyadic operator be the single value immediately to its left as written. If the left operand were larger than this--if it included an operator operating upon the value--then that operator would be seen to be in violation of the right-to-left rule. So the rule is logically complete as stated.

Parentheses can be used at will to explicitly override the right-to-left rule when some other ordering is desired. Any portion of an expression enclosed within parentheses is evaluated separately and then taken as a single value during the evaluation of the remainder of the expression.

```
      1+!2×3                       Examples of order of evalua-
721                                tion.  Taken as 1+(!(2×3)).
      (1+!2)×3                     Parentheses can be used to
9                                  specify a different order.
      1+(!2)×3                     Taken as  1+((!2)×3).
7
      4×2+3                        Not 11, as you would expect
20                                 from the rules of algebra.
      3*¯1+4                       The upper minus sign is not an
27                                 operator.
      3*-1+4                       The subtraction operator is
0.004115226337
      ≠/⍳4                         ≠/⍳4 is the same as 1≠2≠3≠4,
0                                  which is 1≠(2≠(3≠4)), or 0.
      3+1÷7+1÷15+1÷1+1÷292         Continued fraction, same as
3.141592653                        3+(1÷(7+(1÷....+(1÷292)))))).
      3+÷7+÷15+÷1+÷292             The same value written with
3.141592653                        the monadic ÷ operator.
      1+2 3⍴4                      2 3 is not the same as 2,3
                                   when some operator follows it
 5  5  5                           in an expression, because the
 5  5  5                           concatenate operator takes
      1+2,3⍴4                      everything to its right as its
3  5  5  5                         operand.
      -/4,5×⍳2                     Variations on a theme.
9                                  4-5-10.
      -/(4,5)×⍳2
¯6
      (-/4),5×⍳2
4  5  10                           4,5 10.
      (-/4,5)×⍳2
¯1  ¯2                             ¯1×1 2.
```

Scope of Functions

Although functions (stored APL programs) have not yet been introduced, it seems appropriate to discuss at this point how function calls behave with respect to order of evaluation.

There are three kinds of APL functions: those that take no arguments (so-called zero-adic functions), those that take one argument (monadic functions), and those that take two arguments (dyadic functions).

From the standpoint of expression evaluation, zero-adic functions behave exactly as variables. That is, when the function name is encountered in a line of APL being interpreted, the function is evaluated to produce a value. Then that value participates in the remainder of the expression evaluation as a unit, according to the right-to-left rule. Of course, the action of evaluating a function is more complicated than that of evaluating a variable (the function must be called and its statements executed), but from the viewpoint of the expression in which the function reference is found, the function behaves as if it were a single value.

In contrast, monadic and dyadic functions behave as monadic and dyadic operators, respectively. Monadic functions, like monadic operators, accept only one operand, the right operand. Dyadic functions, like dyadic operators, accept both left and right operands, between which the function name is written. The right-to-left rule governs the scope of the operands to functions just as for operators: within any APL expression, a function takes as its right operand everything written to the right of it; a dyadic function takes as its left operand only the single value found immediately to its left.

```
      ∇THREE                        Examples of order of evalua-
FUNCTION 'THREE' NOT FOUND.         tion when function references
INPUT.                              are involved.
Z←THREE                             A zero-adic function that
Z←3∇                                always returns the value 3.
      ∇PLUS
FUNCTION 'PLUS' NOT FOUND.          A dyadic function that returns
INPUT.                              the sum of its operands (it
Z←X PLUS Y                          mimics the dyadic + operator).
Z←X+Y∇
      ∇TIMES
FUNCTION 'TIMES' NOT FOUND.         A function that mimics the
INPUT.                              dyadic multiply operator.
C←A TIMES B
C←A×B∇
      1 PLUS !2 TIMES THREE         Zero-adic functions behave as
721                                 values, others as operators.
      4 TIMES THREE PLUS 5          Same as 4×3+5.
32
      1 PLUS 2,THREEρ4              Same as 1+2,3ρ4.
3  5  5  5
      THREE TIMES ⍳ THREE PLUS THREE   Same as 3×⍳3+3.
3  6  9  12  15  18
```

The input-output symbol ⎕ also behaves exactly as a variable. When used for input, it is evaluated to produce a value. When used for output, it is assigned a value, which is printed and also passed on as the value of the ⎕ during the remainder of the expression evaluation.

```
V←⎕←+/A×⎕+2
```

Here the operators are performed in the order + × + ← and ←. The input occurs prior to the first +, the output occurs between the two assignments.

Lists

A list is a series of expressions separated by semicolons. There are two uses for lists in APL. First, an APL statement itself can be a list of expressions instead of just a single expression. Second, a list of expressions written between square brackets and placed following any value is a subscripting operation on that value.

Statements

A statement in APL is either an expression alone or else a list of expressions separated by semicolons. When an APL statement consists of only a single expression, the behavior of the interpreter depends upon whether the last operator executed in that expression is a left arrow (the assignment operator). If the last operation performed is an assignment, then the interpreter does not print the result of the expression evaluation. If the last operation performed is other than an assignment operation, however, the interpreter prints the value of the expression (if it has one). This printing of the values of lines that do not end in assignments is the commonest method of performing output in the APL language.

The previous paragraph suggested that an expression might not have a value. This is possible only in three cases: (1) the expression is vacuous (an empty line, for example); (2) the expression ends with the jump operator (the jump operator → has no result and can be used only as the final operation in a statement); or (3) the expression consists of a call on a function that returns no result. In each of these three cases, the result of the expression evaluation yields no value, hence nothing is printed. These cases of an expression with no value must be carefully distinguished from that of an expression with a null value.

The null value (an array with zero elements) is a perfectly well-defined and well-behaved value, and it prints as a blank line.

When the APL statement consists of a list of expressions, the values of the expressions are always printed, even when some or all of the expressions conclude with the assignment operator (it is a *VALUE ERROR* if any nonvacuous expression has no value). The values of the expressions are printed, one after the other, with no additional spaces or newlines placed between them. Whatever spaces and newlines may be a portion of the representations of the values themselves are, however, printed as usual. Since some of the expressions may have character values and others numeric, this is a way of forming output with mixed types appearing on one line.

```
      V←1 2 3                       Assignment, no value printed.
      V                             No assignment, value of
1 2 3                               expression printed.
      V;4 5                         A list of expressions; values
1 2 34 5                            are printed side by side.
      V;'  ';4 5                    Explicit spaces included.  An
1 2 3  4 5                          example of mixed-type output.
      S←'HELLO';' THERE.'           Lists always print, even when
HELLO THERE.                        there are assignments made.
      S;2 3⍴⍳6;S                    No extra control characters
HELLO                               beyond those of the values
 1 2 3                              themselves are added (recall
 4 5 6HELLO                         newline before each plane).
      '6÷3 = ';6÷3                  Another example of mixed-type
6÷3 = 2                             output.
      ∇RESULTLESS                   Define a function that
FUNCTION 'RESULTLESS' NOT FOUND.    returns no result.
INPUT.
RESULTLESS
V←V+1∇                              A call on RESULTLESS results
      RESULTLESS                    in no printing.
      0⍴0                           A null value, however, prints
                                    as a blank line.
      V                             The call on RESULTLESS did
2 3 4                               produce this side-effect.
      W←3÷×/0 1∘.+V;                A vacuous expression tacked to
0.125 0.05                          the end of a statement just
                                    to force printing.
```

Dependence upon Unspecified Ordering

While the right-to-left rule specifies the order that operators within an expression are executed, it is important to note some things that it does not specify. There are a number of steps taken during the evaluation of an APL statement that occur in an explicitly unspecified order. Any program that depends upon a certain order of occurrence for any of these unspecified cases is incorrect APL. Such incorrect programs cannot be expected to give consistent results when run on different implementations of the language, or even when run on later versions of any given implementation.

The various expressions within a list are evaluated separately from each other. No portion of any expression participates as an operand of any operator in any other expression of the list. Hence, the semicolons of a list act as barriers to the scope of the right-to-left rule, which holds only within single expressions. There is no specification or definition of the order in which the several expressions are evaluated.

```
D←A[I+3;I←I+1]
```

Examples of unspecified order of evaluation. Here, one expression in a list assigns a new value to the variable I. Another expression uses I. It is unspecified whether the old or the new value of I is obtained in the first subscript.

```
'THIRTY = ';N←3;N←0
```

Here the printed result is definitely 30, but whether 3 or 0 will be found as the value of N following this statement is unspecified.

```
⎕←'ONE';⎕←'TWO'
```

This statement causes three lines of output, the third of which is definitely the string *ONETWO*, that being the final result of the list evaluation. Whether the earlier two lines come in the order *ONE TWO* or *TWO ONE* is unspecified.

The right-to-left rule specifies carefully the order in which operations are performed, but it does not specify the order in which the various operands of a single operator are evaluated or made ready for the operation.

```
(A←A+5)×(3÷A)
```

What does the right-to-left rule say about the order of operations here? It says that the addition must be performed before the assignment, which (because of the first pair of parentheses) must be done before the multiply. The divide must also be done before the multiply (the second pair of parentheses are unnecessary). But there is no specification of the relative order of the assignment and the divide, so whether the old or the new value of A will be used in the divide is unspecified.

`(I←3)⌽[I←1]3 4⍴⍳I←12`

The printed result of this statement is definitely the value of `3⌽[1]3 4⍴⍳12`, but whether the value of I will later be found as 3, 1, or 12 is unspecified.

Even when an operand of an operator is a single variable alone, it still needs evaluation. That is, at some time or other, the current value of the variable must be looked up and made available to the operator. The exact time at which the evaluation is done can lead to different results if the value of that variable is changed elsewhere in the same statement.

`A×A←3`

Here it is unspecified whether the left operand of the multiply is the old value of A (whatever it was) or 3. Note that the assignment of 3 to A definitely takes place before the multiply. However, it is not known whether that was before or after the extraction of the value of A for the left operand.

`⎕÷⎕`

In this example it is not known whether the first number input will be divided by the second, or the second divided by the first, because it is not specified whether the left or right operand is evaluated first. The input-output symbol is not an operator; it behaves exactly as a variable or a zero-adic function.

In summary, dependence on unspecified ordering occurs only in statements containing, as other than the leftmost (i.e., last) operation, assignments that change values of variables used elsewhere in the same statement. In this connection, any input-output activity must be considered as a form of assignment, as must calls on functions that perform such assignments or input-output. Any program that depends on ordering other than that implied by the right-to-left rule must be considered an incorrect program. A dependency of this nature can be very difficult to discover when one has access to only one implementation of the language, of course, since most implementations tend to be at least self-consistent and repeatable in choosing some ordering.

FUNCTIONS

Functions are stored APL programs. They are created and modified with the APL function editor, which is fully described in Section V. More precisely, a function is a stored sequence of APL statements preceded by a special line, called the function header line, which defines several important properties of the function; these are its name, the number of arguments it accepts, whether it may return a result, and whether it has within it any local variables.

The names of all functions defined within a workspace can be listed with the *)FNS* system command, which is discussed under "Name Table Management" in Section V.

Arguments and Results

Every APL function accepts a fixed number of arguments, either zero, one, or two. Functions that accept no arguments are said to be zero-adic; those that accept one argument are said to be monadic; and those that accept two are said to be dyadic.

The header line of every function also specifies whether the function may return a result. If the header line indicates that it may, then the function may sometimes return a result and sometimes not. If the header line indicates that it may not, then it may never return a result.

From the standpoint of usage, a zero-adic function behaves exactly like a variable. If a function name is encountered during the evaluation of an expression, the function is evaluated (by bringing forth its stored definition and executing the lines within it), and any result passed back by the function is taken as its value, and then the expression evaluation resumes. If the function cannot or has not elected to return a result, and the context of the function reference is such as to require a value for the expression evaluation to resume, then a *VALUE ERROR* message results, just as if a reference was made to a variable which had not been assigned a value.

From the standpoint of usage, monadic and dyadic functions behave exactly like monadic and dyadic operators respectively. That is, monadic functions accept only one operand, a right operand. Dyadic functions accept both a left and a right operand, between which the name of the function is written. The right-to-left rule governs the scope of operands and the order of execution of functions just as for operators: each function accepts as its right operand the entire remainder of the expression following it; in the absence of parentheses, operators and function calls within an expression are performed in right-to-left order. Again, a function evaluated in a context that demands a returned value must provide one; otherwise a *VALUE ERROR* occurs.

The manner in which the function header line specifies the properties of a function can now be described. The function header line is in the form of a sample call on the function. For example, the header line *Z←X FN Y* declares the function whose name is *FN* to be a dyadic function whose left argument is known by the name *X* within the function definition, whose right argument is known by the name *Y* within the function definition, and which may return a result known by the name of *Z* within the function definition. Whatever parts of the header line do not apply to the particular function being defined are simply omitted. So, the exact format of a header line is: a name followed by a left arrow, if the function is to be able to return a result; a name, if the function is to have a left argument; the function name itself, which is the only nonoptional constituent of the header line; and finally, another name, if the function is to have a right argument. All names in the function header line must obey the APL rules for constructing names (see "Names" earlier in this section).

Since each APL function has a choice of three argument options (0, 1, or 2 arguments) and two result options (0 or 1 result), there are six different kinds of function header lines. Examples of the six kinds are:

```
Z←L NAME R          (dyadic, result)
Z←NAME R            (monadic, result)
Z←NAME              (zero-adic, result)
L NAME R            (dyadic, no result)
```

```
        NAME R                (monadic, no result)
        NAME                  (zero-adic, no result)
```

Some examples of some simple functions are shown on pages 3-58 and 3-59.

Local and Global Variables

When a function is called, variables with the names mentioned in the header line come into existence (that is, the result variable and the two argument variables, in the fullest case). The value of the left operand from the calling expression is then assigned to the left argument variable; the value of the right operand from the calling expression is assigned to the right argument variable; no assignment is made to the result variable.

These variables, called local variables, are created even if variables of the same name are already in existence at the time of the function reference. Throughout the execution of the function, any reference to one of the local variable names is considered a reference to the corresponding local variable, and not to any variable that was in existence at the time of the function call. Thus, the local variable names effectively mask off any access from within the function to possibly similarly-named variables outside the function.

During the execution of the function, local variables can be used freely, and they behave exactly as normal variables. Their values can be changed at any time by the assignment operator. Note that changing the value of an argument local variable has no effect upon the values of variables outside the function that may have been used to express the corresponding operand of the function; the values of the operands are copied into the argument local variables at the time the function is invoked and there is no further reference to the actual operands. As with any other variable, a reference to the value of a local variable before it has been assigned one is an error (this error, of course, never occurs with argument local variables, because they start off with values automatically assigned to them).

An APL function returns a result by assigning a value to its result local variable. Whatever value is found for the result local variable at the time the function execution terminates is taken as the value of the function and returned to the calling expression. If a function never assigns a value to its result local variable, then the function will be found not to have a value in the referencing expression (whether or not this is an error depends upon whether the referencing expression demands a value at this point).

When a function terminates, its local variables disappear. Other than the extraction of the value of the result local variable prior to its erasure, no history or trace is left of its existence. Variables that had been rendered inaccessible because their names were masked by similarly named local variables again become accessible, with their old values unchanged.

The masking of variable names by similarly named local variables nests to any depth within function calls. That is, if a function, say F, is invoked with certain local variable names, say A, B, and C, then those names mask off any access to other variables similarly named. Until the function F terminates, any references to A, B, or C will be to F's local variables, even from within functions called by the function F (unless local variables in the newly called functions further mask from accessibility the local variables of F). For example, if F calls a function G whose local variables are C and D, then any reference to A or B within G will be taken as a reference to F's local variables. Any reference to C or D will be taken as a reference to G's local variables (the local name C masks off access to F's local variable C). Any reference to still another name, say E, will not be found as local to either F or G, but will be searched for in whatever function called F, and so on. If a referenced name is ultimately found to be local to no function invocation, then it is said to be global, and a permanent variable of that name is accessed or created. Global variables are not erased when functions terminate.

In addition to the arguments and the result, any number of other names can be listed in the function header line so that they become local variables. Such additional names are listed, each preceded by a semicolon, after the sample function invocation which has already been discussed. For example, the header line *Z←F X;A;B* refers to a monadic function named F, with argument X and result Z, and additional local variables A and B. Such variables can be used within the function to store intermediate results without danger of destroying the values of any of the caller's variables. Labels, discussed on page 3-62, are automatically local variables and should not be entered into the local variable list.

The *)SIV* system command can be issued to obtain a listing of current function invocations, together with the variable names that are local to each function. Thus, the *)SIV* output can be used to determine unambiguously the referent of any particular name. The *)SIV* command is discussed under "Name Table Management" in Section IV.

```
      ∇TAX3                      Some very simple examples of
FUNCTION 'TAX3' NOT FOUND.       functions.  The function edit-
INPUT.                           or is discussed in Section V.
TAX3                             Header line: no args or result
'PLEASE TYPE AMOUNT OF SALE'     for this function.

'TAX IS ';0.01×⌊0.5+3×⎕∇
      TAX3                       Call the function.
PLEASE TYPE AMOUNT OF SALE       Output produced by executing
⎕:                               function.  Function requests
      48.55                      input and gets it.
TAX IS 1.46                      From last line of function.
      ∇TAXN                      Now a function of one argument
FUNCTION 'TAXN' NOT FOUND.       but still no result.
INPUT.
TAXN R                           Header: right argument R.
'AMOUNT..'                       Still only a two-line function
R;' PERCENT TAX IS ';0.01×⌊0.5+R×⎕∇    as before.
      TAXN 5                     Execute it.
AMOUNT..
⎕:
      10.22
5 PERCENT TAX IS 0.51
      ∇TAX                       OK.  Now we're ready for the
FUNCTION 'TAX' NOT FOUND.        big leagues.  Here is a dyadic
INPUT.                           function that returns a re-
T←R TAX AMT                      sult.  Takes rate, amount.
T←0.01×⌊0.5+R×AMT∇               Only one line to this one.
      3 TAX 48.55                This time the function itself
1.46                             has no output, but it gives a
      5 TAX 10.22                value to the calling expres-
0.51                             sion.
      (3 TAX 48.55)+(5 TAX 10.22)  And those values can be
1.97                             further operated upon.
      (3 TAX 48.55)+5 TAX 10.22    The parentheses above
1.97                             were not needed.
      3 TAX 48.55+5 TAX 10.22      But the other set were.
1.47                             This is tax on 48.55+0.51.
      5 TAX 20 TAX 83.97         A 5 percent surtax on a 20
0.84                             percent tax.
      ∇MX
FUNCTION 'MX' NOT FOUND.
INPUT.
Z←MX X                           A simple function to multiply
Z←M×X∇                           by the value of M.
      M←3                        We'd better provide a value.
      MX 5                       Try M times 5.
15
      ∇CALL_MX
FUNCTION ' CALL_MX' NOT FOUND.
INPUT.
```

```
R←CALL_MX Q
R←MX Q∇
      CALL_MX 5
15
      ∇CALL_MX C 3 /Q/M/ ∇
R←CALL_MX M
R←MX M
      CALL_MX 5
25
      M
3
      MX 5
15
```

Another simple function, whose only job is to call MX.
Q passes 5 to MX, which returns 15, as before.
But wait, let's change the Q to an M in CALL_MX and see what happens.
This time MX picks up the M which is local to CALL_MX.
The global M is unchanged.

MX works fine here.

Branching

Branching, or the ability to alter the normal sequential flow of control from one statement to the next, is an essential feature of any programming language. In APL the jump operator → performs the jobs of conditional and unconditional branching as well as function return. In order to understand how it works, it is first necessary to learn about statement numbers.

STATEMENT NUMBERS

Each executable statement of an APL function is numbered, beginning with one for the line after the header line (the header line itself is sometimes referred to as statement zero, but this has no real meaning because it cannot be branched to). One line is normally one APL statement, but occasionally several lines combine to make one statement due to newline characters imbedded in character constants.

```
      Z←I OP R
[1]   →2××I,Z←R+1+R=2
[2]   Z←2
[3]   →××R←R-1
[4]   →3××Z←(I-1) OP Z
```

An example of statement numbering. The numbers in the brackets show the statement numbers considered to be associated with each statement of the function.

```
      Z←N ROUND V;S
[1]   S←⌊0.5++/Z←1|V←V÷N
[2]   Z←N×(S≥⍳⍴V)[⍋⍒Z]+⌊V
```

Another example. Note that the header line is not counted.

```
      ADD3;A
[1]   A←3⍴~×I←30
[2]   →2×⍳×I←I-⍴A←A+⎕
[3]   A
```

A final example.

THE JUMP OPERATOR →

The jump operator → is a monadic operator which accepts as its operand a scalar or vector of integers. If the operand is a scalar, it is simply the statement number of the statement to which to jump. If the operand is a vector with at least one element, then the first element is taken as the target statement number (any remaining elements are ignored). If the operand is a vector with no elements, then no jump is performed--control passes on to the next statement in sequence.

The jump operator is unusual in that it has no resultant value; hence, it must be the last (i.e., left-most) operator of any statement in which it is found.

Since the operand of the jump operator can be an arbitrary APL expression, the jump target can be calculated instead of merely being a constant. Thus, the same operator → becomes a conditional or computed branch at will and serves as an unconditional branch when necessary.

The target statement of the jump operation always lies in the same function as the jump itself. There is no way in APL to jump out of one function and land on a specified statement in another.

→3	Unconditional branch to statement 3.
→*I*	Branch to statement I. The next statement executed will be determined by the current value of I, which must be a numeric integer.
→(*N*>12)/7	Can be read as "if N is greater than 12, then branch to statement 7". Work it out. Remember a null vector is no branch.
→7×⍳*N*>12	In index origin 1, this can be read as "branch to statement 7 if N is greater than 12".
→7+⍳*N*>12	This is the way it is written in origin 0.
→*S*⌽7 13 32	Switch, or dispatch table. Branch to statement 7, 13, or 32 depending upon whether S is 0, 1, or 2.

FUNCTION RETURN

Any operand of the jump operator other than a numeric integer is an error, but a jump to an integer that is simply not a statement number of the current function is legal, and is considered to be a function return. That is, the function ceases execution, any value found to be associated with its result local variable is passed back as the value of the function, all local variables are erased, and the interpretation of the calling expression resumes from the point at which it invoked the function.

Thus, there are two ways in APL to return from a function: to run off the end of the function sequentially, or to jump to a nonexistent statement number. Statement number 0 is conventionally used in a function return jump, but actually any integer smaller than 1 or greater than the largest statement number present in the function can be used.

```
         ∇FACT                         A factorial calculator will
FUNCTION 'FACT' NOT FOUND.             illustrate a conditional re-
INPUT.                                 turn jump.
F←FACT I                               Header line, one arg, result.
F←1                                    Answer is 1 if return now. [1]
→3×I>1                                 Return if I≤1, else go on. [2]
F←F×I                                  Multiply in this integer.  [3]
I←I-1                                  Step down one.             [4]
→2∇                                    And go to test if done.    [5]
         FACT 3
6
         ∇DRILL                        Addition drill, types random
FUNCTION 'DRILL' NOT FOUND.            small integers to be added by
INPUT.                                 the user.
DRILL;I;J;K                            No args, three local vars.
I←?10;'+';J←?10;' = ?'                 Type the question.         [1]
→3×'Q'≠K←⎕                             Read answer, stop if 'Q'.  [2]
→⍳K=I+J                                Go to 1 if correct.        [3]
→2,ρ⍞←'SORRY, TRY AGAIN.'∇             Else gripe and go to 2.    [4]
         DRILL
6+3 = ?
⎕:
         9
10+5 = ?
⎕:
         8
SORRY, TRY AGAIN.
⎕:
         15
2+7 = ?
⎕:
         'Q'                           Enough for now.
```

STATEMENT LABELS

As can be seen from the above examples, counting statements to determine their statement numbers is awkward at best. To overcome this difficulty, statement labels are provided. A statement label is a name preceding any statement within a function and set off from it by a colon. All statement labels are considered to be local variables of the function (they need not be listed again in the header line) that are automatically assigned the values of their respective statement numbers at the time the function is invoked.

It is recommended that statement labels rather than absolute statement numbers always be used for branching. In addition to being less error-prone and more readable, labels continue to give correct results when new statements are added to a function.

```
      ∇DRILL2
FUNCTION 'DRILL2' NOT FOUND.
INPUT.
DRILL2;I;J;K
NEW:I←?10;'+';J←?10;' = ?'
AGAIN: →CHECK×'Q'≠K←⎕
CHECK: →(K=I+J)/NEW
→AGAIN,ρ⎕←'NICE TRY, BUT NO CIGAR.'∇
```

Another version of DRILL, this one using labels. During the execution of DRILL2, the variables new, again, and check have the values 1, 2, and 3, respectively. The interpreter has done the necessary counting for us.

```
      ∇F
FUNCTION 'F' NOT FOUND.
INPUT.
          ANS←F INPUT
          ANS←1
LOOP:     →NEXT×INPUT>1
NEXT:     ANS←ANS×INPUT
          INPUT←INPUT-1
          →LOOP∇
      F 4
24
```

Let's also rewrite the factorial function with labels. Horizontal tabs used to space the program and improve readability.

After their initialization at function invocation time, statement label local variables behave just as other local variables. Nothing prevents the assignment of new values to them, or the usage of their values for purposes other than branching. However, it is doubtful if such practices are useful, and they can make a program difficult to understand and dangerous to modify.

Recursion

Since each invocation of a function creates new instances of its local variables, even if similarly named variables are already in existence, functions are fully recursive in APL. Any function can be invoked at any time, regardless of it being already invoked. Functions can call themselves, or call others that will ultimately call themselves. The rules for determining referents of local names ensure that each invocation of a function has access to its own local variables, and that access to the caller's variables is restored upon a return.

```
      ∇F̲
FUNCTION 'F̲' NOT FOUND.
INPUT.
A←F̲ N
→(N≤A←1)/0
A←N×F̲ N-1∇
      F̲ 5
120
```

The factorial function written recursively. The answer is initialized to 1 in case the argument is 0 or 1 (immediate return). Otherwise, call ourself to do N-1 and multiply.

```
      ∇WINS
FUNCTION 'WINS' NOT FOUND.
INPUT.
W←N WINS K;X
W←(1,N)⍴⍳N
→(K<2)/0
X←(N*K-1)×¯1+⍳N
W←X∘.+N WINS K-1
W←(,W),(,2 1 2⍉W),(,2 1 2⍉⌽W),(,(⍳N*K-1)∘.+X)
W←((0.5×-/(N+2 0)*K),N)⍴W∇
```

A recursive program to calculate the winning lines in N*K Tic-Tac-Toe (K-dimensional, with N on a side). Calls itself to do the wins of one less dimension, then figures out how to spread each of them over the highest dimension.

```
      4 WINS 1

 1  2  3  4
```

Ultra-trivial Tic-Tac-Toe: one line of four cells. There is only one win.

```
      3 WINS 2

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

Normal 3-by-3 game. There are eight wins.

Extension of Scalar Functions to Arrays

Since arrays as well as scalars are legal APL values, arrays can be passed as arguments to functions. Whether an array argument makes sense in any particular function depends on what operations it is subjected to in the body of the function. One interesting special case is that of a function definition that uses only scalar operations. Scalar operations are precisely defined earlier in this section, but, generally speaking, they are those operations that manipulate a single scalar operand or a pair of operands at a time. Since scalar operators extend element-by-element in their activity to arrays, any function that uses only scalar operators within its definition also extends automatically in the same way to handle array arguments.

```
      MX 3 8 11.53 ¯17.3          MX uses only scalar operators
9  24  34.59  ¯51.9               so it extends to arrays.
      3 TAX 0.79 1.33 5.95        The scalar 3 is replicated to
0.02  0.04  0.18                  match the right operand.
      3 5 7 TAX 0.79 1.33 5.95  Element-by-element, 3% of
0.02  0.07  0.42                  0.79, 5% of 1.33, 7% of 5.95.
      3 5 TAX 1 2 3 4             Here shapes do not match.  An
LENGTH ERROR.                     error report occurs at the
TAX[1] T←0.01×⌊0.5+ρ×AMT          first conflict.  APL marks the
                       |          operation in difficulty.
      )SIV                        Display state indicator and
TAX[2]*          T         AMT       R    local variable names.
      R                           Local variables are accessible
3  5                              because the function is cur-
      AMT                         rently invoked.
1  2  3  4
      R←3 5 7 9                   Supply a better value for R
      →1                          and jump back to line 1.
0.03  0.1  0.21  0.36             Function completes normally.
```

Library Functions

A number of generally useful functions for manipulating workspace parameters are available in the public workspace *>APL>WSFNS*. Any of these functions can be copied with the *)COPY* system command from the public workspace into any user workspace. The *)COPY* command is described under "Saving and Reloading Workspaces" in Section IV. The available functions are:

DELAY	*ORIGIN*	*SFEI*
DIGITS	*SETLINK*	*SFII*
E	*SFCI*	*WIDTH*
FUZZ		

Each of these functions performs the same activity as its similarly named system command, except that the function has the additional capability of being called from within an APL program. The system commands are described in detail in Section IV, to which reference should be made to obtain the exact meaning of each library function.

All the functions are monadic and expect an APL expression of correct type as argument (*E*, *SFCI*, *SFEI*, and also *SFII* take a character-string argument; the others, a numeric argument). Functions corresponding to system commands that print the former value of a workspace parameter (all except *DELAY* and *E*) instead return that value as the result of the function (*DELAY* and *E* have no result).

The *DELAY* library function has no system command counterpart. It takes a numeric integer argument and delays execution for that many seconds. It uses "timer_manager_$sleep", and returns no value.

In Multics APL, the library functions are implemented as external functions, which is why their names are tagged with an asterisk it a *)FNS* listing. Further details about external functions can be found in Section IV.

```
      )COPY >APL>WSFNS
SAVED 10/12/72  1202.0
      )FNS
 DELAY*  DIGITS* E*                  FUZZ*   ORIGIN* SETLINK*
SFCI*   SFEI*   SFII*   WIDTH*
      DIGITS 19                Same as )DIGITS 19.
10
      *1
2.7182818284590045235
      I←ORIGIN 0               Set the workspace index origin
      ⍳4                       to 0 and remember the old set-
0  1  2  3                     ting.
      I←ORIGIN I               Restore the old setting.
      D←'0123456789'[1+(6⍴10)⊤⌶25]        Get date, mmddyy.
      D←D[1 2],'/',D[3 4],'/19',D[5 6]   Get mm/dd/19yy.
      T←'0123456789'[1+(3⍴10)⊤⌊(⌶19)÷60) Get cpu seconds.
      E 'FO APL_LOG;IOA_ DATE:',D,',TUSED:',T,'SEC.;CO'
      E 'PR APL_LOG 1 1'                 Log usage.
date:10/17/1972,tused:006sec.            Print log entry.
      I←WIDTH ⎕←WIDTH 100      Print the workspace page width
80                             without changing it.
```

ERROR HANDLING

Many different kinds of errors can be detected by the APL processor. When an error is detected during the execution of a statement, APL aborts further interpretation of the statement, types a message naming the kind of error, and types a copy of the statement being interpreted with a marker (a vertical bar) placed under the character that was being processed at the time of the error. The interpreter then reads the terminal normally to obtain further statements to interpret. Since any APL statements can be typed at this time, APL is its own debugging language:

variables can be displayed; values can be changed with the assignment operator; the suspended program (if it was a function being interpreted rather than a line from the terminal) can be restarted with the jump operator.

Error Messages

The most common error messages are listed below, each with a short definition.

CHARACTER	The user typed an illegal overstrike combination or an undefined escape sequence. One of the constituents of the combination is marked with the error marker. No interpretation or other action is taken upon lines that have illegal characters in them.
COMMAND	The user typed an incorrect system command. If the marker is under the command name, then that name could not be recognized as a legal system command; perhaps it is misspelled. Otherwise, the command itself is legal but something is wrong with the arguments supplied to it. Argument requirements vary considerably from command to command; the user should refer to the detailed description of the particular command in Section IV.
DEPTH	Too many function invocations are nested in the state indicator. Perhaps some function is calling itself recursively without limit, or perhaps too many error suspensions have been allowed to accumulate in the state indicator. No source line is printed with this error message, because the problem is not associated with any specific statement. The interpreter has cut the state indicator back to the previous console suspension point and suspended there again. The user must inspect the state indicator with the)*SI* system command and possibly use the → operator to discard useless function invocations being held there.
DEFN	Function definition error, or an illegal call on the function editor. This can be due to an illegally constructed function name, an attempt to create a function with the same name as some already existing group or global variable, an attempt to edit a pendent function (see "Restarting Following a Function Edit" in this section), an attempt to edit an external function or to define one with an illegal pathname (see

"External Functions" in Section IV), or the occurrence of the editor-call character ∇ in an incorrect context.

DOMAIN	An operation demands an operand of a different type than supplied, or the value supplied is out of the range of values meaningful to the operator. The operator is marked.
INDEX	An index or subscript value was outside the bounds or dimensions of the subscripted array. The error marker is under the left bracket of the subscript list.
LENGTH	The lengths of the left and right operands of an operator did not match. The operator is marked.
RANK	The number of dimensions of the left and right operands of a scalar operator did not match, or some operator demanding an operand of a specific rank did not find it. The operator in question is marked.
SYNTAX	An illegally-formed statement was encountered. Some examples of syntax errors are mismatched parentheses and brackets, missing operators, and missing operands. Everything to the left of the error marker is acceptable, but the item marked with the error marker cannot logically follow what preceded it, or, if the the error marker is off the end of the statement, something more was expected.
VALUE	A variable's value was called for before a value was assigned, or a function did not return a result when one was required. The name lacking a value is marked.
WS FULL	Workspace full. The workspace could not accommodate an item for which the interpreter needed space. As with a *DEPTH* error, no source statement is printed with a *WS FULL* error, and the state indicator may need to be cut back to the previous console suspension point. This error can be caused by a genuine lack of room in the workspace, which can be remedied by erasing needless objects; or by an attempt to create an APL value larger than a Multics segment, which has no remedy (though a workspace can grow to many segments in size, each individual value must fit wholly within one segment). Remember that local variables and saved function invocations consume space too; they can be erased by clearing the

state indicator.

Due to the extremely large workspace size possible on Multics, *DEPTH* and *WS FULL* errors do not occur frequently.

All error messages from Multics APL are written onto the stream "user_i/o". Furthermore, following an error, Multics APL no longer reads and writes the streams "user_input" and "user_output", but instead performs all its input/output on the stream "user_i/o". The stream attachments are not disturbed; the implementation is such that the former streams are simply not used. At present there is no way to cause an invocation of APL to return to using "user_input" and "user_output". The only way presently of returning to "user_input" and "user_output" is to save the current workspace with the)*SAVE* system command, exit from APL with the)*Q* system command, re-invoke APL (the new invocation will again read "user_input" and write "user_output"), and finally get back the workspace with the)*LOAD* system command.

```
      ι3.4
DOMAIN ERROR.
      ι3.4
      |
```

Domain error. The operand of the ι operator must be an integer. The operation is marked.

```
      22+ι5 6 7
RANK ERROR.
      22+ι5 6 7
         |
```

Rank error. The operand of the ι operator must be a scalar, not an array. Again the ι is marked.

```
      'ABCDEF'[1 4 9 3]
INDEX ERROR.
      'ABCDEF'[1 4 9 3]
              |
```

Index error. The 9 is out of range. Again, the operator is marked.

```
      +/ιNOTTHERE×5
VALUE ERROR.
      +/ιNOTTHERE×5
         |
```

Value error. The variable *NOTTHERE* has no value yet.

```
      (ι1)+(((6ρ10)⊤(⌶25)÷60)[3 4]
SYNTAX ERROR.
      (ι1)+(((6ρ10)⊤(⌶25)÷60)[3 4]
                                  |
```

Syntax error. The interpreter would like a closing parenthesis.

```
      1 0 1\'ABC'
LENGTH ERROR.
      1 0 1\'ABC'
           |
```

Length error. The number of ones in the left operand is not equal to the length of the right operand.

Pendent Statements and the State Indicator

To fully explain the action of the APL processor on an error, the concepts of pendent and suspended statements and the state indicator must be defined.

When the APL interpreter is executing an APL statement, most often it can process the statement completely from beginning to end without turning its attention to any other APL statement in the process. However, if the statement contains a reference to a function or to the evaluated input symbol ⎕, then the evaluation of the given statement cannot be completed until the interpreter has evaluated one or more additional statements (the statements in the body of the function definition, or the statement typed as input). In these cases, the further execution of the given statement is said to be pendent upon the successful completion of the function or evaluated input line invoked by the given statement.

Many things must be safely stored away by the interpreter when a statement becomes pendent so that its evaluation can be resumed accurately once the invoked statements have been completed. An area of the workspace called the state indicator is reserved for this information. Whenever a statement becomes pendent, the APL processor creates an item for the state indicator stack containing all the relevant information (the state indicator behaves as a stack because items on it are needed on a last-in, first-out basis).

Since evaluated input statements as well as statements within a function can contain further function and evaluated input references, the interpreter often needs to make several successive entries to the state indicator stack. For example, in a nest of several function calls, the outermost statement is pendent upon completion of the first function call, and some statement in that function is pendent upon the completion of another call, and so on. During execution of the innermost function, the state indicator remembers the partial evaluations of all the pendent statements. As each invoked function returns, the corresponding item is recalled from the state indicator stack so that the execution of the invoking line can be resumed. When eventually all functions return, the state indicator will be empty again.

Suspended Statements

The exact state of the APL interpreter following an error message can now be clarified.

When an error occurs during the interpretation of a line, the interpretation of the line stops, any temporary storage consumed by intermediate values incident to evaluating the line is reclaimed, and the line is discarded. If the portion of the line that was already evaluated at the time of the error included assignments to variables or input/output activity via the ⎕ symbol, then such assignments and input/output activity will have taken place, and the interpreter does not and cannot undo them.

If the line in error was a line read from the console (as either immediate or evaluated input), then there is no change to the state of the interpreter in response to the error. The interpreter again requests the input it was awaiting by typing either the immediate go-ahead signal (six spaces) or the evaluated go-ahead signal (⎕:).

If the line in error was a line from the body of a function definition, then execution of the function is temporarily <u>suspended</u>. This means that statements cease to be drawn from the function definition for execution. Instead, an entry is made to the state indicator remembering the number of the next statement of the function to be executed, and the interpreter reverts to reading statements from the console. The state indicator contains the saved partial evaluations of the pendent statements that ultimately led to the function line containing the error, as well as to the newly made entry for the function suspension. Note that the suspension entry on the state indicator is different from the pendent entries in that it does not include the partial evaluation of any line, and that it represents a point at which control was given to the user, by letting him execute statements from the console as immediate input.

Following an error and a suspension, arbitrary APL statements can be typed, and they execute completely normally. In particular, the normal rules for establishing the referents of names are still in effect: the suspended function's local variables are still in existence, hence they are fully available for display and alteration. Furthermore, the local variables of the pendent functions that invoked the suspended function, if any, are in existence too, and those that are not masked by similarly named variables in later invocations are equally accessible. Since the full power of APL is available for manipulating these variables, APL is truly its own debugging language.

The)*SI* and)*SIV* system commands are useful for printing out the contents of the state indicator following an error. The)*SI* command prints the name of each function saved in the state indicator together with the statement number on which it is to resume execution. The command)*SIV* is the same except that it also lists the names of the local variables associated with each function invocation. The listing is in the order of most recent to least recent, and the suspended function is tagged with an asterisk to distinguish it from the pendent functions.

```
      ∇PLUS                       Define a function for the pur-
FUNCTION 'PLUS' NOT FOUND.        poses of illustrating errors
INPUT.                            and the state indicator.
C←A PLUS B
C←A+B∇
      )SI                         The state indicator is empty.
      1 2 PLUS 3 4 5              Cause an error in the func-
LENGTH ERROR.                     tion.  The interpreter marks
PLUS[1] C←A+B                     the operation in error and
           |                      suspends.
      )SI
PLUS[2]*                          The * means PLUS is suspended.
      )SIV                        Once more with the local
PLUS[2]*        C        A              B   variable names.
      A                           The variables are accessible.
1  2
      B
3  4  5
      ⌽A+3 PLUS 8×2↑B             Any APL statement can be
37  28                            executed.
```

Since any arbitrary statement can be entered following a suspension, further function invocations and evaluated inputs can be performed. Such additional invocations cause additional entries to be made in the state indicator to record the saved partial evaluations of the pendent invoking lines. The new entries are stacked on top of the old entries, because they will be needed first when the new invocations terminate and control unwinds back to the previous suspension point again.

If some new invocation runs into trouble and causes an error message, the new invocation will suspend also. Thus, the state indicator may contain several suspensions stacked on top of one another. The paragraphs following the example below show to dispose of how to dispose of these suspensions.

```
      ∇SIXPLUS                    Continuing the above example,
FUNCTION 'SIXPLUS' NOT FOUND.  here we define two more
INPUT.                            functions.  SIXPLUS calls PLUS
Z←SIXPLUS X                       to add six to its argument.
←6 PLUS X∇
      ∇QTIMES                     QTIMES multiplies its argument
FUNCTION 'QTIMES' NOT FOUND.   by ⎕, that is, by whatever
INPUT.                            expression is typed in for the
B←QTIMES A                        value of the ⎕ symbol.
```

```
B←⎕×A∇
      )SIV                          The state was unaffected by
PLUS[2]*        C       A              B      the function edits.
      QTIMES 31                     Quad times 31.
⎕:                                  APL would like a value for ⎕.
      )SI                           Let's look at the state now.
⎕                                   A ⎕ line is currently pendent.
QTIMES[1]                           Preceded by pendent QTIMES.
PLUS[2]*                            And PLUS is still suspended.
⎕:                                  APL again requests ⎕ input.
      SIXPLUS ABLE                  Give it an erroneous line.
VALUE ERROR.                         ⍺⍳⎕∊  is undefined.
      SIXPLUS ABLE
              |
⎕:                                  APL says try again, please.
      )SI                           Errors in immediate or evalu-
⎕                                   ated input do not change the
QTIMES[1]                           state.
PLUS[2]*
⎕:                                  Input requested yet again.
      SIXPLUS 'ABC'                 This time cause an error down
DOMAIN ERROR.                       in the function PLUS.  Charac-
PLUS[1] C←A+B                       ters are not in the domain of
           |                        the + operator.
      )SIV                          Now the state is changed.
PLUS[2]*        C       A             B       Most recent suspen-
SIXPLUS[1]      Z       X                     sion of PLUS.
⎕
QTIMES[1]       B       A
PLUS[2]*        C       A             B       Previous one.
      A                             The names A and B refer to the
6                                   local variables of the most
      B                             recent invocation of PLUS.
ABC
      X                             X in SIXPLUS also has the
ABC                                 value 'ABC' .
```

Restarting Suspended Functions

Following the suspension of a function, the jump operator → is a legal constituent of a statement typed from the console. A jump can be made to any of the lines of the most recently suspended function. Of course, this is generally useful only after whatever caused the error condition has been remedied by the execution of other APL statements. The user must also be careful to jump to the correct line -- whether the line that triggered the error needs to be executed again depends on the particular function. Statement labels are defined and accessible, so they can be used to make jumps when convenient.

As always in APL, there is no way to jump to a line of other than the most recently invoked function.

```
      B←1 2 3                   Continuing the example begun
      B                         in the previous section, let's
1 2 3                           correct the bad value of B.
      X                         This doesn't change X, but X
ABC                             will not be referenced again.
      →1                        Jump back into PLUS at line 1.
217 248 279                     All invocations unwind.
      )SI                       The state indicator is now
PLUS[2]*                        back to the previous suspen-
      B                         sion of PLUS.  The previous
3 4 5                           local variables again become
      →1                        Restart this one too.
4 6                             1 2+3 4.
      )SI                       State indicator empty now.
```

Clearing the State Indicator

Often it is desired to merely discard a suspension in the state indicator rather than restart it. The jump operator → alone, with no operand, erases the state indicator back to the previous suspension point. Successive entries of → alone can be used to totally clear the state indicator.

```
      )SI                       Assume that we have this state
PLUS[2]*                        indicator, as from the example
SIXPLUS[1]                      (see "Suspended Statements").
⎕
QTIMES[1]
PLUS[2]*
      →                         Clear out the most recent
      )SI                       suspension.
PLUS[2]*                        Now we have only this one.
      →                         Another → clears the state
      )SI                       indicator entirely.
```

Restarting Following a Function Edit

A pendent function cannot be edited, because at least one of its lines is only partially evaluated and saved in the state indicator. There is no meaningful way to associate the saved information with an edited function definition when restarting. Consequently, if it is necessary to alter a pendent function, the state indicator must be cleared past it with the → operator, as discussed above.

Suspended functions, however, can be edited and then restarted again successfully. When a function is restarted after an edit, the following restrictions apply.

No local variables are created or destroyed when a function is restarted. If any change was made in the local variable names during the edit, variables with the new names will be created only at the next time the function is invoked. All previous invocations will have accessible to them only the local names that were created at their respective invocations. Any reference to a new name by a restarted function will be satisfied by the normal name referencing rules: it will be searched for in the calling function, in its caller, and so on, until ultimately it may access or create a global variable.

The values of the label variables in previous invocations are not altered, even if statements are moved around in the function definition. However, on any new invocation, the new label local variables will correctly correspond to the new function definition.

Other header line changes, such as changing a function from monadic to dyadic, or adding a returned value, for example, similarly affect only new invocations of the function.

The statements of the function, however, are defined immediately by the new definition, even in previously suspended invocations. For example, if, following a suspension of a function, an operator is added to line 2 of the function definition, and then a jump is made to line 2 of that function, then the new definition of line 2 is fetched and executed, and the added operator is performed.

The erasing of local variables on a function return is consistent in that precisely the ones that were created upon function invocation are erased upon termination, even if the local variable names were altered in an intervening function edit.

When there is any doubt as to the meaning of restarting an edited function, it is best to clear the state indicator and start fresh.

Panic

In certain exceptional circumstances, the APL interpreter may find itself so confused that it cannot proceed. This is usually due to an error in the interpreter itself or else a serious malfunction in the Multics operating system or hardware, rather a user's error. When this happens, APL prints the message

```
...APL PANIC...COMMAND LOOP CALLED...
```

and invokes a Multics listener beneath itself. The APL session in progress cannot be continued.

The recommended procedure following a panic is to notify the system programmers at your installation. Forward to them a stack trace, which can be obtained with the Multics "trace_stack" command, and your console sheet, with as much history as possible retained concerning what you were doing. You may continue to use APL by invoking it again, but there is no way to recover the state of your interrupted session.

SECTION IV

SYSTEM COMMANDS

SYSTEM COMMANDS GENERALLY

System commands are lines typed by the user to adjust or control the operation of the APL interpreter. They are distinguished from expressions to be interpreted by always beginning with a right parenthesis (no legal expression could ever begin that way). The right parenthesis is followed by the name of the particular system command, which is then followed by arguments separated by spaces. The arguments required by each command vary from one to the next and are discussed under the individual command descriptions below.

System commands can be typed whenever the APL processor is awaiting immediate input or evaluated input. They cannot be typed while editing a function, nor can they be placed in functions for execution when the function is called, nor can they be typed while the interpreter is listening for character input. A system command is performed as soon as it is typed, and then APL requests again the input it was awaiting before encountering the command (unless the action of the command itself was such as to cancel the need for that input).

WORKSPACE PARAMETERS

Each workspace contains eight data values used for specific, fixed purposes by the APL processor. These are the index origin, the page width, the number of significant digits, the fuzz, the random number seed, the string for immediate input, the string for evaluated input, and the string for character input. Each of these workspace parameters has associated with it a system command to set its value.

The workspace parameters are also affected by the)*CLEAR* and)*LOAD* system commands. The)*CLEAR* command resets all parameters to their default values, while the)*LOAD* command sets them to their values as recorded in the saved workspace.

In addition, each workspace parameter is also changeable by an APL function of the same name as the system command; the functions that accomplish this are called library functions. Their definitions can be copied with the)*COPY* system command from the public workspace >*APL*>*WSFNS* into any user workspace. Section III describes the use of library functions in more detail.

The)*ORIGIN* System Command

The)*ORIGIN* system command takes as its argument either the constant 0 or the constant 1. The command establishes its argument as the new value of the index origin and types out the old value. The index origin of a clear workspace is 1 by default. The index origin of a workspace is saved and restored by the)*SAVE* and)*LOAD* system commands.

The index origin determines whether numbers from 0 to *N*-1 or from 1 to *N* will be used to number coordinates and elements for various operators:

A[A;A;...;A]		interpretation of subscripts;
⍳S	V⍳A	result of index operators;
⍋A	⍒A	result of grade operators;
?A	S?S	result of roll and deal;
⍋[S]A	⍒[S]A	interpretation of coordinate numbers.
⌽[S]A	V⌽[S]A	
V/[S]A	V\[S]A	
⍟/[S]A		

Examples:

```
      )ORIGIN 1
 WAS 1
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ,. '
      A⍳'HOW ARE YOU?'
 8  15  23  29  1  18  5  29  25  15  21  30
      A[6 9 14 5 27 29 20 8 1 14 11 19 28]
 FINE, THANKS.
      ⍳4
 1  2  3  4
      LOTS←?10000⍴100
      (⌊/LOTS),(⌈/LOTS)
 1  100
      ⎕←M←?3 4⍴100

  80  88  24  82
 59   24   59   32
 84    1   74   24
      +/[1]M
223   113   157   138
      +/[2]M
274   174   183
      ⍋[1]M

 2   3   1   3
 1   2   2   2
 3   1   3   1
      )ORIGIN 0
WAS 1
      A⍳'HOW ARE YOU?'
7  14   22   28   0   17   4   28   24   14   20   29
      A[5  8  12  4  26  28  19  7  0  13  10  18  27]
FINE, THANKS.
      ⍳4
0   1   2   3
      LOTS←?10000⍴100
      (⌊/LOTS),(⌈/LOTS)
0   99
      +/[0]M
223   113   157   138
      +/[1]M
274   174   183
```

The)*WIDTH* System Command

The)*WIDTH* system command takes as its argument an integer constant from 30 to 130. The integer supplied is established as the new page width to be observed by the APL output routines, and the previous value of the page width is typed out. The page width of a clear workspace is 80 characters. The page width of a workspace is saved and restored by the)*SAVE* and)*LOAD* system commands.

The page width of a workspace determines the maximum number of characters which the APL output routines place on a line before deciding that the line is full and overflowing to the next line. Successive elements of a vector continue to be placed on one line until one of the elements no longer fits within the established page width. Then that element and all later ones are deferred to a following line or lines. The exact formatting of the output of values is discussed in Section III.

The page width of a workspace affects only the printing of system command responses and the printing of data values. It does not affect the printing of function definitions by the editor, nor does it affect the printing of error messages. Source lines printed by the editor and in error messages always appear in their actual length, regardless of the workspace page width. The page width of a workspace does not affect input, but only output; as discussed in Section II, any input line to the APL processor may be up to 256 characters in width at all times.

```
      A←⍳17
      B←3 4⍴○⍳12
      )WIDTH 80
WAS 80
      A
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
      B

  3.141592654   6.283185307   9.424777961  12.56637061
 15.70796327   18.84955592   21.99114858   25.13274123
 28.27433388   31.41592654   34.55751919   37.69911184
      )WIDTH 50
WAS 80
      A
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
      16  17
      B

  3.141592654   6.283185307   9.424777961
       12.56637061
 15.70796327   18.84955592   21.99114858
       25.13274123
 28.27433388   31.41592654   34.55751919
       37.69911184
      )WIDTH 30
WAS 50
      A
1  2  3  4  5  6  7  8  9  10
      11  12  13  14  15  16
      17
      B

  3.141592654   6.283185307
        9.424777961
       12.56637061
 15.70796327   18.84955592
       21.99114858
       25.13274123
 28.27433388   31.41592654
       34.55751919
       37.69911184
      'NOTE THAT THE WIDTH AFFECTS CHARACTER OUTPUT TOO.'
NOTE THAT THE WIDTH AFFECTS CH
      ARACTER OUTPUT TOO.
```

The *)DIGITS* System Command

The *)DIGITS* system command takes as its argument an integer constant from 1 to 19. The integer supplied is established as the new number of significant digits for the workspace, and the old value is typed out. The number of significant digits of a clear workspace is 10. The number of significant digits of a workspace is saved and restored by the *)SAVE* and *)LOAD* system commands.

The number of significant digits of a workspace determines only how values are formatted for printing. It does not affect the stored values themselves, nor does it affect their calculation. All values in Multics APL are calculated to 63 bits of precision (approximately 19 decimal digits). As a value is printed, it is rounded to the desired number of significant digits when it is converted to printable characters.

```
      A←9999.4×0.01*⍳4
      B←3 4⍴○⍳12
      C←6830 2.2E8 2.718281828 3.14159265358979
      )DIGITS 10
WAS 10
      A
99.994  0.99994  0.0099994  9.9994E¯5
      B[;1]
3.141592654  15.70796327  28.27433388
      C
6830  220000000  2.718281828  3.141592654
      )DIGITS 5
WAS 10
      A
99.994  0.9994  0.0099994  9.9994E¯5
      B[;3]
9.4248  21.991  34.558
      C
6830  2.2E8  2.7183  3.1416
      )DIGITS 4
WAS 5
      A
99.99  0.9999  0.009999  9.999E¯5
      B[3;]
28.27  31.42  34.56  37.7
      C
6830  2.2E8  2.718  3.142
      )DIGITS 3
WAS 4
      A
100  1  0.01  0.0001
      B[2;]
15.7  18.8  22  25.1
      C
```

```
 6.83E3  2.2E8  2.72  3.14
      )DIGITS 19
WAS 5
      3÷⍳13
3  1.5  1  0.75  0.5999999999999999999  0.5
      0.4285714285714285714  0.375  0.3333333333333333333
      0.3  0.2727272727272727273  0.25
      0.2307692307692307692
      )DIGITS 18
WAS 19
      3÷5
0.6
```

The)FUZZ System Command

The)FUZZ system command takes as its argument a constant greater than or equal to zero but less than one. The argument supplied is taken as the new workspace fuzz, and the old fuzz is typed out. The fuzz of a clear workspace is 1.0E¯13. The fuzz is saved and reloaded by the)SAVE and)LOAD system commands.

The workspace fuzz is used as an acceptable tolerance limit whenever APL is comparing two numbers for equality. Two numbers are considered equal if they are within fuzz of each other. The purpose of having a tolerance for such tests is to eliminate dependence of the decision on the last few digits of the numbers involved; the last few digits of results are not always correct due to the finite precision of computer arithmetic.

The fuzz is also used when a particular value is required to be an integer. Any number within fuzz of an integer is accepted. Examples of this are dimension numbers and subscript values.

Of course, setting the fuzz wide is not a cure-all for domain errors. In general, a comprehensive error analysis should precede any tampering with the fuzz.

```
      A←50×0.01*⍳9
      A
0.5  0.005  5E¯5  5E¯7  5E¯9  5E¯11  5E¯13  5E¯15  5E¯17
      A=0
0  0  0  0  0  0  0  1  1
      'XYZ'[0.999999999999999]
X
      'XYZ'[0.99]
DOMAIN ERROR.
      'XYZ'[0.99]
          |
      )FUZZ 0.000001
WAS 1E¯13
      A=0
0  0  0  1  1  1  1  1  1
      )FUZZ 0.1
WAS 1E¯6
      A=0
0  1  1  1  1  1  1  1  1
      'XYZ'[0.99]
X
      )FUZZ 0
WAS 0.1
      A=0
0  0  0  0  0  0  0  0  0
      'XYZ'[0.9999999999999999999]
DOMAIN ERROR.
      'XYZ'[0.9999999999999999999]
           |
```

The)*SETLINK* System Command

The)*SETLINK* system command takes as its argument an integer constant between 1 and 34359738367. Its argument is established as the new seed for the multiplicative congruential random number generator, and the old value of the seed is typed out. The initial seed of a clear workspace is derived from the real-time clock at the time the workspace is cleared. The random number seed is saved and restored by the)*SAVE* and)*LOAD* system commands.

In the random number generator used by Multics APL, the seed used to produce each random number is a function of the seed used to produce the previous one. The sequence of random numbers generated in a clear workspace cannot be predicted in advance, as the seed of a clear workspace depends upon the calendar clock. For applications requiring a reproducible sequence of random numbers, the seed can be initialized with the)*SETLINK* system command or the *SETLINK* library function. Since the seed is restored by the)*LOAD* system command, the random number sequences produced following repeated loads of the same saved workspace will also be identical.

```
      )CLEAR
CLEAR WS.
      )SETLINK 123456
WAS 18817044392
      ?10ρ100
94  86  40  46  15  62  57  75  11  58
      )CLEAR
```

```
 CLEAR WS.
      )SETLINK 123456
WAS 18882306764
      ?10ρ100
94  86  40  46  15  62  57  75  11  58
```

The)*SFII* System Command

The)*SFII* system command takes as its argument a character string of up to sixteen characters enclosed between quotes. The string may contain any of the APL graphics as well as control characters such as newlines and tabulates. The given string is established as the string to be typed by the APL interpreter whenever it is awaiting immediate input; i.e., a line to be interpreted. If the given string is null, the interpreter will not type anything as a go-ahead signal. The string for immediate input in a clear workspace is six blanks. The string for immediate input is saved and restored by the)*SAVE* and)*LOAD* commands.

If either of the Multics streams "user_input" or "user_output" is not attached to a terminal, then none of the go-ahead strings are typed, regardless of whether they are null.

```
      )SFII '==>'
==>ι4
1  2  3  4
==>)SFII 'TYPE:
'
TYPE:
ι4
1  2  3  4
TYPE:
)SFII ''
ι4
1  2  3  4
)SFII '      '
      ι4
1  2  3  4
```

The)*SFEI* System Command

The)*SFEI* system command is identical to the)*SFII* system command except that it concerns the string for evaluated input. The interpreter awaits evaluated input in response to the execution of the ⎕ operator in a line it is interpreting. In a clear workspace, the string for evaluated input is a quad, a colon, a newline, and six blanks.

```
      2×⎕
⎕:
      )SFEI '==>'
==>⍳4
2  4  6  8
      ⎕+1
==>)SFEI 'o'
o)SFII 'I: '
o⍳4
2  3  4  5
I: )SFEI '⎕:
      '
I: )SFII '         '
      12÷⎕
⎕:
      ⍳4
12  6  4  3
```

The)SFCI System Command

The)SFCI system command is identical to the)SFII or the)SFEI system commands except that it concerns the string for character input. The interpreter awaits character input in response to the execution of the ⍞ operator in a line it is interpreting. In a clear workspace, the string for character input is null; that is, nothing is typed before awaiting character input.

```
      ⌽⍞
)SFCI 'WOW.'
'.WOW' ICFS)
      )SFCI 'WOW.'
      ⌽⍞
WOW.ABLE BAKER
REKAB ELBA
      )SFCI '.
'
      'E'=⍞
.
THERE
0  0  1  0  1
      )SFCI ''
      5↑⍞
BRIEFLY, THE PURPOSE IS TO MAKE THIS STRING
BRIEF
```

NAME TABLE MANAGEMENT

The name table is an area of the workspace set aside for remembering names and the objects to which they refer. Names are used to refer to variables, functions, and groups. A number of system commands exist for inspecting and manipulating the name table.

A variable is simply an APL value that has been given a name. Recall that a value in APL is a rectangular array of numeric or character elements that can have any number of dimensions. Variable names can be either local or global; the distinction is explained in Section III, but briefly, a name is local when it is listed in the header line of a function. Local names exist only during the execution of the function to which they are local. All remaining variables are global. The names of all the global variables which exist in a workspace can be printed with the)*VARS* system command. All of the local variables currently in existence, together with the names of the functions to which they are local, can be listed with the)*SIV* system command.

A function is a stored APL program. The names of all the functions defined in a workspace can be listed with the)*FNS* system command. The names of the ones currently in execution may be listed with the)*SI* command.

A group is a list of names of other objects. Grouping the objects allows them to be copied and erased as a unit, without repetitively typing their individual names. Groups have no significance other than in the copy and erase commands. Groups are manipulated with the)*GROUP*,)*GRP*, and)*GRPS* system commands.

The)*ERASE* system command can be used to remove any object listed by the)*VARS*,)*FNS*, or)*GRPS* commands from the workspace. Erased objects are completely discarded, freeing the name and storage they occupied for other uses. Local variables cannot be erased by the)*ERASE* command, but they are erased automatically when the function to which they are local ceases execution.

The)*CLEAR* command is used to erase an entire workspace and everything in it. It is a way of getting a fresh start. The)*CLEAR* command not only erases all objects in a workspace, but it also clears the state indicator (the record of functions currently in execution) and resets the workspace parameters (such as page width and index origin) to their default values. The workspace created when APL is entered from Multics is a clear workspace.

The)*VARS* System Command

The)*VARS* system command is used to print out a list of the global variable names defined in a workspace. The list is produced in alphabetic order, and is printed in as many columns across the page as the workspace page width allows.

The)*VARS* system command can be issued without any arguments, in which case it lists all the global variables. Optionally, it can also be issued with a name as its argument, in which case it lists only names which match or follow the argument name in alphabetic order. Thus, if it is desired to list only a subset of the variable names, the starting name can be supplied as an argument to)*VARS* and the QUIT button can be pressed when enough names have been displayed. The name supplied as the argument need not exist in the workspace; it is used only in an alphabetic comparison to decide which names to print.

```
      )VARS
A       B       DELTA   DIVIDE  DOGWORK LL_PTR  MAT
PRIN    RATE    SYZYGY  THIS_IS_A_VERY_LONG_ONE TOKEN
X       Y
      )VARS D
DELTA   DIVIDE  DOGWORK LL_PTR  MAT     PRIN    RATE
SYZYGY  THIS_IS_A_VERY_LONG_ONE TOKEN   X       Y
      )VARS DOODLE
LL_PTR  MAT     PRIN    RATE    SYZYGY
THIS_IS_A_VERY_LONG_ONE TOKEN   X       Y
      )VARS ZINSMEISTER
      )CLEAR
CLEAR WS.
      )VARS
      ABLE←3
      )VARS
ABLE
```

The)*FNS* System Command

The)*FNS* system command is used to print a list of the names of the functions defined in a workspace. Like the)*VARS* command, the)*FNS* command prints its list in alphabetic order, and a name is accepted as an optional argument indicating where to begin the list.

One special feature of the)*FNS* system command relates to external functions. External functions are thoroughly discussed later in this section, but briefly, an external function is one whose definition is not supplied in the APL language itself, but in some other Multics language, such as PL/I. The names of such externally defined functions are tagged with an asterisk in the)*FNS* listing.

```
      )FNS
F        FOLLOW  JPZ      ORIGIN* PLOT    PLOTC   ZETA
      )FNS PLOTB
PLOTC    ZETA
      )MFN WIDTH
      )FNS PLOTB
PLOTC    WIDTH*  ZETA
```

The)*GRPS* System Command

The)*GRPS* system command is used to print a list of the names of the groups defined in a workspace. Like the)*VARS* and)*FNS* commands, the)*GRPS* command prints its list in alphabetic order, and accepts a starting name as an optional argument.

```
      )GRPS
GRAPHING_ROUTINES      STAT_PKG
      )GRPS S
STAT_PKG
```

The)*GRP* System Command

The)*GRP* system command lists the names of the members of a group. It takes as its argument the name of a group.

```
      )GRP GRAPHING_ROUTINES
PLOT     PLOTC
      )GRP STAT_PKG
DELTA    F       FOLLOW  JPZ     SYZYGY  ZETA
```

The)*GROUP* System Command

The)*GROUP* system command is used to gather objects into a group, to append more objects to a group, or to disband a group.

The first argument of the)*GROUP* command must be the name of the group upon which the command is to operate. If the purpose of the)*GROUP* command is to create the group, then the group need not yet exist at the time the command is issued; otherwise, it is an error if the group mentioned as the first argument does not exist.

If no further arguments beyond the group name are supplied, then the *)GROUP* command disbands the group. Disbanding a group has no effect upon its members; the only consequence is that they are no longer considered to be in a group. This should be carefully contrasted with erasing a group, which also erases its members. The *)ERASE* command is discussed in the next section.

If some additional names are typed following the group name argument, then the *)GROUP* command establishes a group of the designated name having the indicated members. Any name can be a member of a group: a variable name, a function name, the name of another group, or even a name which has never been used (a name which has no referent yet--if subsequently an object is created with that name, then the new object will be considered to be a member of the group).

There is only one name that cannot be made a member of a group: the group's own name itself. That is because the occurrence of the group's own name in the list of proposed members has a special meaning to the *)GROUP* command: it means that all the previous members of the group are to be retained in the new group along with the new members. This provides a way of appending new members to an existing group.

An old group of the same name as the newly established group is disbanded by the *)GROUP* command.

```
      )GROUP ONE A B C
      )GRPS
ONE
      )GRP ONE
A     B     C
```

Create a group named *ONE* . It is immaterial whether or not the objects *A* , *B* , or *C* exist.

```
      )GROUP TWO D E F ONE
      )GROUP TWO TWO G
      )GRP TWO
D     E     F     G
```

A group can be a member of a group. Append an object to group *TWO* .

```
      )GROUP ONE
      )GRPS
TWO
```

Disband group *ONE* .

```
      )GROUP ONE A B ONE C
COMMAND ERROR.
      )GROUP ONE A B ONE C
                     |
```

Error. One cannot append to a group that does not exist.

```
      A←1
      )GROUP A X Y Z
NAME DUP ERROR.
      )GROUP A X Y Z
             |
```

Another error. A name cannot by used to refer to more than one object at once; in this case, a variable as well as a group.

The)*ERASE* System Command

The)*ERASE* command is used to delete objects from a workspace. Its arguments are any number of names of objects to be deleted. The objects can be global variables, functions, or groups. When an object is erased, it is completely removed from the workspace and discarded. No record of its previous existence remains. Its name and the storage it occupied become available for other uses.

Local variables cannot be erased by the)*ERASE* command. However, they are automatically erased when the function to which they are local completes its execution.

When an argument to an)*ERASE* command is a group, in addition to the group being disbanded, the variables and functions that are members of that group are erased. Groups that are members of the mentioned group are disbanded, but their members in turn are not affected. Thus, erasure of groups containing groups is not fully recursive; only the members of the groups mentioned in the)*ERASE* command itself are erased.

```
      )VARS
 A      B        DELTA    DIVIDE  DOGWORK LL_PTR   MAT
 PRIN   RATE     SYZYGY   THIS_IS_A_VERY_LONG_ONE TOKEN
X       Y
      )FNS
F       FOLLOW   JPZ      ORIGIN* PLOT     PLOTC    ZETA
      )GRPS
GRAPHING_ROUTINES      STAT_PKG
      )GRP GRAPHING_ROUTINES
PLOT     PLOTC
      )GRP STAT_PKG
DELTA   F        FOLLOW  JPZ      SYZYGY  ZETA
      )ERASE A B X STAT_PKG
      )ERASE DIVIDE GRAPHING_ROUTINES ORIGIN SYZYGY LL_PTR
NOT ERASED:      SYZYGY         Because it was already gone
      )VARS                     from the previous erase.
DOGWORK MAT      PRIN     RATE     THIS_IS_A_VERY_LONG_ONE
TOKEN   Y
      )FNS                      No functions left.
      )GRPS                     No groups, either.
```

The)*CLEAR* System Command

The)*CLEAR* system command is used to obtain a fresh workspace. When the)*CLEAR* command is issued, the APL processor types *CLEAR WS*, erases all objects, discards the state indicator and all local variables; resets the index origin, page width, number of significant digits, fuzz, go-ahead strings, and workspace identification to their default values; and reads the calendar clock to obtain a new seed for the random number generator.

The *)CLEAR* command does not affect the input/output stream switching or the timers maintained by the I-beam system-dependent operator.

```
      )VARS
A     B         DELTA  DIVIDE  DOGWORK LL_PTR  MAT
PRIN  RATE      SYZYGY THIS_IS_A_VERY_LONG_ONE TOKEN
X     Y
      )FNS
F     FOLLOW    JPZ    ORIGIN* PLOT    PLOTC   ZETA
      )GRPS
GRAPHING_ROUTINES      STAT_PKG
      )CLEAR
CLEAR WS.
      )VARS
      )FNS
      )GRPS
```

The *)SI* System Command

The *)SI* system command is used to inspect the state indicator of the APL processor. The state indicator is an area of the workspace set aside to record the state of functions currently invoked.

The meaning and workings of the state indicator are fully explained in Section III, but, briefly, the state indicator acts as a stack. As one APL statement invokes another (either as a function or as evaluated input), the information pertaining to the partially evaluated invoking line is stacked in the state indicator. The APL processor is then free to evaluate the invoked lines, knowing that when it finishes it can return to complete the evaluation of the invoking line by restoring the saved state of evaluation from the state indicator stack. Since the invoked statements can further invoke other statements, many partial evaluations may need to stack successively in the state indicator. As their respective evaluations complete, the stack is popped back in parallel.

A statement whose execution is stopped temporarily because the processor must execute another statement (which it invokes) is said to be pendent. Thus, the use of the state indicator discussed so far is to remember all pendent statements.

Another item of information remembered in the state indicator is function suspensions. When the execution of a function produces an error report, statements cease to be drawn from the function definition and are instead read from the user's terminal until the function is explicitly restarted. During this interval, the function is said to be suspended. An entry in the state indicator for a suspension differs from that for a pendent statement in that no partially evaluated statement is remembered, and also in that a suspension marks a place where the user obtained control and was able to type new statements.

In the listing printed by the)*SI* command there is one line per entry on the state indicator stack. The stack is printed in the order of most recent item first to least recent last; thus, the first line printed corresponds to the most recent entry made into the state indicator. Each line shows the name of the function in execution (or the symbol ⎕ if the entry refers to an evaluated input line), the statement number upon which execution will resume (the pendent statement itself for pendent entries; the statement following the error for suspended entries), and finally an asterisk if the entry represents a suspension (lines without an asterisk correspond to pendent entries).

```
      ∇PLUS                     Create some functions for the
FUNCTION 'PLUS' NOT FOUND.      purpose of illustrating the
INPUT.                          )SI and )SIV commands.
C←A PLUS B                      This one mimics the built-in
C←A+B∇                          dyadic + operator.
      ∇SIXPLUS
FUNCTION 'SIXPLUS' NOT FOUND.
INPUT.                          This one uses the above to
Z←SIXPLUS X                     add 6 to its operand.
Z←6 PLUS X∇
      ∇QTIMES
FUNCTION 'QTIMES' NOT FOUND.
INPUT.                          And this one does an evaluated
B←QTIMES A                      input.
B←⎕×A∇
      )SI                       Initially, the state is empty.
      1 2 PLUS 3 4 5            Now cause an error, hence a
LENGTH ERROR.                   suspension of the PLUS func-
PLUS[2] C←A+B                   tion.
           |
      )SI                       Now the state has one entry, a
PLUS[2]*                        suspension.
      QTIMES 31                 Call a function which requests
⎕:                              evaluated input.
      )SI                       Now look at the state.
⎕                               Evaluated input is pendent.
QTIMES[1]                       Line 1 of QTIMES is pendent.
PLUS[2]*                        PLUS is still suspended.
```

```
⎕:                                   Input requested again.
      SIXPLUS 'ABC'                  Cause a further error.
DOMAIN ERROR.
PLUS[1] C←A+B
           |
      )SI                            Now the state has another sus-
PLUS[2]*                             pension of  PLUS .
SIXPLUS[1]                           SIXPLUS  is pendent on  PLUS .
⎕                                    Evaluated input is pendent on
QTIMES[1]                            SIXPLUS , and  QTIMES  on ⎕.
PLUS[2]*                             PLUS  still suspended here too.
      →                              Clear out the most recent sus-
      )SI                            pension.
PLUS[2]*                             Now we are back to the pre-
      →                              vious one.  Clear it out too.
      )SI                            The state is empty again.
```

The)SIV System Command

The)SIV system command performs the same function as the)SI system command, except that each line of the display shows, in addition to everything shown for the)SI command, also the names of all variables local to the particular invocation.

Since a reference to a variable name is satisfied by the most recently created local variable of that name, the referent of any given name is easily found by scanning the)SIV list downward. The first instance of the sought name is the satisfying referent. If the name is not found anywhere in the)SIV list, then the reference will be satisfied by a global variable. Local and global name referencing is treated more fully in Section III.

```
      1 2 PLUS 3 4 5                 The same example now done with
LENGTH ERROR.                        )SIV in place of )SI.
PLUS[1] C←A+B
           |
      )SIV
PLUS[2]*         C         A         B
      QTIMES 31
⎕:
      )SIV
⎕
QTIMES[1]        B         A
PLUS[2]*         C         A         B
L:
      SIXPLUS 'ABC'
DOMAIN ERROR.
PLUS[1] C←A+B
           |
      )SIV
PLUS[2]*          C         A         B
SIXPLUS[1]        Z         X
⎕
QTIMES[1]         B         A
PLUS[2]*          C         A         B
      →
      →
      )SIV
```

SAVING AND RELOADING WORKSPACES

One of the most important features of the APL language is the ability to save the complete contents of a workspace and then take it up again later. Work can then be continued as if there had been no interruption. On Multics, workspaces are saved as segments (or, if necessary, as multisegment files) anywhere in the storage system hierarchy. Normal Multics quota and access conventions govern the storage of saved workspaces.

When a workspace is saved, everything necessary to resume the session in progress is remembered. The values of all variables, both local and global, the definitions of all functions and groups, everything in the state indicator, and the settings of all the workspace parameters are saved. When a workspace is to be taken up again, the user has a choice of how much of the saved workspace to recall. He can copy individual variables or functions or groups (the *)COPY* system command with a specific object named); or he can copy all objects but not the state of execution (the *)COPY* command with no object named); or he can recall the entire workspace (the *)LOAD* command). Objects can be moved from workspace to workspace or duplicated in several workspaces by loading one workspace, copying any desired objects from another, and saving the newly updated workspace again.

Workspace Identification

The current, or active, workspace has associated with it a workspace identification, which is the absolute or relative pathname of the saved workspace that was most recently loaded. Internally, all workspace identifications end in the four-character suffix ".apl" to clearly distinguish saved workspaces in the Multics hierarchy. However, the user never need type the suffix, as it is appended automatically by the APL processor to all identifications typed in.

Until any saved workspace is loaded in an APL session, the current workspace has the identification *CLEAR WS*, indicating that it was never loaded from any saved copy. The identification *CLEAR WS* does not actually mean that the workspace is clear, as many objects may have been constructed in it since it was created.

When a)*LOAD* command is issued, the APL processor replaces the current workspace identification with the pathname typed in the load command (after appending the suffix ".apl" to it), locates the saved workspace in the Multics hierarchy, and then loads its contents into the current workspace.

When a)*SAVE* command is issued with no pathname as argument, the active workspace is stored according to its current identification; that is, from where it was loaded. Alternatively, if a)*SAVE* command is issued with some pathname as argument, then the workspace identification is replaced by the new pathname, and the workspace is saved at the new place in the Multics hierarchy. It is an error to issue a)*SAVE* command with no pathname given when the current workspace identification is *CLEAR WS*.

The workspace identification can also be inspected or changed at any time with the)*WSID* system command. No commands other than)*CLEAR*,)*SAVE*,)*LOAD*, and)*WSID* affect or concern themselves with the workspace identification.

Note that no workspace identification typed into APL ever shows the ".apl" appendage; however, the actual saved workspaces in the Multics hierarchy always have the suffix. Also note that the current workspace identification is always set apart from the pathname as typed in the)*SAVE*,)*LOAD* or)*WSID* system command (with the ".apl" suffix added), so several different identifications often can access the same saved workspace (relative vs. absolute pathnames, or multiple names on a segment or containing directory). The user is cautioned that changing the working directory when the workspace identification is only a relative pathname can change the meaning of that pathname--and a succeeding)*SAVE* command with no argument may not necessarily refer to the previously loaded workspace.

Passwords

It is possible to associate a password with a saved workspace. When a workspace has been saved with a password, APL will prevent a load or copy operation from it unless the password can be supplied. No password is required to delete a saved workspace. Cautious users should note, however, that nothing prevents another user from supplying a program of his own construction to search for interesting things in your saved workspaces.

The mechanics of supplying passwords are as follows. Passwords are accepted on the *)SAVE*, *)LOAD*, *)COPY*, and *)PCOPY* system commands. To indicate that a password is to be supplied, a colon is typed following the last character of the pathname constituting the workspace identification in the command line (blanks are optional between the workspace identification and the colon). The remainder of the command line is typed normally. APL responds *PASSWORD*: and when the password, which is zero to eight characters of the user's choosing, is received, proceeds to process the command.

If a saved workspace with a password is accessed by a *)LOAD*, *)COPY*, or *)PCOPY* command without a password or with an incorrect password, the load or copy does not occur.

If a *)SAVE* command is issued without a colon, the workspace is saved with its current password. To remove or change the password of a workspace, the *)SAVE* command must be issued with a colon. A password of zero characters (simply a newline) is considered the same as no password. A clear workspace has no password.

The *)SAVE* System Command

The *)SAVE* system command is used to save a copy of the current APL workspace so that it can be taken up again later. Everything in the workspace is saved. Workspaces become Multics segments or multisegment files.

The *)SAVE* command can be issued in three forms: the command alone,

```
          )SAVE
```

or with a workspace identification (i.e., pathname),

```
          )SAVE  identification
```

or with a workspace identification and a colon,

```
          )SAVE  identification  :
```

In the first form, the command alone, the workspace is saved under its current identification and with its current password. See the preceding two sections for a description of workspace identifications and passwords. This form of the *)SAVE* command is illegal when the workspace identification is set to *CLEAR WS* i.e., when the current workspace was not loaded from any saved copy. The response to this command is the current date and time, followed by the workspace identification as a reminder.

In the second form, the current workspace is saved under the new identification (i.e., at a new place in the storage system hierarchy), but with the current password if any. The current workspace identification is replaced by the new identification typed in the *)SAVE* comamnd. The response to this form of the command is the current date and time.

In the final form of the *)SAVE* command, the APL processor asks the user to type a password. Then the current workspace is saved under the new identification with the new password. A password consisting of a newline alone is the same as no password. The current workspace identification and password are both changed to those of the *)SAVE* command. Again, the response to this command is the date and time.

The *)SAVE* command does not alter the current workspace in any way, except that its identification and password may change. Following the *)SAVE*, computations can resume in the current workspace.

If there are any Multics errors encountered in performing the save, such as record quota or access violations, they are reported and the current workspace remains unchanged. In such cases, the partially-created workspace segment in the Multics hierarchy is almost certainly bad, but the current workspace is still available to save again once the problem has been rectified. The partially-created saved workspace will also be bad if the *)SAVE* command is QUIT out of and not restarted. The mechanism by which a previously saved workspace is resaved is that the previous copy is truncated and the new one is written in its place.

```
      )CLEAR                    Set up a workspace to save.
CLEAR WS.
      I←3                       Put some things in it.
      A←'SUPERCALIFRAGILISTICEXPIALIDOCIOUS'
      P←ı1
      )SAVE <ALGORITHMS         A save command with a relative
11/15/72  2211.2                pathname.
      )SAVE >UDD>M>MGS>NEW      Save same workspace elsewhere,
11/15/72  2212.3                using an absolute pathname.
      )SAVE                     Save using the preceding path.
11/15/72  2213.1 >UDD>M>MGS>NEW.APL     APL reminds us of
```

the current workspace identification since we have not typed it.

The)*LOAD* System Command

The)*LOAD* system command takes as its argument a workspace identification, (i.e., pathname), optionally followed by a colon. If the colon is supplied, APL requests a password. If the password supplied does not match that of the saved workspace whose identification is given in the)*LOAD* command, then the error is reported and the current workspace remains unchanged. Otherwise, the current workspace is discarded and replaced by a copy of the saved workspace. The response to the)*LOAD* system command is *SAVED* followed by the date and time that the workspace was saved. The saved workspace itself is not altered by the)*LOAD* command.

The input/output stream attachments and the timers maintained by the I-beam operator are the only items not affected by the)*LOAD* command. All other features of the current workspace become identical to those following the)*SAVE* command which saved the copy, including local and global variable names and values, functions, groups, state indicator, and workspace parameters. Even the random number generator seed is loaded from the saved copy, so that the sequences of random numbers generated from reloaded workspaces are reproducible.

```
      )LOAD >UDD>M>MGS>NEW     Load a saved workspace.
 SAVED 11/15/72  2213.1        APL tells when it was saved.
      )VARS                    What's in it?
 A      I       P              These variables.
      I
 3
      )FNS                     No functions.
```

The)*COPY* System Command

The)*COPY* system command copies functions, groups, and global variables from a saved workspace into the current workspace. The current workspace remains unchanged except for the addition of the new object or objects.

The *)COPY* system command can be issued in two forms. The first is

)COPY identification

where "identification" is the pathname of a saved workspace. If the workspace was saved with a password, then a colon must be typed following its identification in the *)COPY* command so that APL will request a matching password. The meaning of this form of the command is to place copies of all functions, groups, and global variables contained in the saved workspace into the current workspace. If any object of the same name as an object to be copied already exists in the current workspace, it is erased and replaced by the copy. All other objects of the current workspace remain unchanged, as do the workspace parameters and the state indicator.

In the second form of the *)COPY* command, a list of specific objects is mentioned, as

)COPY identification object object...

where, as before, "identification" is the pathname of a saved workspace, possibly ending in a colon if a password is required, and each "object" is the name of a function, group, or global variable in the saved workspace. The function of this form of the *)COPY* command is to copy only the objects mentioned from the saved workspace into the current workspace. If any object is a group, however, all of its members are copied as well. As in the other form of *)COPY*, naming conflicts between copied objects and existing objects are resolved by erasing the existing objects. If a specified object does not exist in the saved workspace, an error report tells that it was not copied.

In any case, there is no change to the saved workspace. The response to the *)COPY* command is the date and time the donor workspace was saved.

```
      )COPY >APL>WSFNS ORIGIN   Copy the library function
SAVED 10/12/72  1202.0         ORIGIN  out of the public work-
      )FNS                     space.  Now it is defined in
ORIGIN*                        our workspace.
      )COPY >APL>WSFNS         Copy all objects out of WSFNS.
SAVED 10/12/72  1202.0
      )VARS                    Variables unchanged, as there
A       I       P              were none in WSFNS.
      )FNS                     But there were many functions.
DELAY*  DIGITS* E*      FUZZ*   ORIGIN* SETLINK*
SFCI*   SFEI*   SFII*   WIDTH*
                               The asterisks in the )FNS list
                               mean these are all external
```

functions, as discussed in
)*PCOPY* below.

The)*PCOPY* System Command

The)*PCOPY* (protected copy) system command behaves exactly like the)*COPY* system command with the exception of its treatment of naming conflicts. With the protected copy command, when an object to be copied has the same name as an existing object, the existing object remains unchanged, and the saved object is simply not copied. The names of any objects not copied are reported.

Like the)*COPY* command, the)*PCOPY* command can copy either specified objects or all global objects out of a saved workspace. The normal response to the command is the date and time the workspace was saved. There is no change made to the saved workspace.

```
      )CLEAR                          Create a workspace with some
 CLEAR WS.                            names used.
      E←*1
      DIGITS←1 2 3 4 5 6 7 8 9
      )PCOPY >APL>WSFNS               Now try a protected copy.
 NOT COPIED: DIGITS  E                These two objects could not be
 SAVED 10/12/72  1202.0               copied because the names were
      E                               in use.  The names retain
 2.718281828                          their former meanings.
      )COPY >APL>WSFNS                In contrast, an unprotected
 SAVED 10/12/72  1202.0               copy overrides the previous
      E 'IOA_ HELLO.'                 usages of duplicate names.
 HELLO.
```

The)*CONTINUE* System Command

The)*CONTINUE* system command provides no new capability to the APL system, but is simply a convenient way to terminate an APL session that must be resumed again later. The)*CONTINUE* command behaves identically to the sequence

```
                    )SAVE  CONTINUE
                    )OFF
```

of commands. That is, the current workspace is saved under the name *CONTINUE*, and the APL session is terminated. Later, the command

```
                )LOAD  CONTINUE
```

can be used to pick up the work again as if there had been no interruption.

```
       )CONTINUE                     Exit from APL with the con-
 11/15/72  2217.8 CONTINUE           tinue command.
 r 2218  13.832  48+503

 list continue.** -dtm               There is now a saved workspace
                                     named CONTINUE.
 Segments = 1,  Records = 2.

 11/15/72  2217.8  rewa  2  continue.apl

 r 2219  2.173  2+25

 apl                                 Enter APL.
       )LOAD CONTINUE                Resume interrupted session.
 SAVED 11/15/72  2217.8
```

The)*WSID* System Command

The)*WSID* system command is used to inspect or change the current workspace identification. If the)*WSID* command is typed with no arguments, as

```
          )WSID
```

then the current workspace identification is printed out. As explained earlier in this section, the workspace identification is the pathname of the saved workspace from which the current workspace was loaded. If the current workspace has not been loaded from any saved copy, its identification is *CLEAR WS*.

The)*WSID* command can also be used to set the current workspace identification. In this case, the user types

```
                    )WSID  identification
```

where "identification" is any absolute or relative pathname. APL appends the suffix ".apl" to the pathname and places it as the current workspace identification. The former identification is typed in reply.

The only purpose of setting the workspace identification is to allow a later)*SAVE* command given without an identification to save the workspace in the desired place. Beyond)*WSID*, the only commands that affect or concern themselves with the workspace identification are)*CLEAR*,)*SAVE*, and)*LOAD*.

```
      )WSID                     Observe the current workspace
CONTINUE.APL                    identification.
      )CLEAR
CLEAR WS.
      )WSID                     Observe the identification of
CLEAR WS                        a fresh workspace.
      )WSID >UDD>M>MGS>NEW      Reset the identification.
WAS CLEAR WS
      )SAVE                     Do a save under the new name.
11/15/72  2222.2 >UDD>M>MGS>NEW
```

The)*LIB* System Command

The)*LIB* system command is provided as a convenience to APL/360 users who are not yet accustomed to Multics. On APL/360, the)*LIB* command is used to list the names of saved workspaces within any APL library. On Multics, the)*LIB* command is implemented as a call on the Multics "list" command with an argument of "*.apl". This has the effect of listing all segments in the current working directory whose names have a second component of "apl". Of course, the Multics "list" command is far more general than this usage. Experienced Multics users will probably prefer to issue their own calls directly to "list" via the)*E* system command, as discussed later in this section.

```
      )LIB                      Issue the )LIB command and
                                change the type-ball for a
Segments = 2, Records = 3.      neater list.

rewa   2  continue.apl
rewa   1  new.apl

      )WSID                     Restore the APL type-ball.
>UDD>M>MGS>NEW.APL
```

The)*DROP* System Command

The)*DROP* system command is used to delete the saved copy of a workspace. The form of the command is

)*DROP* identification

where "identification" is the pathname (less the ".apl" suffix) of the saved workspace to be deleted. A password is not required to delete a saved workspace. The)*DROP* system command is essentially equivalent to the Multics "delete" command.

The *)DROP* command has no effect on the current workspace.

```
      )DROP CONTINUE
      )LIB

 Segments = 1, Records = 1.

 rewa    1  new.apl

      )DROP NEW
      )LIB
 ls: *.apl not found.
```

COMMUNICATING WITH MULTICS

In addition to the commands relating to the saving and reloading of workspaces, a number of other commands involve communication between APL and Multics. The *)Q*, *)QUIT*, and *)OFF* system commands are used to exit from APL. The *)PORTS* system command prints the names of other users currently logged in to Multics. Finally, the *)E* system command provides a means of executing Multics commands without exiting from APL.

The *)Q*, *)QUIT*, and *)OFF* System Commands

The *)Q*, *)QUIT*, and *)OFF* system commands are all identical. They cause the APL processor to return to its caller. If APL was invoked as a Multics command, this amounts to a return to Multics command level.

Before returning, APL deletes all segments of the current workspace so that it no longer consumes space in the user's process directory (recall that during the execution of APL the current workspace is maintained in a number of segments in the process directory having the names "apl.?", where "?" is a 15-character unique identifier). In addition, the former settings of the ring-zero teletypewriter device interface module are restored, so that character input and output over the user terminal again obey Multics conventions instead of the APL conventions described in Section II.

When APL is running, it maintains a cleanup handler so that the above actions occur even if an invocation of APL is released from the stack rather than commanded to return.

Following a return to Multics, the current workspace is no longer accessible. If the user wishes to save the results of an APL session, a *)SAVE* command must be issued before returning, or else the *)CONTINUE* command should be used to exit instead of *)Q*, *)QUIT*, or *)OFF*.

```
      )OFF                           That's all, folks.
 r 2225  7.103  16+122               Multics ready message.

 apl                                 Re-enter APL.
      )WSID                          No history left of the previous
 CLEAR WS                            workspace.
```

The *)PORTS* System Command

The *)PORTS* system command prints a list of the Multics users currently logged in. It is implemented as a call on the Multics "who" command. Any arguments typed after the *)PORTS* command are simply passed on to the "who" command, so a certain amount of selectivity is possible. Refer to the description of the "who" command in the Multics Programmer's Manual -- Commands and Active Functions, Order No. AG92, for further information.

```
      )PORTS .MULTICS                Issue the )PORTS command and
 Chang.Multics                       change type-ball for neater
 Snyder.Multics                      listing.
 MSmith.Multics
 DSLevin.Multics*
 Morris.Multics
```

The *)E* System Command

The *)E* system command is used to execute an arbitrary Multics command line from within APL. The entire remainder of the command line following the *)E* is passed unchanged to "cu_$cp", the Multics command processor, for execution.

During the time that control is passed out of APL, the special APL character set processing discussed in Section II is turned off. The former settings of the ring-zero teletypewriter device interface module are restored during this interval. Thus, the executed commands have available to them the normal Multics terminal input/output environment. When APL control is returned, it establishes its own character set processing again.

The user is cautioned, however, that the command line itself has been read by APL; hence, it has undergone the APL rather than the Multics input processing. While the APL and the Multics character sets largely overlap, there are some differences. It is up to the user to anticipate the translations mentioned in Section II and compensate for them where necessary. For example, if one types

```
        )E SM MGS M W̲HY ERROR WHEN ¨COPY_¨ING FILE FROB?
```

the actual message transmitted, in ASCII, will be

```
        Why error when ¢223py_¢222g file frob?
```

Corresponding to the *)E* system command there is an *E* library function. The library function is monadic; it accepts a character vector as its operand and returns no result. The character vector is passed to the Multics command processor for execution. Additional details on library functions can be found in Section III.

```
      )E IOA_ HELLO.
HELLO.
      )E APL
      I←53
      J←'JABBERWOCKY'
      )SAVE IT
11/15/72  2225.3
      )Q
      I
VALUE ERROR.
      I
      ∧
      )LOAD IT
SAVED 11/15/72  2225.3
      I
53
```

EXTERNAL FUNCTIONS

The Multics APL interpreter permits APL programs to make external calls out to object segments which have been created by other Multics translators, such as PL/I, provided that those object segments obey a specified interface. To the APL program, such a call looks like an ordinary reference to a defined function; the function may accept zero, one, or two arguments, and it may optionally return a result.

)DFN, *)MFN*, *)ZFN* Define External Function Names

The system commands *)DFN*, *)MFN*, and *)ZFN* are used to define external function names. The command *)DFN* is used to declare a dyadic function; i.e., one accepting two arguments. The command *)MFN* is used to declare a monadic function; i.e., one accepting one argument. And the command *)ZFN* is used to declare a zero-adic function; i.e., one accepting no arguments. Whether an external function produces a result need not be specified at the time its name is defined; in fact, the same function can at times return and at other times not return a value, as it chooses.

Definition Syntax

The syntax of an external name definition is

```
          )DFN aplname pathname
```

or else

```
          )DFN pathname
```

where *)DFN* can be replaced by *)MFN* or *)ZFN* as appropriate to the function being defined. The first form defines the name "aplname" to be an externally-coded dyadic function. When an APL program makes a function reference to "aplname", the APL interpreter performs a call on the object segment "pathname" with the calling sequence described below. When "pathname" returns, any returned value is considered as the result of "aplname", and execution of the APL program resumes.

The "pathname" may be an absolute or relative pathname, or it may be a reference name, in which case the Multics search rules will be used to obtain its referent. The "pathname" may contain both a segment name and an entry point name separated by a dollar sign, as "a$b", or it may simply contain a segment name, as "a", which will be considered "a$a"; i.e., a call to entry point "a" in segment "a". Note that the dollar sign must be typed as an iota ι on Selectric-type terminals, with the APL typing element mounted (see Section II for translations performed by the APL terminal input processor).

In the second form of definition, where "aplname" is not specified, it is considered to be the same as the entry point name of "pathname", (for example, "b" if pathname were "a$b", or "xyz" if pathname were simply "xyz").

Definition Errors

A definition error report can be due to: (1) an illegal character in the function name; (2) a global variable or group already in existence with the proposed function name; or (3) an illegally constructed pathname.

External Functions Cannot be Edited

External functions cannot be edited in any way by the APL editor. An attempt to open one for editing results in a definition error report. However, external functions can be erased or redefined, and their definitions can be copied from one workspace to another.

External Functions Tagged with "*" in *FNS* Listing

External function names are listed along with internal function names in a)*FNS* listing, but the external names have an asterisk appended to them to identify them as external.

External Function Calling Sequence

A procedure, say "f", which is to be called by APL as an external function, must conform to the following calling sequence:

```
     f:   proc(leftp,rightp,infop,valuep,Alloc,Error,Static);

     dcl  (leftp,rightp,infop,valuep) ptr,
          Alloc entry(fixed,fixed,fixed),
          Error entry(char(15) var),
          1 Static aligned,
          2 Reserved(5) fixed,
          2 (Digits,Width,Iorg,Niorg,Seed) fixed,
          2 Fuzz float bin(63);
```

Pointers "leftp" and "rightp" are input parameters corresponding to the left and right argument values in the reference to "f". If either argument is absent (as, "f" is monadic or zero-adic), the corresponding pointer will be null. Otherwise, the pointer points to the value bead for the argument supplied. The value bead structure is declared as:

```
dcl  1 vb aligned based(leftp),
       2 (v_reserved, v_type, v_number, v_rhorho) fixed,
       2 v_rho(vb.v_rhorho) fixed,
       2 v_value(vb.v_number)
                bit(vsize(vb.v_type))unaligned;
dcl  vsize(4) fixed bin(7) int static init(1,9,36,72);
```

In this structure, "v_reserved" is reserved for use of the APL storage manager; "v_type" is the internal type code of the value: 1 for bit, 2 for character, 3 for single-precision integer, 4 for double-precision floating-point; "v_number" is the number of elements in the value; i.e., the times reduction of the rho of the value; "v_rhorho" is the rank or rhorho of the value; "v_rho" is the shape or rho of the value; and finally, "v_value" is an array of bit strings, each of which represents one element of the value: a 1-bit bit, a 9-bit character, a 36-bit integer, or a 72-bit double-precision floating-point number. Note that this structure is adjustable and also self-referencing. Note also that it is aligned only to a word boundary, not too doubleword. If a "v_value" is declared as floating-point because it is known to be of type 4, then it must also have the "unaligned" attribute.

External Function Returned Value

Pointers "infop" and "valuep" relate to a possible value to be returned by "f". They are initialized to null by APL before calling "f". If "f" desires to return a value, it should make "infop" point to a properly filled-in information structure declared as:

```
dcl  1 info aligned based(infop),
       2 (i_type, i_number, i_rhorho) fixed,
       2 i_rho(info.i_rhorho) fixed;
```

and "valuep" point to the array of element values themselves. The information in the structure must obey the same conventions as the corresponding items of the value bead, as discussed above. When "f" returns to APL, the APL storage manager will allocate room in the workspace of sufficient size to contain the returned value and to copy all the elements into it.

Use of Supplied "Alloc" Entry

For convenience and efficiency, the external function may desire to directly allocate space for the returned result in the workspace itself, thus avoiding an unnecessary second allocation and copy. For this purpose, the entry point "Alloc" to the APL storage management package is provided. If "f" makes the call

```
call Alloc(type, number, rhorho);
```

then "Alloc" will create a value bead of sufficient size to contain an array of "number" elements of type "type" and having a rhorho of "rhorho". It will set "infop" to point to the information area of the value bead, fill in "v_type", "v_number", and "v_rhorho" (and also "v_rho" if rhorho equals 1), and set "valuep" to point to the value area of the value bead. The external function "f" is then responsible for filling in "v_rho" (if rhorho is greater than 1) and the element values themselves. The user is cautioned that the value bead is not doubleword-aligned in general.

Use of Supplied "Error" Entry

Entry point "Error" is provided to the external function so that it can generate a standard APL error message if it so desires. The call

```
call Error("string");
```

where "string" is up to 15 characters long, produces the message "string ERROR" followed by a reproduction of the source line calling the external function with the error marker under the external function name. "Error" does not return to its caller; instead it exits with a nonlocal go-to back to the main loop of the interpreter where the user's next console line will be read.

The string "string" must be expressed in APL internal character codes; see Section II.

Use of Supplied "Static" Structure

The structure named "Static" makes available to the external function the key parameters of the active workspace. If any of them are changed by the function, they should obey the domain limits specified for them; no validity checking will be performed on them when the external function returns. The domain limits are:

$$1 \le \text{Digits} \le 19$$
$$30 \le \text{Width} \le 130$$
$$0 \le \text{Iorg} \le 1$$

$$1 \leq \text{Seed} \leq 34359738367$$
$$0.0e0 \leq \text{Fuzz} < 1.0e0$$

Niorg ("not Iorg") must always be equal to 1-Iorg.

SECTION V

THE APL FUNCTION EDITOR

INVOKING THE FUNCTION EDITOR

Creation, inspection, and alteration of APL function definitions are performed by invoking the APL function editor. The function editor can be invoked whenever the APL processor is expecting immediate input or evaluated input. When the function edit is complete and the user exits from the editor, the system again requests the immediate or evaluated input it was previously awaiting.

The Multics APL function editor is different from the APL/360 editor; APL/360 users are cautioned to read all the material in this section. The Multics APL function editor is modelled after (and is functionally equivalent to) the Multics context editor "edm".

The editor is invoked by typing the editor-call character ∇ followed by the name of the function to be edited. For example, one types

∇*NAME*

to begin editing the function named *NAME*.

One possible response to an attempted editor invocation is the *DEFN ERROR* error report; this occurs if the function specified cannot be edited. The possible causes of a definition error when invoking the editor are discussed under "Errors Invoking the Editor" in this section. When a definition error occurs the editor is not invoked, and the APL processor requests again the input it was awaiting before the editor invocation was attempted.

If the editor invocation succeeds, then there are two cases. First, if the function mentioned in the invocation already exists, the editor responds *EDIT* and awaits editing requests. Editing requests are discussed under "Basic Editing Requests" in this section. Second, if the function mentioned does not already exist, the editor responds *FUNCTION NOT FOUND, INPUT* and then awaits the function to be input. Input mode is discussed in this section under "Input Request".

When the modifications to the function definition are complete and the user desires to exit the editor, he may do so either by issuing the *Q* editing request while in edit mode, or by typing a closing editor-call character ∇ at the end of any line, whether in input mode or edit mode. The termination of an edit is discussed in more detail under "Leaving the Editor" in this section.

The invocation of the editor to create a new function in Multics APL is different from that in APL/360. In APL/360, the entire function header line must be typed in the editor invocation. In Multics APL, only the function name is typed.

ERRORS INVOKING THE EDITOR

A definition error report *DEFN ERROR* occurs if the attempted invocation of the editor is unsuccessful. This can be due to one of four causes.

First, a definition error occurs if the proposed function name is not a legal APL name. The rules for governing which characters can be used to make names in APL can be found in Section III.

Second, an error occurs if the proposed function name is already being used as a group name or a global variable name. A name can refer to only one object at a time. The names of obsolete objects can be freed for other uses by erasing those objects with the)*ERASE* system command.

Third, an error occurs if an attempt is made to edit an external function. An external function is one whose definition is provided in some Multics language other than APL. The APL function editor cannot be used in any way with external function definitions. External function names are identified with an asterisk in a)*FNS* listing. Additional information on external functions can be found in Section IV.

Fourth and last, an error occurs if an attempt is made to edit a pendent function. Suspended functions and inactive functions can be edited, but not pendent functions. Generally speaking, a function becomes pendent when the execution of some line of it is begun, but cannot be completed before executing some other function (because the line contains a function call) or before interpreting a line read from the user's terminal (because the line contains an evaluated input request). In these cases, completion of the execution of the function line is pendent upon receiving another result, so the interpreter remembers the state of evaluation of the function line to the point of the new function call or input request. The interpreter then temporarily abandons it in order to interpret the new function or input line. If the user attempts to edit the given function before its pendent line has been allowed to complete execution, he receives a *DEFN ERROR* message. The names of functions pendent can be listed with the *)SI* system command. Pendent functions can be made available for editing again by clearing the state indicator with the → operator.

THE FUNCTION HEADER LINE

The first line of every APL function definition must be the function header line. The function header line mentions the name of the function; the names of its arguments; the name of its results; and the names of variables which are local to it, if any.

The format of a header line is a sample function reference followed by the names of the local variables set off by semicolons. For example, a dyadic function named *NAME*, taking a left argument named *L* and a right argument named *R*, producing a result named *Z*, and having local variables *A*, *B*, and *C* would have the header line:

```
Z←L NAME R;A;B;C
```

A monadic function *F* of one argument *ZAP* producing no result and having no local variables would have the header line:

```
F ZAP
```

Since in APL there are two choices as to result (either return a result or not) and three choices as to number of arguments (0, 1, or 2), there are six different kinds of functions and hence six corresponding different kinds of header lines. Examples of the six kinds are:

Z←L NAME R	(dyadic, result)
Z←NAME R	(monadic, result)
Z←NAME	(zero-adic, result)
L NAME R	(dyadic, no result)
NAME R	(monadic, no result)
NAME	(zero-adic, no result)

Each of these six can have a list of local variables on it as well.

The function header line can be edited as any other line. It is not necessary for a valid header line to be present at all times during the edit, but one must be present in order to exit from the editor.

LEAVING THE EDITOR

When the user has made his desired modifications to the function definition, he terminates the edit either by issuing the *Q* editing request or by typing the editor-exit character ∇ at the end of a line (or on a line by itself). The ∇ character terminates the edit, whether typed in input mode or in edit mode, following the normal processing of whatever precedes it on the line. The ∇ character may appear only at the end of a line; it is not possible to input one into a function definition (unless it is in a quoted string, in which case it is treated as data and will not terminate the edit). Any other use of the ∇ character results in an error message.

At the time the user requests termination of the edit, if the function has no lines then it is deleted. Otherwise, the editor verifies that the first line of the function is a valid header line and analyzes all lines of the function into their syntactic constituents (names, operators, constants). If any errors are detected either in the header line or in the syntactic scan, they are reported and editor remains in edit mode. The editor does not terminate until the header line and syntactic scan are verified as correct.

If the name of the function in the function header line was edited, the function will be known by the new name when the edit terminates. The new name will have been checked for naming conflicts with groups, global variables, and other functions before the edit is allowed to terminate. The name under which the function was previously known is freed for other uses. Any other information in the header line can be changed as well: the names or number of arguments required, the name or existence of the result, and the names of local variables. At the conclusion of the edit, the function acquires whatever characteristics its new header line specifies for it, regardless of its previous characteristics.

When the editor invocation terminates, the APL interpreter again requests the input it was awaiting before the edit.

EDITING REQUESTS

The APL function editor is a line-by-line editor. The editor's attention is always on a single line of the function being edited; whatever line that happens to be, at any given instant, is known as the "current line". The user issues "requests" to the editor which either initiate some editing activity on the current line or else move the editor's attention to some other desired line. When the editor is invoked, the current line is initially a hypothetical, nonexistent line positioned in front of the first line of the function.

Each request is identified by a single-character name. The user issues a request by typing a line beginning with that character. Some requests require arguments or additional information to be typed following their single-character identifier. Such information can be separated from the request identifier by zero or more blanks.

Note that what the editor considers to be a "line" is not always the same as what the APL interpreter considers to be a statement. Because character constants can contain newline characters, a single statement of APL code to be interpreted may have been assembled from several source lines (editor lines) in the definition of the function. The editor is unaware of the structure of APL; it is simply a general text editor.

Basic Editing Requests

The requests discussed in this section are the minimum set necessary to make use of the editor. They are the requests that a beginner will want to learn to use first.

PRINT REQUEST *P* n

Beginning with the current line, "n" lines of the function are printed. The last line printed becomes new current line. If "n" is omitted, it is assumed to be 1; i.e., only the current line is printed.

If the current line does not exist (because it was deleted, or because it is the hypothetical line preceding the first actual line of the function), then *NO LINE* is printed. If the end of the function is encountered before the requested number of lines have been printed, then *EOF* (for end-of-function) is printed.

```
      ∇PLOT                          Open function PLOT for edit-
 EDIT.                               ing.  It is already defined.
 P3                                  Print three lines.
 NO LINE.                            The nonexistent line.
 X PLOT Y;M;A                        Two more lines.
 A←3053ρ' '
 P                                   Print request with no integer
 A←3053ρ' '                          prints only the current line.
 P 99                                Print the entire remainder of
 A←3053ρ' '                          the function. (Starting with
 M←35÷⌈/|X,Y                         the current line.)
 A[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'
 43 71ρA
 EOF.                                End of the function reached.
 P                                   Any attempt to print more is
 EOF.                                unsuccessful.
```

LOCATE REQUEST   *L* string

Beginning with the line after the current line, successive lines of the function are searched for an occurrence of the character string "string" in them. As soon as a line containing "string" is found, it becomes the current line, and it is printed. The character string "string" can be set off from the *L* request by zero or one blank. If additional blanks are typed, they are considered to be part of "string" and are included in the search.

If the end of the function is reached before a line containing the string is found, the search resumes from the beginning of the function and continues until the line before the current line (i.e., the current line is the only line not searched). If none of the searched lines contains an instance of "string", the editor types *SORRY* and the current line status remains unchanged.

The sequential nature of the search, plus the exclusion of the current line from the search, implies that all instances of a particular string in a function can be located successively and in order by issuing repeated locate requests for the string. As a typing convenience, the editor considers the *L* request alone, without a string following it, as a request to locate the same string last mentioned in a locate or find request.

```
 L A                                 Locate a line containing the
 X PLOT Y;M;A                        string A.  Editor prints it.
 L A                                 Locate the next A.
 A←3053ρ' '                          It is in this line.
 L                                   Locate the next again.  Editor
 A[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'  knows we mean A.
```

```
L 3 7                              String can contain blanks.
43 71ρA
L ÷⌈/                              Or special characters.
M←35÷⌈/|X,Y
L '                                Locate quote.
A[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'
L  '                               Locate blank quote.
A←3053ρ' '
L   '                              Locate blank blank quote.
SORRY.                             No such string in the func-
P                                  tion.  Current line unchanged.
A←3053ρ' '
```

INSERT REQUEST   *I* string

The string "string", which is typed to the right of the *I* request character, is inserted as a new line in the function, immediately following the current line. Thus, successive insert requests allow a sequence of lines to be inserted at any desired point in a function. As in the locate request, the "string" can be separated from the *I* character by an optional blank; any additional blanks are considered to be part of "string" and are inserted into the function. No typed response occurs from the editor for the insert request.

```
P                                  Where are we now?
A←3053ρ' '                         Still here.
I ⍝ 3053 IS 43×71.                 Insert a new line into the
P                                  function.  Now print it.
⍝ 3053 IS 43×71.                   Yes, it's really there.
L 3053                             Make sure it got put in the
A←3053ρ' '                         right place by going back to
P 2                                the line in front of it and
A←3053ρ' '                         printing two lines.
⍝ 3053 IS 43×71.
I THIS IS A NEW LINE.              Insert some more lines.
I SO IS THIS.
P 2
SO IS THIS.                        Current line is last line
M←35÷⌈/|X,Y                        inserted.
L 1ρ
43 71ρA
I ⍝ END OF FUNCTION.
P 333
⍝ END OF FUNCTION.
EOF.
```

DELETE REQUEST   *D* n

Starting with the current line, "n" lines of the function are deleted. If "n" is omitted, the default is 1; i.e., only the current line is deleted. The editor types no response to the delete request, unless the end of the function is reached before the requisite number of lines are deleted, in which case it responds *EOF* (for end-of-function).

Following a delete request, the current line is a hypothetical, nonexistent line holding the place of the last line deleted. Thus, an insert request issued after a delete is one way of replacing a line of the function with a new line.

```
L THIS                           Take out the two non-APL lines
THIS IS A NEW LINE               to make the function correct
D 2                              again.
P
NO LINE.                         Current line was deleted.
L A←
A←3053ρ' '
P 3
A←3053ρ' '
⍝ 3053 IS 43×71.                 No evidence remains of the
M←35÷⌈/|X,Y                      deleted lines.
L 05
A←3053ρ' '                       Back to our favorite line.
D  P 2                           Bye-bye.
NO LINE.
⍝ 3053 IS 43×71.                 We were still at the same
L ⍝                              place in the function.
⍝ END OF FUNCTION.               Locate that line we put at the
D 77                             end.
EOF.                             Delete all from here out.
L PLOT                           End of function reached.
X PLOT Y;M;A                     It is time to stop and take a
P 1000                           look at our entire function
X PLOT Y;M;A                     again.
⍝ 3053 IS 43×71.
M←35÷⌈/|X,Y
A[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'
43 71ρA  EOF.
L ⍝
⍝ 3053 IS 43×71.
D                                Oops, we better fix up that
I A←(43×71)ρ' '                  initialization of A again.
P 4
A←(43×71)ρ' '
M←35÷⌈/|X,Y
```

```
A[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'
43 71ρA
```

RETYPE REQUEST   *R* string

The retype request replaces the current line with "string". It behaves exactly like a delete request followed by an insert request. As in the locate and insert requests, if more than one blank separates "string" from the *R* request character, the additional a blanks are taken as part of the retyped line.

```
L 43                              Let's make plot return its
A←(43×71)ρ' '                     answer instead of printing it.
L                                 Oops, wrong 43.  Get the next
43 71ρA                           one.  That's better.
R Z←43 71ρA                       Retype the line.
P
Z←43 71ρA                         Make sure.
L PLOT
X PLOT Y;M;A                      We need to fix the header line
R Z←X PLOT Y;M;A                  too.
P 38934934                        Display the new definition.
Z←X PLOT Y;M;A
A←(43×71)ρ' '
M←35÷⌈/|X,Y
A[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'
Z←43 71ρA
EOF.
```

TOP REQUEST   *T*

The top request moves the editor's line pointer to the hypothetical, nonexistent line located ahead of the first actual line of the function. Due to the sequential or "one-line-to-the-next" nature of many of the other requests, it is often desirable to start an edit pass at the top of the function and work downward, making alterations in sequence.

The editor types no response to the top request. The nonexistent line is printed as *NO LINE* by the print request.

A purpose served by the hypothetical line in front of the first real function line is to permit insertion of a new line ahead of the first line of a function (for example, supplying an omitted header line).

```
T
P 3
NO LINE.
Z←X PLOT Y;M;A
A←(43×71)ρ' '
T
I NEW HEADER LINE.
T
I NEWER HEADER LINE.
T
I NEWEST HEADER LINE.
I NOT QUITE SO NEW.
I NOT NEW AT ALL.
T
P 7
NO LINE.
NEWEST HEADER LINE.
NOT QUITE SO NEW.
NOT NEW AT ALL.
NEWER HEADER LINE.
NEW HEADER LINE.
Z←X PLOT Y;M;A
T
D 6
P 3
NO LINE.
Z←X PLOT Y;M;A
A←(43×71)ρ' '
```

CHANGE REQUEST *C* n /string1/string2/

The change request is used for character editing within a line. For the next "n" lines beginning with the current line, all occurrences of the character string "string1" are changed to "string2". If "n" is omitted, only the current line is affected. At the conclusion of the change request, the new current line is the last of the lines inspected (whether or not there were any occurrences of "string1" in that line).

All lines actually changed are printed in their new form. If no lines changed (because there were no instances of "string1" in any of the selected lines), then *SORRY* is printed as a warning. If the end of the function is reached before the requisite number of lines has been processed, then *EOF* is printed.

The character used to delimit the strings in the request line need not be "/". Any character not appearing in the strings can be used. Any character encountered before "string1" is used as the delimiter. If there are no trailing blanks in "string 2", then the last delimiter can be omitted. Using less than two or more than three delimiters in the request is an error, however.

Either "string1" or "string2" can be zero characters in length (i.e., adjacent delimiters with no characters between them). When "string1" is null, it is considered to match the beginning of the function line; hence, "string2" will be prefixed to the existing function line. When "string2" is null, it means that occurrences of "string1" disappear (they are replaced by no characters).

```
P                             Where are we?
A←(43×71)ρ' '
C /(43×71)/3053/              Put the line back the way it
A←3053ρ' '                    was.  Editor prints new line.
C /3/+⌽?/
A←+⌽?05+⌽?ρ' '                Both instances of' 3 'changed.
C .+⌽.⌈/⌊.                    Use of a different delimiter
A←⌈/⌊?05⌈/⌊?ρ' '              character.
C .⌈.3.
A←3/⌊?053/⌊?ρ' '
C ./⌊?..                      Character deletion: null
A←3053ρ' '                    replacement string.
T                             Start at the top of the func-
C 999 /A/ARRAY/               tion and change all references
Z←X PLOT Y;M;ARRAY            of ' A ' to ' ARRAY '.  The editor
ARRAY←3053ρ' '                prints the changed lines.
ARRAY[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'
Z←43 71ρARRAY
EOF.                          End of function hit.
I THIS IS A TEST.             Get a line to play with.
C /IS/AT/
THAT AT A TEST.               Both strings changed.
C /AS/IS/
SORRY.                        No ' AS ' in the line.
C /AT A /WAS A/               Carefully specify enough char-
THAT WAS ATEST.               acters to exclude ' THAT '.
C /AT/A T/                    Now need a blank before
THA T WAS A TEST.             ' TEST '.
C / T /T
THATWAS A TEST.               Some days nothing works right.
C /THATWAS/THAT WAS/          Do it more carefully now.
THAT WAS A TEST.
C/S A T/S SOME T/
THAT WAS SOME TEST.
C/EST/RICK
THAT WAS SOME TRICK.
C ..JOE, .                    Null string1.
JOE, THAT WAS SOME TRICK.
C / //                        Null string2.
JOE,THATWASSOMETRICK.
C /JOE,T//
```

```
HATWASSOMETRICK.
C /TWA//
HASSOMETRICK.
C /METRI//
HASSOCK.
```

INPUT REQUEST   .

The character . (period) identifies the input request. When . is typed, the editor responds *INPUT* and switches to input mode. In input mode, lines typed by the user are inserted into the function as new lines. When the editor responds *INPUT*, all lines typed are inserted into the function following the current line. To leave input mode, the user types a line consisting solely of . (another period). Then the editor responds *EDIT*, and any following lines are interpreted as editing requests. The current line at the conclusion of input mode is the last line inserted. Thus, input mode behaves exactly like repeated insert requests.

Note that if an editing request is typed erroneously in input mode, it is not recognized as such, and the typed line is inserted into the function without any warning message. Also, if the terminating period for exiting from input mode is not typed by itself on a line of its own, it goes into the function and does not cause a termination of input mode.

When the editor is invoked to edit a new function (one which does not already exist), it responds in input mode. If the function already exists, the editor responds in edit mode with the current line set before the first line of the function.

```
L A←305                          Rework the plotting program to
SORRY.                           adjust all the borders of the
L Y←305                          graph to hold just the data to
ARRAY←3053ρ' '                   transpose be displayed.
.
INPUT.
M←58÷(⌈/X)-XORG←⌊/X              Scale the data to a width of
XORG←1.5-M×XORG                  5.8 inches.
YSIZE←1-⌊(YORG←0.5-0.6×M×⌈/Y)-0.6×M×⌊/Y
.                                And as much height as the data
EDIT.                            requires at that scale.
L 53
ARRAY←3053ρ' '
P 5
ARRAY←3053ρ' '
M←58÷(⌈/X)-XORG←⌊/X              The new lines were inserted at
XORG←1.5-M×XORG                  the correct place.
```

```
YSIZE←1+⌊(YORG←0.5+0.6×M×⌈/Y)-0.6×M×⌊/Y
M←35÷⌈/|X,Y
D

.
INPUT.
A←(YSIZE×59)ρ' '

.
EDIT.
L PLOT
Z←X PLOT Y;M;ARRAY            Add new local variables.
C /AY/AY;XORG;YORG;YSIZE/
Z←X PLOT Y;M;ARRAY;XORG;YORG;YSIZE
C 99 /RRAY//
Z←X PLOT Y;M;A;XORG;YORG;YSIZE
A←3053ρ' '
A[(⌊36.5+X×M)+71×⌊×21.5-Y×0.6×M]←(ρX)ρ'*'
Z←43 71ρA
EOF.
L A←
A←3053ρ' '
D
L M←
M←58÷(⌈/X)-XORG←⌊/X
L
SORRY.                        Already deleted the other one.
L A[
A[(⌊36.5+X×M)+71×⌊21.5-Y×0.6×M]←(ρX)ρ'*'
D 2
INPUT.
A[(⌊XORG+X×M)+59×⌊YORG-Y×0.6×M]←(ρX)ρ'*'
Z←(YSIZE,59)ρA

.
EDIT.
T
P 999
NO LINE.
Z←X PLOT Y;M;A;XORG;YORG;YSIZE
M←58÷(⌈/X)-XORG←⌊/X
XORG←1.5-M×XORG
YSIZE←1+⌊(YORG←0.5+0.6×M×⌈/Y)-0.6×M×⌊/Y
A←(YSIZE×59)ρ' '
A[(XORG+X×M)+59×⌊YORG-↑×0.6×M]←(ρX)ρ'*'
Z←(YSIZE,59)ρA
HASSOCK.                      Oops.
EOF.
L H                           Forgot about that.
HASSOCK.
D                             There.
```

QUIT REQUEST *Q*

The quit request *Q* is the last of the basic editing requests. It is issued when editing is completed and it is desired to exit from the editor. Issuing the quit request is not necessary; an alternate method of terminating an edit session is to type an editor-exit character ∇ at the end of a line. The ∇ character terminates the edit whether typed in input mode or edit mode; the *Q* editing request is recognized only in edit mode.

When the edit is ending, the editor checks the syntax of the function header line and verifies that all lines of the function have recognizable syntactic constructs. See also "Leaving the Editor" in this section.

If the checks are successful, the editor allows termination of the edit. The APL processor again requests the immediate or evaluated input it was awaiting at the time the editor was invoked.

```
Q                            Exit from the editor.  APL
      )FNS                   types six spaces, it is again
PLOT                         ready for immediate input.
      (ι20) PLOT 2×1○(ι20)   Try plotting a sine curve.

   *                 *                 *                 *
*                 *                       *
                        *           *
      *                                               *

               *                             *
         *                 *     *                 *
            *                 *                 *
```

Other Useful Requests

While the nine requests already described are the basics that a user needs to accomplish any editing task, the next four requests, described on the following pages, can be very convenient.

NEXT REQUEST *N* n

The next request is used to move the editor's attention "n" lines further along in the function. More precisely, beginning with the current line, "n"-1 lines are skipped, the next line becomes the current line. If "n" is omitted, it is assumed to be a 1; i.e., the next line becomes the current line.

The new current line is printed. If the end of the function is encountered before a sufficient number of lines have been advanced, then *EOF* is printed.

```
      ∇PLOT
EDIT.
N
Z←X PLOT Y;M;A;XORG;YORG;YSIZE
N
M←58÷(⌈/X)-XORG←⌊/X
N
XORG←1.5-M×XORG
N 2
A←(YSIZE×59)⍴' '
N 28
EOF.
T
N 7
Z←(YSIZE,59)⍴A
N
EOF.
```

BACKUP REQUEST - n

The backup request is similar to the *N* request, except that it moves the editor's attention backward. That is, the current line becomes the "n"th previous line. The backup request is useful when one has accidentally passed by a line that requires changing.

The new current line is printed. If the beginning of the function is encountered before a sufficient number of preceding lines have been located, then *BOF* (for beginning-of-function) is printed.

```
P                                  Where are we?
EOF.                               At the end of the function.
-                                  Back up one to see the last
Z←(YSIZE,59)⍴A                     line.
- 2                                Back up two more.
A←(YSIZE×59)⍴' '
- 9999                             Back up a lot.
BOF.                               Too far.  The -9999 request
N                                  has the same effect as T.
Z←X PLOT Y;M;A;XORG;YORG;YSIZE
N 9999                             Now down to the bottom again.
EOF.
-
Z←(YSIZE,59)⍴A
```

BOTTOM REQUEST *B*

The bottom request is a convenient way to add lines to the end of a function. The bottom request sets the current line following the last line of the function, prints *INPUT*, and enters input mode. All lines typed will be appended to the end of the function. Exit from input mode with the ∘ character.

The result of the *B* request is the same as that of a *N* 9999 followed by a . except that *EOF* is not printed.

```
B
INPUT.
NEW LINE.
NEWER LINE.
.
EDIT.
P
NEWER LINE.
-
NEW LINE.
-
Z←(YSIZE,59)⍴A
P 7
Z←(YSIZE,59)⍴A
NEW LINE.
NEWER LINE.
EOF.
```

FIND REQUEST *F* string

The find request is similar to the locate request except that to satisfy the search, the "string" must be found at the beginning of a line searched instead of being embedded anywhere in it. The find request is useful for finding labels or assignments to variables.

```
P                                Where are we?
EOF.
T                                Go to the top.
F M                              Find an ' M ' at the beginning
M←58÷(⌈/X)-XORG←⌊/X              of a line.  Editor prints it.
F                                Find another one.
SORRY.                           There are no others.
L                                But there are others not at
XORG←1.5-M×XORG                  the beginning of a line.
L
YSIZE←1+⌊(YORG←0.5+0.6×M×⌈/Y)-0.6×M×⌊/Y
F NEW                            Get to those lines we added.
NEW LINE.
```

```
D 2                             Delete them.
F
SORRY.
```

Infrequently Used Requests

The nine requests described on the following pages serve specialized purposes. They are not used as frequently as the previous requests, but each is useful when used appropriately as described below.

KILL REQUEST *K*

The kill request causes the editor to suppress normal responses to the change, locate, find, next, and backup requests. After a *K* request has been issued, the editor no longer prints the current line reached by *L*, *F*, *N*, and - requests, nor does it print the lines changed by the *C* request.

The kill request does not suppress the *INPUT* and *EDIT* responses to the . request, nor does it suppress abnormal responses like *SORRY* and *EOF*. There is no response to the kill request itself.

Editing of a large function can be accelerated if responses are killed, but the user is cautioned to specify long enough strings to uniquely locate and change the desired items. If a locate request is satisfied by an unexpected match, additional editing can occur before discovery of the discrepancy, with the result that a large number of corrections may have to be made.

```
K                               Suppress responses.
T
C 999 /M/MAG/                   Change all M 's to MAG 's.
EOF.                            Changed lines not printed.
T
C 999 /Z/ANSWER/
EOF.
T
L ←(YSIZE
C .YSIZE×.×/YSIZE←YSIZE,.       Located line not printed.
.L                              Get the next use of  YSIZE .
C /(//                          Edit it, too.
C /,59)//
T
P 99                            Now print the whole new
NO LINE.                        function.
```

```
ANSWER←X PLOT Y;MAG;A;XORG;YORG;YSIZE
MAG←58÷(⌈/X)-XORG←⌊/X
XORG←1.5-MAG×XORG
YSIZE←1+⌊(YORG←0.5+0.6×MAG×⌈/Y)-0.6×MAG×⌊/Y
A←(×/YSIZE←YSIZE,59)ρ' '
A[(⌊XORG+X×MAG)+59×⌊YORG-YORG-Y×0.6×MAG]←(ρ'*'
ANSWER←YSIZEρA
EOF.
```

VERIFY REQUEST *V*

The verify request permits printing of responses suppressed by the kill request to begin again. There is no response to the *V* request itself.

```
V                                   Responses back on again.
L 1.5
XORG←1.5-MAG×XORG                   Located line printed.
T
C 500 /YSIZE/SIZE/                  Changed lines printed.
ANSWER←X PLOT Y;MAG;A;XORG;YORG;SIZE
SIZE←1+⌊(YORG←0.5+0.6×MAG×⌈/Y)-0.6×MAG×Y
A←(×/SIZE←SIZE,59)ρ' '
ANSWER←SIZEρA
EOF.
K                                   Now let's put it back.
T
C 50 /ANSWER/Z/
EOF.
T
C 50 /AG//
EOF.
T
C 50 /ORG/G/
EOF.
T
C 50 /IZE//
EOF.
T
P 50                                See what we got.
NO LINE.
Z←X PLOT Y;M;A;XG;YG;S
M←58÷(⌈/X)-XG←⌊/X
XG←1.5-M×MG
S←1+⌊(YG←0.5+0.6×M+⌈/Y)-0.6×M×⌊/Y
A←(×/S←S,59)ρ' '
A[(⌊XG+X×M)+59×⌊YG-Y×0.6×M]←(ρX)ρ'*'
Z←SρA
EOF.
```

LINE-NUMBER REQUEST =

The line-number request, triggered by the equal-sign character, prints the line number of the current line. For the purposes of this request, the lines of the function are numbered sequentially, beginning with the header line as number one. The numbers are dynamic, and change as lines are added to and deleted from the function. If there is no current line, then *NO LINE* or *EOF* is printed as a response to the = request.

The line numbers returned by the line-number request are determined by the locations of newline characters in the textual definition of a function. They do not correspond to the interpreter statement numbers printed out by the)*SI* system command. The two sets of numbers differ because the interpreter considers the header statement of a function to be statement number zero, and because newline characters within character constants can cause several editor lines to be taken as one statement to be interpreted.

```
L XG←
=
2
L XG←
=
3
L XG←
=
2
L S←
=
4
L S←
SORRY.
N 9999
EOF.
=
EOF.
-
=
7
T
N 4
=
4
```

COMMENT REQUEST ,

The comment request is initiated by the comma character. After a comment request, successive lines of the function, beginning with the current line, are typed out one at a time, without a newline character at the end. The user may then add additional characters to the line displayed, or simply type a newline to leave the line unchanged. When the newline is typed, either with or without additional characters having been appended to the displayed line, the editor advances to the next line of the function and repeats the operation.

The user can exit from the comment mode by typing a single period on a line. The period cannot be preceded by a comment for the current line. Upon conclusion of the comment operation, the current line is the last line displayed. An automatic exit from the comment mode occurs if the end of the function is encountered.

```
 V                            Verify mode turned on.
 L Z←
 Z←S⍴A
 L
 Z←X PLOT Y;M;A;XG;YG;S
 ,                            The user requests comment mode
 Z←X PLOT Y;M;A;XG;YG;S       and the editor prints the cur-
                              rent line, without a carriage
                              return. The user now appends
                              his comments.
 (already typed)              (new text typed by user)
 Z←X PLOT Y;M;A;XG;YG;S       ⍝ PLOT GRAPH ON TYPEWRITER.
 M←58÷(⌈/X)-XG←⌊/X            The editor now advances to the
                              next line and displays it.
                              Again, we append some comments
                              to the line.
 (already typed)              (new text)
 M←58÷(⌈/X)-XG←⌊/X            ⍝ 58 COLUMNS FOR RANGE OF X'S.
 XG←1.5-M×XG                  Editor prints the next line.
                              This time we elect not to add
                              anything, so we enter an im-
                              mediate newline. The editor
                              responds with the next line.
S←1+⌊(YG←0.5+0.6×M×⌈/Y)-0.6×M×⌊/Y
                              To which we comment:
 (already typed)              (new)
 S←1+⌊(YG←0.5+0.6×M×⌈/Y)-0.6×M×⌊/Y   ⍝ VERTICAL SIZE NEEDED.
 A←(×/S←S,59)⍴' '             Next line displayed. Now we
                              exit from comment mode by
                              typing a period.
 (already typed)              (edit)
 A←(×/S←S,59)⍴' '.  EDIT.
P                             Editor responds in edit mode.
 A←(×/S←S,59)⍴' '
 T
 P 6
```

```
NO LINE.
Z←X PLOT Y;M;A;XG;YG;S           ⍝ PLOT GRAPH ON TYPEWRITER.
M←58÷(⌈/X)-XG←⌊/X                ⍝ 58 COLUMNS FOR RANGE OF X'S.
XG←1.5-M×XG
S←1+⌊(YG←0.5+0.6×M×⌈/Y)-0.6×M×⌊/Y    ⍝ VERTICAL SIZE NEEDED.
A←(×/S←S,59)ρ' '
```

EXECUTE REQUEST   *E* command

The execute request is used to execute a Multics command without exiting from the function editor. The remainder of the line following the *E* request character is passed to the Multics command processor, via "cu_$cp", for execution. The editor responds *EDIT* when the command has been completed. The current line is unchanged.

The APL character set conventions are suspended when control leaves APL and are restored when control returns. The user is cautioned that APL character set conventions are in effect during reading of the execute request line itself. The APL character set and the Multics character set have a considerable amount of overlap, but the user must be aware of the differences and compensate for them when necessary. Character set considerations are fully discussed in Section II.

```
E CWD >UDD>M>MGS                       Execute a Multics command.
EDIT.
E PWD
>UDD>|ULTICS>|⌈MITH                    Multics thinks we have a
EDIT.                                  standard type ball mounted.
E HMU                                                 This time we
Multics 20.12b, load 26.5/50.0; 28 users              will change the
Absentee users 1/2                                    ball.
edit.
P                                      Change back to APL ball.
A←(×/S←S,59)ρ' '                       Current line, as before.
```

DELETE-ABOVE REQUEST   *X*

The *X* request causes all lines above the current line to be deleted. The current line is unchanged. This request is most useful for copying, deleting, and rearranging large blocks of text, particularly in conjunction with the *U* and *M* requests, discussed below. There is no response to the *X* request.

```
T                                      Put in some lines to delete.
.
```

```
INPUT.
ONE
TWO
THREE
FOUR
FIVE
.
EDIT.
F TWO
TWO
X
P
TWO                                Current line unchanged.
T
P 4
NO LINE.                           But nothing above it left.
TWO
THREE
FOUR
X
T
P 4
NO LINE.
FOUR
FIVE
Z←X PLOT Y;M;A;XG;YG;S             ⍝ PLOT GRAPH ON TYPEWRITER.
X                                  Function is now restored.
L A←
A←(×/S←S,59)⍴' '
=
5
X
=                                  The line numbers change as the
1                                  function is edited.
```

WRITE REQUEST   *W* pathname

The write request causes the current source text definition of an APL function to be written as a Multics segment. The argument of the *W* request is an absolute or relative pathname. The source is represented in terms of APL internal character codes. The editor's current line is at the end of the function following a write request. There is no response typed to this request.

The write request bears no relation to the status of the function definition as far as APL is concerned. All editing operations instantly change the definition of the function stored in the workspace; no write or other request is necessary to cause them to take effect. The write request is used solely to communicate a function definition outside of the APL environment.

The APL function editor and the write request form a convenient way of generating text in APL internal character codes, as is needed when input to APL comes from a segment rather than a terminal (absentee usage of APL, or the "&attach" feature of "exec_com"). If, after the write request is issued, all the lines of text are deleted (as, with the *X* request), then no function will be placed in the current workspace.

```
D 99                                Discard the original function
EOF.                                entirely.
.
INPUT.
'ARBITRARY APL TEXT'
A←3 4ρι12
A+.×⍉A
⌹A+?3 4ρ10
.
EDIT.
W TEST
E LS TEST

⌈EGMENTS= 1, ρECORDS= 1.            Multics thinks we have a
                                    standard type ball.
08/31/72  2305.5  RWA     1  TEST

EDIT.
E PR TEST 1 99                      Change to standard type ball
'arbitrary apl text'                for this print.  Note that
a{3 4%$12                           some APL character codes do
a+.&¢234a                           not have ASCII equivalents.
¢244a+?3 4%10
edit.                               Change back again.
Q                                   If we try to end the edit at
INVALID HEADER LINE.                this point, APL will not let
PLOT[0] 'ARBITRARY APL TEXT'        us, because we do not have a
        |                           valid header line.
X                                   We are not interested in the
Q                                   function any more.
      )FNS                          There is no longer a PLOT
                                    function.  Deleting all of a
                                    function's lines is a way of
                                    erasing it.
```

MERGE REQUEST   *M* pathname

The merge request is the complement of the write request. All lines of the segment whose pathname is given in the merge request are inserted into the function definition following the current line. The new current line becomes a hypothetical, nonexistent line placed after the last inserted line. The text in the segment being merged must be in APL internal character codes if the resultant function definition is to be usable by APL. The segment is unchanged. No response is typed to the *M* request.

The effect of the merge request is similar to repeated *I* requests in which each line of the segment is typed, one after the other, except that the current line is set beyond the last line inserted.

```
      ∇TEST                         Enter the editor again.
 FUNCTION 'TEST' NOT FOUND.
 INPUT.
 .                                  We do not desire to type in a
 EDIT.                              definition.  Go to edit mode.
 M TEST                             Merge in segment  TEST .
 T
 P 99
 NO LINE.
 'ARBITRARY APL TEXT'
 A←3 4ρι12
 A+.×⍉A
 ⌹A+?3 4ρ10
 EOF.
 L A←
 A←3 4ρι12
 M TEST                             Merge in another copy here.
 T
 P 99
 NO LINE.
 'ARBITRARY APL TEXT'
 A←3 4ρι12
 'ARBITRARY APL TEXT'
 A←3 4ρι12
 A+.×⍉A
 ⌹A+?3 4ρ10
 A+.×⍉A
 ⌹A+?3 4ρ10
 EOF.
 W TEST                             Write all 8 lines back out.
 M TEST                             Now we have 16.
 W TEST
 M TEST                             32.
 W TEST
 M TEST                             64.
 W TEST
 M TEST                             128.
 -
 ⌹A+?3 4ρ10
 =
 128
```

WRITE-ABOVE REQUEST   *U* pathname

The write-above request is similar to the delete-above request, except that it writes out the lines above the current line as a Multics segment before deleting them from the function definition. The current line is unchanged.

The write-above, delete-above, and merge requests can be used to rearrange blocks of text by using successive *U* requests to distribute the blocks to individual segments (or *X* requests for blocks which need not be saved). Then the *M* request is used to rearrange the into a new order.

```
T                                   An example of moving blocks
D 999                               of text around.
EOF.
.
INPUT.                              Input some arbitrary text.
BLOCK A.
ONE.
TWO.
THREE.                              Assume, for example, that it
BLOCK B.                            is desired to rework this
FOUR.                               definition from the order
FIVE.                               A-B-C-D into the order D-A-C
BLOCK C.                            (B being deleted).
SIX.
SEVEN.
EIGHT.
NINE.
BLOCK D.
BINGO!
.
EDIT.
T
L B                                 Find the beginning of block B.
BLOCK A.                            Oops, wrong B.
L B.
BLOCK B.                            That's better.
U FILEA                             Write out block A as a Multics
L C.                                segment.
BLOCK C.
X                                   Delete block B.
L D.
BLOCK D.
U FILEC                             Save block C.
N 999                               Block D is everything left.
EOF.                                Get to the bottom of block D.
M FILEA                             Merge in the other blocks in
M FILEC                             their correct positions.
T
P 999
```

```
NO LINE.
BLOCK L.
BINGO!
BLOCK A.
ONE.
TWO.
THREE.
BLOCK C.
SIX.
SEVEN.
EIGHT.
NINE.
EOF.
E DF FILEA FILEB TEST        Delete the temporary segments
EDIT.
```

Synonyms for Requests

A number of the editor requests have alternate names. Each of the following alternate names invokes a corresponding request that has already been discussed.

SUBSTITUTE REQUEST *S* n /string1/string2/

The substitute request, invoked by the character *S*, is the same as the change request, invoked by the character *C*.

EXECUTE REQUEST E̲ command

The execute request invoked with the underscored letter, E̲ (represented as upper case "E" on ASCII terminals), is the same as the execute request invoked with the plain letter, *E* (represented as lower case "e" on ASCII terminals).

UPWRITE REQUEST *UPWRITE* pathname

The upwrite request, *UPWRITE*, is the same as the write-above request, *U*. The upwrite request is an exception to the rule that a request urr==zkication consist of a single character: the entire word *UPWRITE* must be spelled out to invoke this request.

UPDELETE REQUEST *UPDELETE*

The updelete request, *UPDELETE*, is exactly the same as the delete-above request, *X*. The updelete request is another exception to the rule that a request identification consist of a single character.

MERGE REQUEST *MERGE* pathname

The merge request is invoked by either spelled-out *MERGE* or by the single character *M*. The spelled-out merge request is the final exception to the single-character request identification rule.

PROGRAM INTERRUPT DURING EDITING

As elsewhere in APL, the Multics "program_interrupt" condition remains enabled while editing. If the user wishes to abort a lengthy printout or other time-consuming editor operation, he presses the QUIT button and issues the Multics "pi" command. The editor returns to edit mode and prints *EDIT*. If the interrupted request was in the process of altering the function, the process may not have finished and the user should inspect several preceding lines of the function to determine the effect of the interrupted request.

The program interrupt feature is safe in the sense that its use never leaves the function definition in an inconsistent or unreadable state.

THE SNEAK REQUEST

As in APL/360, in Multics APL it is possible to "sneak" one editor request onto the same line as the editor invocation. The request is separated from the name of the function being edited by at least one blank.

When a sneak request is found on an editor invocation line, the editor does not respond *EDIT* as usual, but immediately proceeds to execute the request. Whether a response is printed depends on the request. When the request ends with the edit-termination character ∇, the edit terminates after execution. Thus, an entire edit, including invocation, request, and termination, can be accomplished on one typed line.

If a function definition does not exist when a sneak request is found on an editor invocation, it is assumed that a mistake was made by the user. Thus, the editor ignores the request and responds with *INPUT* mode, as usual.

```
Q                            Exit from current edit.
HEADER LINE SYNTAX ERROR.    Oops, the text we were using
TEST[0] BLOCK D.             does not look like a valid APL
               |             function.
EDIT.
T
C 2 /.//                     Make it so.
BLOCK D                      Now it does.
Q                            Exit successfully.
      )FNS                   Note that the function TEST
BLOCK                        was renamed BLOCK by its
     ∇BLOCK P 99∇            header line.  Here is an
NO LINE.                     example of listing a function
BLOCK D                      definition by means of a sneak
BINGO!                       request.  The editor invoca-
BLOCK A.                     tion, request, and termination
ONE.                         are all effected on one line.
TWO.
THREE.
BLOCK C.
SIX.
SEVEN.
EIGHT.
NINE.
EOF.
     ∇BLOCK F T              A sneak request not followed
TWO.                         by a termination.
C/O/O HUNDRED/
TWO HUNDRED.
B
INPUT.
TEN.∇
     BLOCK 5                 Invoke the function.
VALUE ERROR.                 It isn't really a valid APL
BLOCK[1] BINGO!              function.
         |
     ∇BLOCK D 99∇            Delete it with a sneak request
EOF.                         which deletes all of its
      )FNS                   lines.  Not there any more.
```

SECTION VI

DEMONSTRATION APL SESSION

APL IN ACTION

A demonstration of APL in action is shown below. The user's requests and the computer's responses are shown as they occurred, with running commentary.

```
apl
      2×3
6
      6.3×2.7
17.01
      X←1÷3
      X
0.3333333333
      2×3+4
14
      (2×3)+4
10
      ÷X
3
      ⍳4
1   2   3   4
      3  5  7  2
3   5   7   2
      3  5  7  2⋆⍳4
3   25   343   16
      100-⍳3
99   98   97
      ⍴4  2  6
3
      4  2  6+1  2  3  4
LENGTH ERROR.
      4  2  6+1  2  3  4
             |
      X←5⍴⍳3
      X
1   2   3   1   2
      Y←6  6
      X,Y
```

Invoke Multics APL command, change type ball. User types expression to be evaluated, APL types answer. User input is always indented; answer is not. If answer assigned to a variable, then not printed. Value of the variable *X* . Operations are executed in right-to-left order. But parentheses can be used to specify a different order. Monadic ÷ is reciprocal.

Index generator operator generates a vector of sequential integers. A vector may be input by the user too. Scalar operators apply element-by-element to vectors. ⋆ is exponentiation. A scalar is applied to each element of a vector. ⍴ of a vector is its length. Three elements. Length error. The vectors do not have a matching number of elements. APL marks the operator in error. Dyadic ⍴, reshape to a specific length. Elements are repeated if necessary.

Catenation, joins two vectors.

```
1  2  3  1  2  6  6
      ρX,Y                  The result is seven elements
7                           long.
      +/X                   Plus-reduction, same as
9                           1+2+3+1+2.
      ⌈/X                   Maximum-reduction.  Any scalar
3                           operator allowed in reduction.
      ⍳0                    A vector with no elements.
                            It prints as a blank line.
      0ρ5                   Another way to express the
                            same value.  Same result.
      5≥3                   Logical relations produce 1
1                           for true and 0 for false.
      =/Y                   Equality reduction of  Y .
1                           Both elements of  Y  are equal.
      ≠/X                   Remember right-to-left rule,
0                           ≠/X ←→ 1≠(2≠(3≠(1≠2))) ←→ 0.
      ×/⍳0                  Reduction of null vector gives
1                           operator's identity value.
      ⌊/0ρ0                 Identity value of the minimum
1.701411835E38              operator is biggest number.
      ⌊8÷⍳5                 Monadic ⌊ is floor function
8  4  2  2  1               (greatest integer in).
      M←3 3ρ⍳5              The reshape operator used to
      M                     make a two-dimensional array.
                            A blank line precedes each
 1  2  3                    plane of an array.  Elements
 4  5  1                    are used in row-major order.
 2  3  4
      ρM                    The shape of an array is
3  3                        itself a vector.
      ρρM                   The shape of the shape is
2                           called the rank of the array.
      1 0 1/M               Compression selectively in-
                            cludes or omits columns.
 1  3                       First and third columns.
 4  1
 2  4
      0 1 1⌿M               Row compression, select the
                            last two rows of  M .
 4  5  1
 2  3  4
      ?10                   A random number between 1
6                           and 10.
      ?6ρ10                 Six such numbers.  Monadic ?
2  9  4  6  5  3            is a scalar operator.
      N←?3 3ρ10             A random 3 by 3 matrix.
      N

  1   2   8
  4  10   3
  2   5   4
      M×N                   Scalar operations extend
```

```
                                    element by element to matching
 1   4  24                          arrays of any rank, producing
16  50   3                          results of matching shape.
 4  15  16
        M+.×N                       Ordinary matrix multiplication
                                    or inner product.  Multiply
15  37  26                          elements and plus-reduce.
26  63  51
22  54  41
        M⌈.+N                       Any scalar operators can be
                                    used in inner products.  Add
 6  12   9                          elements and maximum-reduce.
 9  15  12
 7  13  10
        Y∘.-X                       Outer product, applies scalar
                                    operator between all pairs of
5   4   3   5   4                   elements, generating higher-
5   4   3   5   4                   rank result.
        M←3 2↑M                     Take the first three rows and
        M                           two columns of  M.

1   2
4   5
2   3
        +/M                         Reduction applies to the last
3   9   5                           coordinate of any array.
        ×/[1]M                      Unless another coordinate is
8   30                              explicitly specified.
        M[2;1]                      Indexing or subscripting to
4                                   extract an element from an
        M[2 3;1 1 2]                array.  The shape of an index-
                                    ed reference follows the shape
 4   4   5                          of the subscript list.
 2   2   3
        M[5;6]                      Index value out of range.  The
INDEX ERROR.                        interpreter marks the index
        M[5;6]                      operator.
         |
        )ORIGIN 0                   Change the numbering of coord-
WAS 1                               inates and indices to start at
        ⍳4                          zero.  The counting vectors
0   1   2   3                       now start at zero.
        +/[1]M                      Coordinate numbers change.
3   9   5
        M[2;0 1]                    So do index values.
2   3
        'HI THERE.'                 A character vector.
HI THERE.                           It prints without quotes.
        ⍴'HI THERE.'                How many characters?  Each
9         2 3,'UHOH'                character is one element.
DOMAIN ERROR.                       Characters and numbers cannot
        2 3,'UHOH'                  be mixed in the same value.
```

```
      ''                          Still another way to write a
                                  null vector.
      ⌊/''                        Null vectors are neither char-
1.701411835E38                    acter nor numeric.
      ⍟100                        Natural logarithm.
4.605170186
      +/÷!⍳!⌊⍟⍟⌊/''               A "numberless" numerical pro-
2.718281828                       gram to calculate  e  from its
      )DIGITS 19                  Taylor series.  Increase the
WAS 10                            precision with which results
      +/÷!⍳!⌊⍟⍟⌊/''               are displayed, and try it
2.7182818284590045235             again.
      *1                          Compare this with the built-in
2.7182818284590045235             value of  e.  Looks good.
      2-6×÷3                      Small inaccuracy due to inter-
2.168404344971008868E¯19          nal binary arithmetic.
      )DIGITS 10
WAS 19
      )ORIGIN 1
WAS 0
      'ABCDEF'[2 1 4 3 1 2]       Any array can be indexed, even
BADCAB                            a constant.
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ,. '
      A[8 9 29 20 8 5 18 5 28]
HI THERE.                         Characters encoded as numbers.
      A⍳'E'                       Dyadic ⍳ is least index of  E
5                                 in  A .
      M←A⍳'HERE WE ARE.'          When the right operand is an
      M                           array, each element is indexed
8  5  18  5  29  23  5  29  1  18  5  28       individually.
      A[⌽M]                       Reverse operator reverses the
.ERA EW EREH                      order of a vector.
      A[M←3 4⍴M]                  Reshape the vector into an
                                  array.  Assignment may be used
HERE                              within an expression.  Again,
 WE                               the shape of an indexed value
ARE.                              follows the index shape.
      A[⍉M]                       The transpose operator.  In-
                                  terchange the rows and columns
H A                               of  M .
EWR
REE
E .
      M←A⍳'DOUBLE TALK.'          Here the elements of  M  are
      A[,⍉(2,⍴M)⍴M]               repeated, transposed, and then
DDOOUUBBLLEE  TTAALLKK..          picked up in row-major order.
    HM M                          Sort  M .  Result is the vector
9  4  1  6  11  5  10  2  8  3  12  7     of indices that
    M[HM M]                               will order  M .
1  2  4  5  11  12  12  15  20  21  28  29    M , sorted.
    A[M[M M]]                     An indexed index, to produce
ABDEKLLOTU.                       alphabetical ordering.
```

```
      ' ⎕'[1+X∘.≥⍳⌈/X]           Indexing a constant to produce
                                 a histogram.  Also illustrates
⎕                                maximum-reduction, and an
⎕⎕                               outer product using the ≥
⎕⎕⎕                              operator.
⎕
⎕⎕
      10 10 10 10⊤983            Encode a value to its repre-
0  9  8  3                       sentation with a given radix.
      10⊥0 9 8 3                 Evaluate the representation.
983                              The value is recovered.
      2 5 2 5⊤17                 Mixed-radix representations
0  1  1  2                       are legal.
      ⌶20                        The time, in sixtieths of a
2661854                          second since midnight.
      24 60 60 60⊤⌶20            Encode as hours, minutes,
12  19  42  38                   seconds, and sixtieths.
      ':0123456789'[1+1 1 0 1 1\1+4↑99 10 6 10 3600⊤⌶20]
12:22                            HH:MM if you like.
      (7⍴5 2)⊤1735               Convert 1735 to the radix of
1  1  2  0  3  1  0              Roman numerals.
      (,((7⍴5 2)⊤1735)∘.≥⍳4)/,⍉4 7⍴'MDCLXVI'
MDCCXXXV                         Or all the way to Roman.
      ∇ROMAN                     Define a function named  ROMAN .
FUNCTION 'ROMAN' NOT FOUND.      One does not exist yet.
INPUT.                           The editor invites its crea-
R←ROMAN D                        tion.  Header line, one arg.
R←(,((7⍴5 2)⊤D)∘.≥⍳4)/,⍉4 7⍴'MDCLXVI'∇  Close definition.
      ROMAN 1735                 Call the function.
MDCCXXXV                         Same result.
      ∇ARABIC                    We need one to go the other
FUNCTION 'ARABIC' NOT FOUND.     way.
INPUT.
Z←ARABIC S                       Also one argument, result.
S←1 5 10 50 100 500 1000['IVXLCDM'⍳S]  Now we can do Roman
Z←+/S××S+1-1↓S,0∇                       arithmetic.
      ROMAN(ARABIC'MDCCXXXV')-ARABIC'CCCLXVIII'
MCCCLXVII
      ∇ACK                       Ackermann's function will il-
FUNCTION 'ACK' NOT FOUND.        lustrate conditional branching
INPUT.                           and recursion.  M ACK N ↔ N+1
Z←M ACK N                        if M=0; (M-1) ACK 1 if N=0;
→(×M)×3-1<Z←N+1                  else, (M-1) ACK (M ACK N-1).
Z←M ACK N-1                      Monadic × is signum, branch to
Z←(M-1) ACK Z∇                   0 (return) if  M  is zero, else
      2 ACK 3                    branch to line 2 or 3.
9                                Many recursive calls on  ACK .
      ∇ACK                       Make it more efficient.
EDIT.                            This time function is found.
N 2                              'Next' request, skip no-line
→(×M)×3-1<Z←N+1                  and header lines, print next.
D                                Delete this line.
P                                Try to print it again.
```

```
NO LINE.                              It is no longer there.
I →(M⌊5)⌽0 2 3 4 5,7-1<Z←N+1          Insert a replacement for it.
I →0,Z←2+N                            Insert some more lines, spec-
I →0,Z←3+2×N                          ial cases of small M.
P                                     Print the most recent line.
→0,Z←3+2×N
.                                     Enter input mode.  Same as
INPUT.                                repeated 'I' requests.
→0,Z←¯3+8×2PN                         A typing error, should be 2*N.
→0,Z←¯3+*/(N+3)ρ2
.                                     The error is noticed.  Back to
EDIT.                                 edit mode.  Use L  request to
L 8×2                                 locate the line to be fixed.
→0,Z←¯3+8×2PN                         Editor prints it.
C /P/*∇                               Change P to * and end edit.
→0,Z←¯3+8×2*N                         Editor prints changed line.
      ∇ACK P 999∇                     Now print the whole function.
NO LINE.                              'No line' because pointer al-
Z←M ACK N                             ways starts before the first
→(M⌊5) ⌽0 2 3 4 5,7-1<Z←N+1          line.  Branch to M  handler.
→0,Z←2+N                              Line 2, for M  equals 1.
→0,Z←3+2×N                            Line 3, for M  equals 2.
→0,Z←¯3+8×2*N                         Line 4, for M  equals 3.
→0,Z←¯3+*/(N+3)ρ2                     Line 5, for M  equals 4.
Z←M ACK N-1                           For M  above four, use recur-
Z←(M-1) ACK Z                         sive evaluation.
EOF.                                  End of function reached.
      2 ACK 3                         Try the new ACK .
9                                     Same result, more efficiently.
      )FNS                            Display the names of all de-
ACK     ARABIC  ROMAN                 fined functions.
      )ERASE ACK                      A variable or function can be
      2 ACK 3                         erased.
SYNTAX ERROR.                         An error report results now,
      2 ACK 3                         because ACK  is unknown.
        |
      ∇PAYMENT                        Mortgage payment example.
FUNCTION 'PAYMENT' NOT FOUND.
INPUT.
Z←PAYMENT;R;N;V                       R , N , and V  are local vars.
R←RATE÷1200                           RATE  (percent per year) isn't.
V←(1+R)*N←12×YEARS                    Number of monthly payments.
Z←PRIN×R×V÷V-1∇                       PRIN , YEARS  also not local.
      PRIN←22500                      Establish the parameters for a
      RATE←7                          $22500, 25-year mortgage at
      YEARS←25                        7 percent.
      PAYMENT                         What will the monthly princi-
159.0253194                           pal+interest payment be?
      RATE←7 7.5 8 8.5                Try a variety of rates.
      PAYMENT                         Scalar functions extend to
159.0253194  166.273015  173.6586494  181.1760938  arrays.
      )VARS                           Type names of all variables.
A      M      N      PRIN    RATE    X      Y
```

```
 YEARS
       N

  1    2    8
  4   10    3
  2    5    4
```

Local variables *R* , *N* , and *V* do not appear. This *N* is the global *N* that was created many lines back, unchanged by the usage of a local variable of the same name.

```
       ∇PAYMENT C 2 /MENT/MENT RATE∇
Z←PAYMENT RATE;R;N;V
```

Change *PAYMENT* so that *RATE* is now a parameter.

```
       PAYMENT 9 10
188.8191818  204.4576677
```

The scalar function still extends to arrays.

```
       RATE
7  7.5  8  8.5
```

The global variable *RATE* was not used, it is unchanged.

```
       YEARS←20 25 30
       PAYMENT 9 10
LENGTH ERROR.
PAYMENT[2] V←(1+R)*N←12×YEARS
                         |
```

Here a scalar function is asked to operate on arrays of differing lengths. An error report results.

```
       ∇MOO
FUNCTION 'MOO' NOT FOUND.
INPUT.
MOO;B;S;M
S←¯1+4?10
L: B←+/S=M←⎕
'BC'[1+B<⍳+/,S∘.=M]
→L×⍳B<4
'CONGRATULATIONS!'∇
```

A program to play the game of *MOO* . User tries to guess the secret four digits in a minimum of turns. Computer awards a 'bull' for each digit hit, a 'cow' for each digit in wrong place. Need four bulls to win.

```
       MOO
⎕:
       1 2 3 4
C
⎕:
       5 6 7 8
BC
⎕:
       6 7 8 0
CC
⎕:
       3 1 0 5
C
⎕:
       3 6 7 9
CCC
⎕:
       9 3 6 8
BBC
⎕:
       7 3 9 8
BBBB
CONGRATULATIONS!
```

Let's play a game.
Quad is input/output symbol, user must type his guess.
One digit right, but wrong place. Branch back to ⎕ .
Next move is assigned to *M* .
One digit in the right place, one in the wrong place.
Trade 0 for 5, mix them up.
0 and 5 both in or both out.

Try both in. Mix up 1 and 3.
0 and 5 both out, one of 1 or 3 in, two of 6, 7, 8.

Proves that 8 was the bull on the second guess. Still do not know whether 6 or 7 is in.
Shucks, it was 7 and not 6, and 9 is in the wrong place.
It has to be 7 3 9 8.

At last!

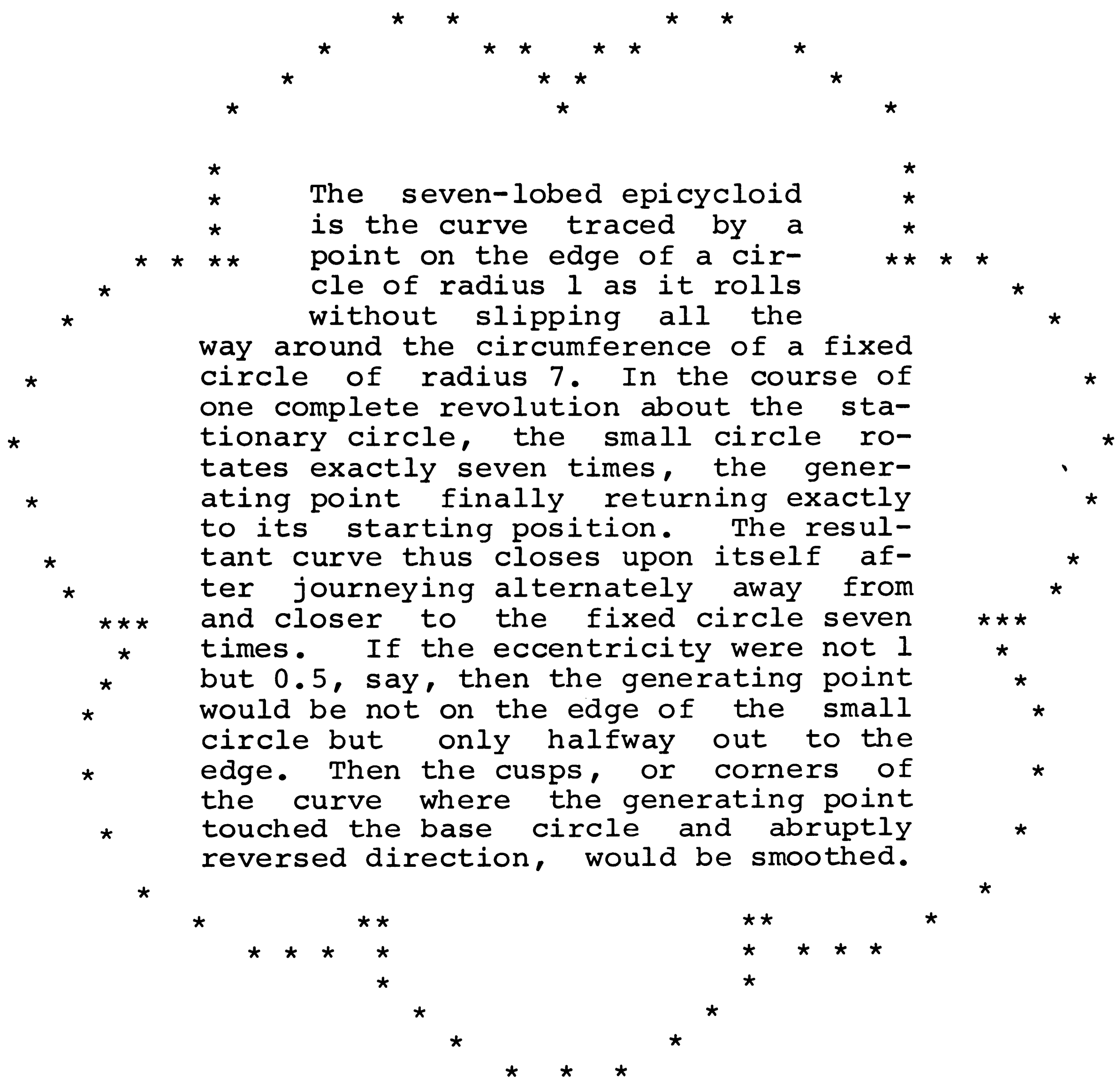
```
       MOO
⎕:
       S
BBBB
```

Another game, please.
This time we'll cheat. Any expression is legal input to the quad operator.

```
CONGRATULATIONS!
      MOO                    It would be more satisfying to
⎕:                           know the secret number.  Use
      ⎕←S                    the output form of quad to
6  1  0  3                   print it before returning it.
BBBB
CONGRATULATIONS!
      MOO                    Once more.
⎕:                           If a system command or a func-
      )ORIGIN 1              tion edit is initiated during
WAS 1                        a quad input, it is done and
⎕:                           then the quad is resumed.
      2+⍋1○⍳4                Any APL expression is legal.
CC
⎕:
      ⍳3                     Of course, not all expressions
LENGTH ERROR.                make sense to the  MOO  program.
MOO[2] L: B←+/S=M←⎕
                 |
      )SIV                   Display the state indicator
MOO[3]* B      M       S     and local variable names.
PAYMENT[3]*    Z       RATE     R       N       V
      M                      (Remember  PAYMENT  suspended.)
1  2  3                      Variables are accessible.
      →2                     Restart line 2 of  MOO  again.
⎕:                           Program executing again.
      ⍳4                     This is the right length.
B
⎕:
      →                      Exit from evaluated input.
      )SIV                   MOO  is removed from the state.
PAYMENT[3]*    Z       RATE     R       N       V
      →                      Another right arrow clears the
      )SIV                   state indicator completely.
      ∇GERMAN                An example of text manipula-
FUNCTION 'GERMAN' NOT FOUND. tion: simulated German accent.
INPUT.                       Translates specific pairs of
T←GERMAN S;I;J;M;N           characters ( M  array) into some
M←8 2⍴'WHTHQUJ∘V∘A∘Z∘W∘'     replacements ( N  array).  Enter
N←9 2⍴'V∘Z∘QVCHF∘AHTZV∘'     the arrays, '∘' means null.
S←S,T←'∘∘'                   Tack nulls onto input string.
A: I←⍴N[9;]←2↑S              Try to match two chars first.
B: →C×⍳11>I+J←(M∧.=N[9;])⍳1  Go to  C  if match found.
→B×I←⍴,N[9;2]←'∘'            No match, try single char now.
C: S←I↓S                     Match or no, advance input and
T←T,N[J;]                    copy or replace in output.
→A×⍳∨/'∘'≠S                  Loop while more input to do,
T←(T≠'∘')/T∇                 else remove nulls and exit.
      GERMAN 'PASS THE SALT, WILL YOU?'
PAHSS ZE SAHLT, VILL YOU?
      GERMAN 'THE FIVE WELDED JOINTS ARE QUIETER.'
ZE FIFE VELDED CHOINTS AHRE QVIETER.
      GERMAN 'THEY HAVE LESS SQUEAKS THAN THE OTHERS.'
```

```
ZEY HAHFE LESS SQVEAHKS ZAHN ZE OZERS.
      GERMAN 'WE USE THEM WHERE WE CAN.'
VE USE ZEM VERE VE CAHN.
      )LOAD ALGORITHMS              Load a previously saved work-
SAVED 03/23/72  1134.4              space.  System tells when it
      )FNS                          was saved.  What's in it?
CFC     CFX     EPI     GREG    JUL
      ∇CFX P 99∇                    Continued-fraction expander.
NO LINE.                            Continued fractions can be
V←CFX N                             used to find convergent ra-
→3,V←⌊N                             tional approximations to any
→3×20>ρV←V,⌊N←÷N                    real number.  This function
→2×0.0000000003≤|N←1|N              develops the first 20 terms in
EOF.                                the expansion of  N .
      CFX 6336÷3937                 Rationals have finite expan-
1  1  1  1  1  3  1  2  8  2  2  1     sions.
      ∇CFC P 99∇                    Continued-fraction compressor.
NO LINE.                            This algorithm prints the
CFC V;A;B;C;I;J                     successive rational approxi-
I←0                                 mations resulting from keeping
J←I←I+A←~B←0                        increasing numbers of terms in
A←B+V[J]×C←A                        the continued-fraction expan-
B←C                                 sion.
→(×J←J-1)ρ3
A;'÷';B;' = ';A÷B                   An example of mixed-type out-
→2×I<ρV                             put:  numbers and characters
EOF.                                combined on the same line.
      CFC CFX 6336÷3937             Kilometers per mile, exactly.
1÷1  = 1                            Successive approximations.
2÷1  = 2                            Two terms.
3÷2  = 1.5                          Three terms.
5÷3  = 1.666666667                  The successive approximations
8÷5  = 1.6                          are alternately smaller, then
29÷18  = 1.611111111                larger, than the limit value,
37÷23  = 1.608695652                but each is better than the
103÷64  = 1.609375                   previous.
861÷535  = 1.609345794
1825÷1134  = 1.609347443             The limit in this case is
4511÷2803  = 1.609347128             rational, so it is finally
6336÷3937  = 1.609347219             achieved exactly.
      )DIGITS 19                    Flex our muscles for a much
WAS 10                              harder one next.
      CFC 10↑CFX ○1                 The first ten rational approx-
3÷1  = 3                            mations to pi.
22÷7  = 3.142857142857142857        Every schoolboy knows  22÷7.
333÷106  = 3.141509433962264151
355÷113  = 3.141592920353982301          Have to work hard to
103993÷33102  = 3.141592653011902604     improve on  355÷113.
104348÷33215  = 3.141592653921421045
208341÷66317  = 3.141592653467436705     The error in  1146408
312689÷99532  = 3.141592653618936623     ÷364913 is only a
833719÷265381  = 3.141592653581077771    trillionth, though
1146408÷364913  = 3.141592653591403978   the divisor is less
```

```
      ○1                            than a million.
3.1415926535897932 38          True limit, to 19 decimals.
      ∇EPI P 99∇               Function to graph epi- (K>0)
NO LINE.                       and hypo- (K<0) cycloids (E=1)
Z←E EPI K                      and trochoids (E≠1) of order
E←E,1ρK←1+K,0                  |K and eccentricity E.
K←(⍉1 2∘.○K∘.×0.02×○⍳100)-.×E×30÷+/|E
Z←2257ρ' '
Z[(⌊31.5+K[1;])+61×⌊18.5+0.6×K[2;]]←100ρ'*'
Z←37 61ρZ
EOF.
      1 EPI 7                  Seven-lobed epicycloid.
```



The seven-lobed epicycloid is the curve traced by a point on the edge of a circle of radius 1 as it rolls without slipping all the way around the circumference of a fixed circle of radius 7. In the course of one complete revolution about the stationary circle, the small circle rotates exactly seven times, the generating point finally returning exactly to its starting position. The resultant curve thus closes upon itself after journeying alternately away from and closer to the fixed circle seven times. If the eccentricity were not 1 but 0.5, say, then the generating point would be not on the edge of the small circle but only halfway out to the edge. Then the cusps, or corners of the curve where the generating point touched the base circle and abruptly reversed direction, would be smoothed.

```
      ∇JUL P 99∇               Gregorian calendar to Julian
NO LINE.                       day number conversion. Julian
D←JUL DMY;M;Y                  day numbers are sequentially
```

```
D←DMY[1]                                 assigned--useful for studying
M←DMY[2]                                 periodic phenomena and theo-
Y←DMY[3]                                 retical calendar calculations.
Y←Y-9<M←12|M+9
D←1721119+D+(⌊0.5+30.6×M)+(365×Y)+(⌊Y÷4)+(⌊Y÷400)-⌊Y÷100
EOF.
      JUL 25 5 1942                      Julian day beginning noon UT
2430505                                  on Monday 25 May 1942.
      JUL 24 11 ¯4713                    Monday 24 November -4713 is
0                                        the origin of the numbers.
      JUL 7 9 ¯3760                      Beginning of Hebrew calendar,
347998                                   Monday 1 Tishri 1.
      JUL 7 9 1963                       Another date, Saturday 7 Sep-
2438280                                  tember 1963.
      (JUL 7 9 1963)-(JUL 25 5 1942)
7775                                     Days between two dates.
      ∇GREG P 99∇                        Now a function to go the hard
NO LINE.                                 way:  this one accepts a Jul-
D←GREG M;Y                               ian day number and returns the
M←M-1721120                              corresponding Gregorian date.
Y←400×(M-D←146097|M)÷146097              Get 400-year cycle.
M←3+4×D+3⌊⌊D÷36524                       Get 29 Feb in leap centuries.
Y←Y+(M-D←1461|M)÷1461                    Get 4-year period.
M←2+5×⌊D÷4                               Get month and also finally
Y←Y+3>M←1+12|2+(M-D←153|M)÷153             get the year itself.
D←1+⌊D÷5                                 Get day.
D←D,M,Y                                  That's all folks.
EOF.
      GREG 2430505                       This function undoes what
25  5  1942                               JUL  does.
      GREG JUL 29 12 1942                Any legal date comes through
29  12  1942                             the two of them unchanged.
      GREG JUL 29 2 1942                 But was 1942 a leap year?
1  3  1942                               No, there was no 29 Feb 1942.
      GREG JUL 28 2 1942                 Of course, there was a 28 Feb.
28  2  1942
      GREG JUL 29 2 1944                 1944 was a leap year.
29  2  1944
      GREG JUL 29 2 1900                 But 1900 wasn't.  Not all cen-
3  1  1900                               turies are leap in the Greg-
      GREG JUL 29 2 2000                 orian calendar.
29  2  2000                              Only the multiples of 400 are.
      GREG 700+JUL 7 9 1963              Calendar addition:  700 days
7  8  1965                               from 7 Sep 1963 is 7 Aug 1965.
      ∇DOW                               The day-of-the-week can be ob-
FUNCTION 'DOW' NOT FOUND.                tained easily from the Julian
INPUT.                                   day number since they are se-
DOW D                                    quentially assigned.
'MONTUEWEDTHUFRISATSUN'[1 2 3+3×7|JUL D]∇
      DOW 25 5 1942                      Verify the days-of-the-week
MON                                      claimed above.
      DOW 24 11 ¯4713
MON
```

```
      DOW 7 9 1963
SAT
```

Looks good.

```
      ∇INT0
FUNCTION 'INT0' NOT FOUND.
INPUT.
```

Some various numeric integration formulas.

```
Z←X INT0 Y
X←X+(⍳100)×Y←0.01×Y-X
'F(X)?'
Z←Y×+/⎕∇
```

Order 0 (rectangular) approximation. Integral from *X* to *Y* of function typed in as evaluated input.

```
      ∇INT1
FUNCTION 'INT1' NOT FOUND.
INPUT.
Z←X INT1 Y
X←X+(0,⍳100)×Y←0.01×Y-X
'F(X)?'
Z←(Y×+/⎕×1⌽1 1,99⍴2)÷2∇
```

Order 1 (trapezoidal) approximation. Integral is the sum of the areas of trapezoids fit to the function at 100 equally-spaced points.

```
      ∇INT2
FUNCTION 'INT2' NOT FOUND.
INPUT.
Z←X INT2 Y
X←X+(0,⍳100)×Y←0.01×Y-X
'F(X)?'
Z←(Y×+/⎕×1⌽1 1,99⍴4 2)÷3∇
```

Order 2 (parabolic) approximation (Simpson's Rule). Sum of the areas of 50 parabolas fit to the function.

```
      0 INT0 ○÷2
F(X)?
⎕:
      1○X
1.0078334198735818 76
```

Integral of sin x dx from 0 to pi/2. Analytic answer is 1. Program requests input of the function. Give it sin x.
0.8% error for order 0.

```
      0 INT1 ○÷2
F(X)?
⎕:
      1○X
0.9999794382396073936
```

Try the trapezoidal approximation with the same integral.

Order 1 error is 0.002%.

```
      0 INT2 ○÷2
F(X)?
⎕:
      1○X
1.0000000003382359 41
```

Now try Simpson's Rule.

Order 2 error is very small.

```
      0 INT0 ⍟3
F(X)?
⎕:
      (1+*X)*÷2
1.8389295089415000 88
```

Another function now. Try sqrt(1+exp x) dx from 0 to log 3. Indefinite integral is 2sqrt(1-exp x)-x+2log(sqrt(1+ exp x)-1), giving an analytic answer of 1.8357077606247.

Order 0 gives 0.2% error.

```
      0 INT1 ⍟3
F(X)?
⎕:
      (1+*X)*÷2
1.8357117480469399 05
```

Order 1 about 0.0002%.

```
      0 INT2 ⍟3
F(X)?
⎕:
      (1+*X)*÷2
1.8357077606302894 34
```

Good to 10 decimal digits.

```
      4-(8*÷2)+○3×(¯1+2*÷2)*2
1.8357077606247486 89         The analytic answer.
      3 EPI ¯3                A three-looped hypotrochoid.
      ** * * * * *                       * * * * * **
    *              *  *             *  *             *
   *                    *         *                    *
  **                      *     *                      **
  *                         * *                         *
  *                                                     *
   *                        * *                        *

     *                  *         *                  *
      * *              *           *              * *
          *                                     *
            *         *             *         *
               * *                       * *
                    ** *           * **
                    *     *  *  *     *

                   *                   *

                   *                   *
                   *                   *

                   *                   *
                    *                 *
                    *                 *

                     *               *
                      **          **
                         *       *
                          ** * **

      )OFF                    All done, time to go away.
r 0937 1.314 1.332 30         Multics ready message.
```

SECTION VII

COMPARISON WITH APL/360

APL/360

Multics APL was designed to appear to the user to be very nearly identical to APL/360. Therefore, users who are already familiar with the APL language through the APL/360 implementation may find the early sections of this document largely repetitious. Such users need a condensation of the material there, a summary which discusses only those aspects of Multics APL which are different from APL/360. This section provides that summary.

To be more precise, this chapter is a list of differences between Multics APL and the APL/360 language as defined by the IBM documentation, specifically the APL/360 User's Manual, IBM form number GH20-0683-1, January 1970.

Since Multics APL was designed, a more recent document describing APL/360 has become available. It is APL/360-OS and APL/360-DOS User's Manual IBM form number SH20-0906-0, December 1970. With the exception of an added Appendix C, SH20-0906-0 is identical to GH20-0683-1 mentioned above. The new features described in Appendix C are not present in Multics APL except as explicitly mentioned below. That is, the standard for comparison remains the GH20-0683-1 manual.

MULTICS APL

Multics APL is a user-ring command which provides to a single Multics process approximately the same capability as APL/360 provides to one of its users. APL/360 is, in contrast, a privileged subsystem that does its own multiplexing of users, independent of the system in which it is imbedded. The three major areas in which Multics APL is different from APL/360 are: 1) the STOP and TRACE control features for functions are not implemented; 2) the function editor behaves like Multics "edm" instead of like the APL/360 editor; and 3) ASCII terminals as well as Selectric-type terminals are supported. Beyond these three principal differences, the discrepancies between the languages are relatively minor.

Entering APL

To enter APL, issue the "apl" command with no arguments. APL will indent six spaces and listen for input. If your console is a Selectric-type terminal, change to the APL type ball. If you have an ASCII console, you will use an ASCII representation of the APL character set. Read Section II, which explains terminal input and output. Your workspace is initially clear. You may now type an APL system command (see "System Commands" in this section, or you may edit an APL function (see "Function Editing", also in this section). Exit from APL with the)*Q* or)*OFF* system commands.

APL Is Recursive

Multics APL is fully recursive. This means it can be called from within itself (for example, via the)*E* system command), or while a previous version of itself is being held in the stack, with no loss of data. Each invocation results in an independent workspace.

Workspace

A workspace in Multics APL is up to 16 segments in the user's process directory; i.e., about four million characters as opposed to APL/360's 32000 characters. The segments are named "apl.?", where "?" is a unique identifier. APL can be used recursively, so several workspaces can be in the process directory at once. Cleanup handlers dispose of these segments when APL is exited. There are APL system commands that permit saving a workspace as a normal Multics segment so that its contents can be retained from session to session.

Terminal I/O

While APL/360 usage is restricted to Selectric-type terminals, Multics APL is also usable (though not as conveniently) from Teletype Model 37, GE TermiNet 300s, Honeywell SRT301 and ARDS terminals. Read Section II of this manual.

Program Interrupt is Enabled

When APL is running, the program interrupt condition is always enabled. If the user issues a Multics "quit" followed by the "pi" command, APL temporarily turns off all output and gives the computation in progress a short grace time to complete. If it completes, APL suspends execution and accepts further input from the console as normal. To restart the suspended function, use the jump operator. If the computation in progress does not run to completion in the grace period, it is aborted; the state indicator is returned to the previous console suspension point (as if the jump operator with no operand were typed); and then APL suspends there again.

Thus, APL never suspends execution in the middle of a line, only at the end.

APL LANGUAGE ITSELF

The language interpreted by Multics APL is almost exactly the same language as that defined by the APL/360 documentation. To get acquainted with Multics APL, the user need not read any of the paragraphs in this section, as they all deal with relatively minor differences. Seasoned APL/360 users are cautioned to read "Order of Execution" following, however.

Order of Execution

As in APL/360, the syntax of Multics APL defines all operators as accepting everything to their right as their operand. Operators are executed in strictly right-to-left order.

However, the order of fetching of operands to make them available for operations is explicitly undefined both in APL/360 and in Multics APL (read carefully the section entitled, "Multiple Specification" on page 3.45 of the APL/360 User's Manual), and the user is warned that APL/360 and Multics APL do indeed differ in their order of operand fetch.

In general, the difference is evident only in statements containing, other than the left-most (i.e., last) operation, assignments that give values to variables used elsewhere in the same statement (or calls to functions that make such assignments). A statement that depends on a specific order of operand preparation is incorrect and cannot be expected to give consistent results in different implementations (or in later releases of APL/360 either, which perhaps is why the point perhaps is carefully mentioned in APL/360 documentation).

Line Length 256

The maximum length of any single line that can be handled by Multics APL is 256 characters. If a single line to interpret extends across many console lines (due to newline characters imbedded in character constants), the limit applies to the sum of the console lines.

Identifiers

An identifier (variable name, function name, group name) in Multics APL consists of an alphabetic character followed by any number of alphabetic or numerics. For the purposes of this definition, there are 53 alphabetic characters: the plain alphabet (A...Z), the alphabet underscored ($\underline{A}$...$\underline{Z}$), and the underscore character itself (_). There are 10 numerics: the digits (0...9).

Unlike APL/360 usage, the underscore is accepted as an alphabetic character, and the delta is not.

Minimum and Maximum Identity Elements

Due to the internal format of floating-point numbers, the identity elements for the minimum and maximum operations are +1.701411834604692317e38 and -1.701411834604692317e38, respectively.

Take and Drop Conformability

The left argument of the take and drop operations must be within the range specified by APL/360 documentation, or else a domain error results in Multics APL.

Coordinate for Compress, Expand, and Grade

In Multics APL, the compress, expand, and grade operators accept the number of a coordinate upon which to act, as specified in the APL/360 documentation.

I-Beam Functions

As might be expected, the system-dependent functions are defined somewhat differently for Multics. Their meanings are (where 60 "jiffies" equal 1 second):

19 Real time in jiffies since this instance of APL was invoked (uses "clock_").
20 The time of day in jiffies since midnight (uses "clock_" and "sys_info $time_delta").
21 CPU time in jiffies since this instance of APL was invoked (uses "hcs_$get_usage_values").
22 Size of workspace remaining available to be used, in units of 9-bit characters (i.e., four times the number of words).
23 Multics "nusers".
24 The time of day, in jiffies since midnight, that this instance of APL was invoked.
25 The date, as a 6-digit integer, MMDDYY.
26 The first element of I-beam 27.
27 The vector of statement numbers in the state indicator.

Note on I-beam 22: This number reflects the fact that a workspace can be up to 16 segments in size. However, since any single datum must completely fit within one segment, it is possible to get workspace-full errors even when I-beam 22 is returning large values. For example, it is impossible to create a 300,000-character item, even in a clear workspace with millions of characters of space available.

There is no dyadic I-beam in Multics APL. The library functions for workspace manipulation, such as *DIGITS* and *ORIGIN*, that are implemented via the dyadic I-beam in APL/360, are implemented differently in Multics APL (see "Library Function" in this section). The dyadic I-beam is not a documented user interface in APL/360.

Jump Operator

In APL/360, the jump operator → is accepted and ignored in many places in which it is meaningless. In general, Multics APL will treat such cases as syntax errors. For example, in response to a quad, only a right arrow alone is legal (exit from evaluated input), not a jump to a specified line number.

Escape from Character Input

To escape from character input in Multics APL, press the interrupt button and then issue the "pi" command. The state indicator will be reset to the most recent console suspension, as if a right arrow had been typed for evaluate input. See "Program Interrupt is Enabled" in this section.

The overstruck OUT character has no special significance in Multics APL since the interrupt button is always operational.

Error Messages

Multics APL error messages are almost identical to APL/360 error messages. Multics APL is a bit more careful to place the offender marker under the proper character.

Messages not in Multics APL

The following messages do not occur in Multics APL: *RESEND*, *NONCE*, *SI DAMAGE*, *SYMBOL TABLE FULL* (symbol table shares workspace with everything else; more symbols can always be added until the whole workspace is full), *LABELS UNSTUCK* (no warning is given when this occurs), *SYSTEM* (instead, a "panic" usually occurs; see "Panic", below).

Stream Switching on Errors

When an error occurs, Multics APL switches the streams it uses for input and output to "user_i/o". In this initial implementation, there is no easy way to restore the stream switching to its former state (a method is described under "Error Messages" in Section III). The attachments of "user_input" and "user_output" are not disturbed by this action; the implementation is such that they are not read after an error.

Panic

In the event of some disastrous internal error, Multics APL will type ...*APL PANIC*...*COMMAND LOOP CALLED*... If this happens, your APL session cannot be continued. A PANIC is generally indicative of an error in Multics APL, and should be brought to the attention of system maintenance personnel.

Library Functions

For programmably changing workspace parameters, the public workspace *APL>WSFNS* has within it definitions of the following library functions, which can be copied into any workspace with the *)COPY* command:

DIGITS	*FUZZ*	*SFCI*
ORIGIN	*SETLINK*	*SFEI*
WIDTH	*DELAY*	*SFII*
E		

Each of these monadic functions performs the same action as the similarly named system command, including returning as a result the old value of the altered workspace parameter. The *DELAY* function has no returned value; it causes execution to be suspended for the specified number of seconds (uses "timer_manager_$sleep"). Likewise, *E* has no returned value (be sure to read the caution about the use of *E* in this section).

In Multics APL, the library functions are implemented as external functions (see "External Functions" in Section IV), rather than by a dyadic I-beam operator as in APL/360. There is no dyadic I-beam in Multics APL.

The use of a library function involves initiating it with its name as a reference name, so the user must be aware of possible naming conflicts should he refer to an identically named segment elsewhere in his process. External function names are tagged with an asterisk in a *)FNS* listing.

Note that in Multics APL the fuzz is an arithmetic tolerance, rather than a number of bits of internal representation to be masked as in the latest version of APL/360 (APL/360-OS and APL/360-DOS User's Manual, IBM form number SH20-0906-0, December 1970, page C.1).

Matrix-Inverse (Domino) Generalized

The matrix-inverse operator of Multics APL is a slight generalization of the APL/360 operator. Let *X* be the result of *B⌹A*. Then for conformability, the shape of *A* must be of the form *(H,U)* where *U* is a single dimension and *H* is a (possibly null) vector of preceding dimensions; the shape of *B* must be of the form (*H*,*T*) where *H* is identical to the *H* in the shape of *A* and *T* is a (possibly null) vector of succeeding dimensions; and finally (×/*H*) must be not less than *U*. The result *X* has the shape (*U*,*T*) and it is (×/*T*) sets of least-squares best solutions in *U* unknowns to the (×/*T*) sets of (×/*T*) linear equations in *U* unknowns. That is, the elements of *X* are chosen to minimize `+/,((B+.×A)-X)*2`. If the matrix-inverse operator is used monadically, then *T* is taken as (×/*H*) and the elements of *B* are taken as those of the order (×/*H*) identity matrix; *X* will be a matrix of shape (*U*,(×/*H*)).

In the APL/360 matrix-inverse operator, *H* is required to be a single dimension, and *T* is required to be either a single dimension or else null.

System Commands

Multics APL has most of the system commands of APL/360. The commands that are available are:

)*Q*	)*FNS*	)*SETLINK*
)*QUIT*	)*VARS*	)*SFII*
)*ORIGIN*	)*GROUP*	)*SFEI*
)*WIDTH*	)*GRP*	)*SFCI*
)*DIGITS*	)*GRPS*	)*E*
)*SAVE*	)*OFF*	)*WSID*
)*LOAD*	)*CONTINUE*	)*LIB*
)*ERASE*	)*SI*	)*DFN*
)*CLEAR*	)*SIV*	)*MFN*
)*COPY*	)*PCOPY*	)*ZFN*
)*DROP*	)*PORTS*	

Most commands operate identically to those of APL/360. Only the differences are discussed below. In Multics APL, all characters in the system command name are significant, not the first four as in APL/360. As in APL/360, some commands are duplicated as library functions, so that they can be called from APL programs (see "Library Functions" in this section).

Use of)*QUIT* or)*OFF* to Exit from APL

These three commands are identical. Use)*Q*,)*QUIT*, or)*OFF* to cause APL to return to its caller. The active workspace in the process directory is destroyed when APL is left. If it is desired to retain a saved copy of the workspace contents, the)*CONTINUE* command should be used instead, or a)*SAVE* command should be issued before exiting. See "Saving and Reloading Workspaces" in this section for a discussion of saving and reloading workspaces.

)*DIGITS* Command Allows up to 19 Digits

Due to the internal representation of double-precision floating-point numbers, the)*DIGITS* command will allow up to 19 decimal digits of output to be requested.

)*SETLINK* Initializes Random Number Generator

The integer given in the)*SETLINK* command is used as the seed for the random number generator. It should be in the range 1 ≤ integer ≤ 34359738367. A clear workspace will have a seed derived from the real-time clock.

)*FUZZ* Sets the Fuzz

The number given in the)*FUZZ* command changes the quantity used by APL to tell when one floating-point number is sufficiently close to another that they can be taken as equal. The fuzz must be in the range 0 ≤ fuzz < 1. The numbers f and g will be considered equivalent if and only if abs(f-g) ≤ fuzz.

Setting the fuzz too wide can interfere with the credibility of results. In general, a careful error analysis should be done before tampering with the fuzz. The default fuzz is 1.0E-1.03.

Note that in Multics APL the fuzz is an arithmetic tolerance, rather than a number of bits of internal representation to be masked as in the latest version of APL/360 (APL/360-OS and APL/360-DOS User's Manual, IBM form number SH20-0906-0, December 1970, page C.1).

)*SFII*, *SFEI*, and *SFCI* Set the Go-Ahead Characters

When APL is ready to accept immediate input, it types 6 spaces as a "go-ahead" signal to the user, and then waits for input. When it is ready to accept evaluated input (in response to the quad operator), it types a quad, a colon, a newline, and then six spaces. When it is ready to accept character input (due to the quote-quad operator), it types nothing.

These go-ahead strings can be changed by the users. The *)SFII* (string for immediate input), *)SFEI* (string for evaluated input), and *)SFCI* (string for character input) commands each accept a character string (in quotes) of up to 16 characters to replace the corresponding default string.

)E Executes a Multics Command

The remainder of the line following the *)E* command is passed to "cu_$cp", the Multics command processor, for execution.

The user is cautioned that the APL terminal input processor has read and canonicalized the command line. Thus, the user must anticipate the character translations defined in Section II, and compensate for them when necessary. For example, if one types

```
)E SM MGS M WHY ERROR WHEN ¨COPY-¨ING FILE FROB?
```

the actual message transmitted, in ASCII, will be

```
Why error when ¢223py_¢222g file frob?
```

This precaution applies equally forcefully to the user of the *E* editor request and the *E* library function. Note, however, that APL does turn off its own input processing and does restore the Multics processing during the actual call on the command processor--thus, the executed commands have the normal Multics input/output environment available to them.

)ERASE Command

The *)ERASE* command works the same as in APL/360. As in APL/360, erasure of groups containing groups is not recursive: members of groups contained in an erased group will not be erased. Also, when a *COMMAND ERROR* occurs during an erase (garbled names supplied, for example), only the names up to the error marker have been processed.

)CLEAR Command

The *)CLEAR* command works the same as in APL/360. It erases all variables, functions, and groups, and resets the state indicator, workspace identification, password, index origin, digits, width, fuzz, seed, and go-ahead sequences. It does not affect the input/output stream attachments, nor does it affect the timers maintained by the I-beam functions.

No)*MSG*,)*MSGN*,)*OPR*,)*OPRN* Commands

Use)*E MAIL* or)*E SM*.

)*PORTS* Command is Multics "who"

The)*PORTS* command is implemented as a call on the Multics "who" command. Any arguments typed after)*PORTS* are passed on unchanged to the "who" command.

Saving and Reloading Workspaces

The workspace save and load capability is present in Multics APL. The commands which relate to this facility are:

)*SAVE*	)*CONTINUE*	)*LIB*
)*LOAD*	)*WSID*	)*DROP*
)*COPY*	)*PCOPY*	

WORKSPACE IDENTIFICATION

In APL/360 a workspace identification consists of a library number and a name; in Multics APL it consists of an absolute or relative pathname. A workspace can be saved as a segment (or, if necessary, a multisegment file) anywhere in the storage system hierarchy. All commands that accept a workspace identification expect on Multics an absolute or relative pathname. Normal Multics quota and access mechanisms govern the storage of saved workspaces. The suffix ".apl" is automatically appended to the workspace identification by the APL processor, so that APL workspaces are readily distinguishable in directory listings.

PASSWORDS

It is possible to associate a password with a saved workspace. Users should note, however, that nothing prevents another user from supplying a program of his own construction to search for interesting things in your saved workspaces.

The syntax of supplying passwords on Multics is slightly different from APL/360, although they are accepted in exactly the same places and treated identically. Whenever in APL/360 a command would be terminated with a colon followed by a password, in Multics the command should be terminated with a colon alone. APL types *PASSWORD*: and turns off the console printer. When the password is received, the printing will be restored. This procedure conforms to the handling of passwords elsewhere in Multics.

Passwords are accepted on the commands)*SAVE*,)*LOAD*,)*COPY*, and)*PCOPY*.

INTERNAL FORMAT OF SAVED WORKSPACES

The variables and functions stored in a saved workspace are encoded in a complicated fashion, subject to redefinition as modifications are made to the APL interpreter. It is not recommended that users attempt to access saved workspaces via means other than the APL command itself.

)*LIB* AND)*DROP* COMMANDS

Since workspaces are saved as normal Multics files instead of in special APL libraries, the)*LIB* and)*DROP* commands are implemented simply as calls on the Multics commands "list" and "deleteforce". An argument of "*.apl" is passed to the "list" command to list only APL workspaces, and the suffix ".apl" is automatically appended to the names listed in the)*DROP* command.

NO APL LIBRARY

No library of APL applications is available on Multics. The only library functions available at the present time are the workspace functions:

DIGITS	*FUZZ*	*SFCI*
ORIGIN	*SETLINK*	*SFEI*
WIDTH	*DELAY*	*SFII*
E		

Their usage is described under "Library Functions" in this section.

External Functions

The Multics APL interpreter permits APL programs to make external calls to object segments that have been created by other Multics translators, such as PL/I, provided that those object segments obey a specified interface. To the APL program, such a call looks like an ordinary reference to a defined function; the function may accept zero, one, or two arguments, and it may optionally return a result. External functions are discussed under "External Functions" in Section IV.

FUNCTION EDITING

The mechanics of Multics APL editing are different from those of the APL/360 editor. The Multics APL editor is patterned after the Multics "edm" command. The reader is advised to read Section V of this manual in its entirety. However, a synopsis of the differences is provided in this section. The remainder of this section presumes that the reader is familiar with "edm" (see the Multics Programmer's Manual, Commands and Active Functions, Order No. AG92).

In APL/360, the source text as typed by the user is discarded, and only an internal representation of the syntactic entities is kept for future editing (the APL user will note that the free-format spacing he used was discarded by the system). In Multics APL, the actual source text is retained as well. This means that subsequent edits of a function will display the original source text, unmodified in any way from the way the user last edited it.

When a Function Can Be Edited

The rules are the same as for APL/360.

The editor can be called whenever APL is accepting either immediate input or evaluated input. At the conclusion of the edit, APL again requests the input it was listening for.

A particular function can be edited only if it is not pendent. It may be suspended or inactive.

The definition error response to an editor call can be due to: 1) the function name has an illegal character; 2) a global variable or group already exists with the proposed function name; 3) the function is pendent; or 4) the function is external (external function names are tagged with an asterisk in a *)FNS* listing; see "External Functions" in Section IV).

Invoking and Terminating the Editor

The editor is invoked by typing a nabla character ∇ followed by the proposed function name. If the function already exists, the editor will respond *EDIT* and you may type an editor request (see "Edit Mode" below). If the function does not exist, the editor will respond *FUNCTION '...' NOT FOUND. INPUT* and you may input the function (see "Input Mode" below).

To exit from the editor, issue either the *Q* request, or type another ∇ at the end of any line (or on a line by itself). The ∇ character terminates the edit, whether typed in edit or in input mode.

When the user requests an exit from the editor, the source text amended by the edit session is scanned. The new function must have a valid header line (see "Function Header Line" following). If the function now has a different name, the new name is checked for validity. All changed lines are analyzed into syntactic entities. If any errors are detected in this scan, they are reported to the user, and APL will return to edit mode. The system will leave the editor only when the scan is successful.

It is not necessary to issue the *W* request before terminating an edit. An edit always reworks the current workspace copy of a function upon termination. The *W* request is for writing a copy of the APL source text for a function into the user's Multics working directory (or any other directory, for that matter), as discussed in " *M*,*U* and *W* Requests to Read and Write Files", below, but not into the APL workspace.

Function Header Line

The very first line of a function must be its header line. The syntax of a header line is the same as in APL/360 (result and left arrow if any, left argument if any, function name, right argument if any, and a list of local variable names set off by semicolons). A function must contain a valid header line in order to conclude the edit.

Note that the header line is not typed in the editor call line. Only the function name is typed there. If a new function is being created, the user must type the header line as the first line of his input, after the editor types *INPUT*.

The header line may be edited as any other line. It is not necessary that the header line be correct or even present until the conclusion of the edit.

Input Mode

When the editor responds *INPUT*, all following lines typed by the user are inserted into the function. A ∇ typed at the end of any line (or on a line by itself) will terminate the edit session. A line consisting of a period only returns the user to edit mode.

Edit Mode

When the editor responds *EDIT*, any editing request may be typed (see the "edm" write-up). If a ∇ character follows any request, the editor terminates after performing that request. The "." request returns to input mode.

Program Interrupt during Editing

During an edit, a Multics quit followed by the "pi" command returns the editor to edit mode. This can be used to abort lengthy output. When the editor exits, the normal APL interrupt action is restored (see "Program Interrupt is Enabled" in this section).

E Request Executes a Multics Command

The *E* and *E̲* requests pass the remainder of the request line (minus a trailing ∇ if present) to "cu_$cp" for execution. The edit terminates when the command processor returns if a ∇ followed the request. Be sure to read the precautions discussed under the *E* system commands in ")*E* Executes a Multics Command" in this section.

M, *U* and *W* Requests to Read and Write Files

The *M*, *U*, and *W* requests read and write Multics files in the user's working directory (or any other directory if path names are given). These requests do not write in the APL workspace. Note that the current workspace copy of a function is always amended upon termination of an edit. The *W* request should not be issued for this purpose.

When files are read and written, no canonicalization or other character conversion is performed. Thus, files suitable to be read by the APL editor *M* request must already contain valid APL character codes. The most convenient way to generate such files is with the APL editor itself, using the *U* and *W* requests.

The *M*, *U*, and *W* requests require a file name as an argument. There is no default file name built into the APL editor.

Saving Individual Function Definitions

The *M*, *U*, and *W* requests provide a way of communicating individual function definitions in source language to and from the Multics file system by making them accessible to other Multics programs (and via the punched-card daemon and a user-written conversion and formatting program, to any computer).

Sneak Request

As in APL/360, it is possible to "sneak" the user edit request onto the editor call line. Following the function name, one may type an editor request. If the function already exists, *EDIT* will not be typed. Instead the request will be executed. The editor then goes to edit mode (unless the request changed the mode or ended with the ∇ character). If the function does not exist, it is assumed that the user should be warned, so the editor will ignore the request and respond in input mode, as usual.

Line Numbers and Labels

Line numbers typed by the editor (in response to the = request) are source line numbers, just as for "edm". They do not correspond to the interpreter line numbers printed in error messages or to which the jump operator (right arrow) refers. These numbers differ because of the header line, which is considered to be interpreter line number zero; and because of newlines in character constants, which can cause two or more source lines to be taken as a single interpreter line.

For this reason, it is recommended that labels be used instead of absolute line numbers in jump operations.

Labels are automatically local variables, and they should not be declared in duplicate in the header line.

No System Commands in Editor

When an edit is in progress, no system commands are accepted by APL.

No Stop or Trace Control

Stop and trace control are not implemented.

No Locked Functions

Locked functions are not implemented. The closest approximation in Multics APL to a locked function is the external function capability. If an externally-coded function is supplied, APL cannot be used to print its definition. Note, however, that if a user knows the pathname of an object segment, there are many dumping and debugging tools available to him for studying it.

External Functions Cannot be Edited

Functions defined as external by the)*DFN*,)*MFN*, and)*ZFN* system commands (see "Communicating with Multics" in Section IV) cannot be edited by the APL editor. They can be erased or redefined. External function names are tagged with an asterisk in a)*FNS* listing.

PITFALLS

This subsection discusses some features of Multics APL that might be more troublesome than others, or that may produce apparently unrelated errors. These points are all covered elsewhere in this document, but they are collected here as a possible service to the new user.

Interrupt Button Is Not for Line Editing

Do not backspace and press the interrupt button to correct typing errors, as is done on APL/360. On Multics, this results in a quit and in loss of the typed line. Instead, use the erase and kill characters defined for your console. See Section II.

APL Translates *)E* Command, *E* Function, and *E* Request

Remember that the APL character code is not identical to ASCII, although there is much overlap. The command line passed to the Multics command processor by the *)E* system command, the *E* library function, and the *E* editor request is in APL internal character code; therefore, the user must anticipate differences from ASCII and compensate for them when necessary. See Section II and " *)E* Executes a Multics Command" in this section.

Remember the Function Header Line

When creating a function, remember that the first line must be the function header line. The header line is not typed in the editor call line, as in APL/360, but instead is typed with the remainder of the function. See "Function Header Line" in this section.

Do Not Use the *W* Request When Editing Is Finished

The *W* request is not intended to update the current workspace copy of a function at the conclusion of an edit; this is automatic. The *W* request is used to write Multics files. Typing the *W* request with no file name results in an error message from the APL editor. See " *M*, *U* and *W* Requests to Read and Write Files" in this section.

Order of Operand Preparation Differs from APL/360

While the operations of Multics APL are executed in a right-to-left order, the order of fetching of operands is undefined. Programs that are sensitive to the order of operand preparations are incorrect APL and can be expected to give different results. See "Order of Execution" in this section.

INDEX

* The abbreviation ff after a page number means "and the following pages".

HONEYWELL INFORMATION SYSTEMS

Publications Remarks Form*

TITLE: MULTICS APL USER'S GUIDE

ORDER No.: AK95, REV. 0

DATED: JANUARY 1974

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME ______________________ DATE: ______________

COMPANY ______________________

TITLE ______________________

____

____

*Your comments will be promptly investigated by appropriate technical personnel, action will be taken as required, and you will receive a written reply. If you do not require a written reply, please check here. ☐

LONG LINE

CUT ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MASS. 02181

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

POSTAGE WILL BE PAID BY:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATTN: PUBLICATIONS, MS 050

Honeywell