GC26-3847-0
File No. S370-22

Systems

APL Language

IBM

PREFACE

This manual is intended to provide a reference for the APL language, and also an introduction to APL to those unfamiliar with it. New users should read the introduction, which provides an overview of some fundamental concepts, and illustrates several kinds of use.

Section 2 presents the fundamentals and introduces the terminology used in later sections, but the sections may otherwise be read relatively independently. For example, a glance at the listing of the primitive functions in Figures 3.1 and 3.5 should suffice for the reader who wishes to concentrate just on the means of defining and using functions (programs) presented in Sections 6 and 7.

Readers familiar with earlier versions of APL but not with the notions of shared variables, will find this topic and the related topics of system functions and system variables covered in Sections 4 and 5.

The following publications are intended to supplement the information in this publication for particular APL systems:

APL Shared Variables (APLSV) User's Guide, Order No. SH20-1460

APL/CMS User's Guide, Order No. SC20-1846

First Edition (March 1975)

This publication serves two different APL systems: APL Shared Variables (Program Number 5799-AJF -- Programming RPQ WE1191) and APL/CMS (Program Number 5799-ALK -- Programming RPQ MF2608).

Changes may from time to time be made to the specifications herein; any such changes will be reported in subsequent revisions or Technical Newsletters. Before using this publication, consult the latest IBM System/360 and System/370 Bibliography, Order No. GA22-6822, for the editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative, or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to

IBM Corporation
Systems Development Division
LDF Publishing -- Department J04
1501 California Avenue
Palo Alto, CA 94304.

SECTION 1: INTRODUCTION

APL is a general-purpose language which enjoys extensive use in such diverse applications as commercial data processing, system design, mathematical and scientific computation, and the teaching of mathematics and other subjects. It has proved particularly useful in data-base applications, where its computational power and communication facilities combine to enhance the productivity of both application programmers and end users.

When implemented as a computing system, APL is used from a typewriter-like keyboard. Statements that specify the work to be done are entered by typing them, and in response the computer displays the result of the computation. The result appears at a device which accompanies the keyboard, such as a printer, or video display. In addition to work purely at the keyboard and its associated display, entries may also invoke the use of printers, disk-files, tapes or other remote devices.

TWO EXAMPLES OF THE USE OF APL

A statement entered at the keyboard may contain numbers, or symbols such as + - × ÷, or names formed from letters of the alphabet. The numbers and special symbols stand for the primitive objects and functions of APL; primitive in the sense that their meanings are permanently fixed, and therefore understood by the APL system without further definition. A name, however, has no significance until a meaning has been assigned to it.

Names are used for two major categories of objects. There are names for collections of data, made up of numbers or characters. Such a named collection is called a variable. Names may also be used for programs made up of sequences of APL statements. Such programs are called defined functions. Once they have been established, names of variables and defined functions can be used in statements by themselves or in combination with the primitive functions and objects.

AN ISOLATED CALCULATION

If the work to be done can be adequately specified simply by keying a statement made up of numbers and symbols, names will not be required: entering the expression to be evaluated causes the result to be displayed. For example, suppose it is required to compare the rates of return on money at a fixed interest rate but with different compounding intervals. For 1000 units at 6%, compounded annually, quarterly, monthly, or daily, for 10 years, the transaction (at a typewriter terminal) would look like this:

```
      1000×(1+.06÷1 4 12 365)*10×1 4 12 365      (entry)
1790.85 1814.02 1819.4 1822.03                   (response)
```

(The largest gain is apparently obtained in going from annually to quarterly, and after that the differences are relatively insignificant.)

Several distinctive features of APL are illustrated in this example: familiar symbols such as + × ÷ are used where possible; symbols are introduced where necessary, (as the * for the power function); and a group of numbers can be worked on together.

A PREPARED WORKSPACE

Although there are many problems that can be solved by keying in the appropriate numbers and symbols, the greatest benefits of using APL come when named functions and data are used. Since a single name may refer to a large array of data, using the name is far simpler than keying in all of its members. Similarly, a defined function, invoked by entering its name, may be composed of many individual APL statements that would be burdensome to type individually again and again.

Once a function has been defined, or data collected under a name, it is usually desirable to retain the significance of the names for some period of time; perhaps for just a few minutes, but more often for much longer periods, possibly months or years. For this reason APL systems are organized around the concept of a workspace, which might be thought of as a notebook in which all the different items needed during some piece of work are recorded together.

The following example illustrates the use of an application package - a workspace prepared for a particular data-processing application. It is not expected that the reader new to APL will immediately follow all details, but the example should nevertheless provide, on the one hand, an indication of the ease of exploring and using such a package, and on the other hand, the relative ease of constructing it.

The example is a skeletal system for sales, billing, and inventory control for, say, a wholesale distributor of gadgets. The system maintains an inventory of items in stock, with part numbers, descriptions, unit prices, quantities on hand, and reorder levels. It automatically adjusts the quantity on hand when an invoice is prepared, and signals when the stock level goes below the reorder level. It also provides for entering new stock items in the inventory list and has provision for order entry and printing of invoices.

To complete this basic system for actual application, it would be necessary to add functions such as the recording of a journal and the maintenance of the customer list, and to add appropriate checks on errors of input and on security. Nevertheless, the package does work, as far as it goes, and in addition to showing the use of many functions of the language, also demonstrates the conciseness of APL programming.

To start, the stored workspace containing the application package is activated, using a system command:

```
      )LOAD 1 SBIC
SAVED 18.42.25 02/06/75
```

To become acquainted with the contents of the package the names of variables and defined functions are listed:

```
      )VARS
CUSTLIST        DESCRIBE        INVNO   LOW     MTH     ORDLIST
STOCKLIST
      )FNS
ADDRESS DATE    GET     INVOICE NEWSTOCK        ORDER   ORDERENTRY
PRINT   PUT     RESTOCK
```

The variable name *DESCRIBE* is suggestive. Its value can be displayed by simply entering the name:

```
      DESCRIBE
THIS WORKSPACE CONTAINS FUNCTIONS FOR RECORDING ORDERS,
MAINTAINING AN INVENTORY, AND PREPARING INVOICES.  IT IS A
SKELETAL SYSTEM DESIGNED TO ILLUSTRATE THE USE OF APL IN
COMMERCIAL DATA PROCESSING.

THE VARIABLES  CUSTLIST,  ORDLIST, AND  STOCKLIST  CARRY,
RESPECTIVELY, CUSTOMER NAMES AND ADDRESSES KEYED TO A
CUSTOMER NUMBER, ORDERS BY STOCK NUMBER AND CUSTOMER NUMBER,
AND THE INVENTORY LIST OF STOCK ITEMS.  INVNO  IS A COUNTER
FOR SUCCESSIVE INVOICE NUMBERS,  LOW  INDICATES ITEMS TO BE
REORDERED, AND  MTH  HOLDS NAMES FOR MONTHS OF THE YEAR.

WHEN DISPLAYING THE STOCK LIST OR CUSTOMER LIST ON A
TYPEWRITER TERMINAL IT WILL BE DESIRABLE TO CHANGE TO A
TYPE-ELEMENT HAVING BOTH UPPER AND LOWER CASE ALPHABETS.

MEANINGFUL STATEMENTS IN THIS WORKSPACE ARE:

      NEWSTOCK
      ORDERENTRY
      PRINT INVOICE ORDER N
      RESTOCK

WHERE N IS A CUSTOMER NUMBER.  FOR CONSISTENCY WITH THE DATA
ALREADY ENTERED,  NEWSTOCK  AND  PRINT  SHOULD ALSO BE USED
WITH A TYPE-ELEMENT THAT HAS BOTH ALPHABETS.

THE FUNCTIONS  ADDRESS,  DATE,  GET, AND  PUT  ARE NOT
NORMALLY INVOKED BY A DIRECT KEYBOARD ENTRY, BUT ARE USED BY
THE VARIOUS FUNCTIONS NOTED ABOVE.
```

It might be helpful at this point to display *STOCKLIST* to see just what the information looks like. But the list might be very large, and it is good practice to find out its size before asking for the display:

```
      ρSTOCKLIST
6 39
```

The list has 6 rows and 39 columns, which is not too much to display in its entirety. Following the advice in *DESCRIBE* the type-element can be changed and the variable name entered:

```
      stocklist
1135 First Great Item      9.95 350  55
9993 High Flyer Widget    88.73 240  35
3569 Second Moneymaker    24.75 200  30
5613 Mail Order Special   14.99 600  95
2583 A Real Winner        49.99 125  10
9998 Nonesuch Frammis      2.69 500  50
```

Next it might be useful to look at the customer list:

```
      ρCUSTLIST
3 5 20
```

The customer list is apparently represented by a three-dimensional array, and it will be interesting to see how it is displayed:

```
      custlist
     7
City Traders Inc.
41 Postage Road
Rimela, N. Y. 12345


    55
Mail House Ltd.
7-11 Ramblers Lane
Offshore Island
City, S. Dak. 54321

   312
Mantup Sales Corp.
Ruralia Farms
RFD 2
Suburbian, Wis. 0000
```

Finally, it is probably worth looking at the order list:

```
      ⍴ORDLIST
6 3

      ORDLIST
  55 3569    5
 312 9998   12
   7 1135    2
 312 3569   10
 312 5613   45
   7 5613   75
```

This is a 3-column numerical array, and the significance of each column is easily deduced from the previous displays: the sucessive columns appear to represent customer number, stock number, and quantity.

Use of the package can be demonstrated by trying the meaningful statements:

```
      ORDERENTRY
CUSTOMER  ITEM  QUANTITY
```

In response to the statement *ORDERENTRY* the system has displayed column headings, and the keyboard has unlocked for further entry. These can be supplied, using some of the existing customer and item numbers. The entire transaction at the terminal might be this:

```
      ORDERENTRY
CUSTOMER  ITEM  QUANTITY
55 1135 12
312 1135 24
55 2583 4
```

Since customer 55 had one item on order before, three should be on order now.

Although it is not listed as a meaningful statement, what happens if the function *ORDER* is used by itself?

```
   ORDER 55
55 3569    5
55 1135   12
55 2583    4
```

This entry produces a result which seems to confirm the number of items on order. It turns out that this result will be used by *INVOICE* to produce a result which will in turn be used by *PRINT*.

```
      invoice order 55
 1336                                      55
3569 Second Moneymaker      24.75    5  123.75
1135 First Great Item        9.95   12  119.40
2583 A Real Winner          49.99    4  199.96
```

If the complete statement given in *DESCRIBE* is entered rather then these piecemeal entries, the result is a formatted output intended to be used with a pre-printed form:

```
print invoice order 55
```

To:

Mail House Ltd.
7-11 Ramblers Lane
Offshore Island
City, S. Dak. 54321

Invoice No. 1336

Feb 8, 1975

Stock	Description	Price	Quan.	Exten.
3569	Second Moneymaker	24.75	5	123.75
1135	First Great Item	9.95	12	119.40
2583	A Real Winner	49.99	4	199.96

Total $ 443.11

Distributors Corp. 123 First Ave., City, St. 99999

For the sake of completeness, and because they illustrate variations in the method of requesting input, the remaining two statements can be entered. *NEWSTOCK* prints a prompting message and then unlocks the keyboard, waiting for entry of the response on the same line:

```
          newstock
stock number/ 8760
description/ The Best Gadget
unit price/ 1.73
initial inventory/ 875
reorder level/ 100
```

RESTOCK not only prints a prompting message, but uses an additional prompt □: supplied by the system:

```
          RESTOCK
STOCK NUMBER
□:
      1135
STOCK INCREMENT
□:
      30
```

It might now be of interest to display the inventory once more to see the effect of the previous operations:

```
                stocklist
1135 First Great Item       9.95 368  55
9993 High Flyer Widget     88.73 240  35
3569 Second Moneymaker     24.75 195  30
5613 Mail Order Special    14.99 600  95
2583 A Real Winner         49.99 121  10
9998 Nonesuch Frammis       2.69 500  50
8760 The Best Gadget        1.73 875 100
```

The quantities on hand do indeed show the effect of completing the order, replenishing the stock, and adding a new item.

The complete set of ten functions in this package is displayed in Figure 1.1. The first line in each definition is the function header, which specifies, among other things, the function name and syntax. The remainder in each individual display is the function body. The longest function, *INVOICE*, has twelve lines, the shortest, *ORDER*, has only one.

```
FORM←PRINT N;T;X
X←1↑⍴N
FORM←¯4⍕ 12 43 ↑ADDRESS 1 ¯6 ↑N
FORM[1;37+⍳6]←6↑N[1;]
FORM[3;31+⍳12]←DATE ⎕TS
FORM[6+⍳X-1;]← 1 0 ↓N
FORM[12;]←¯43↑⍕+/⍎, 1 35 ↓N


Z←INVOICE N;R;X;PQR;EXT
X← 6 0 ⍕INVNO←INVNO+1
Z← 1 43⍴⍴(X, 37 0 ⍕N[1;1])
L1:X←N[1;]
PQR←⍎25↓R←GET X[2]
EXT←X[3]×PQR[1]
Z←Z,[1](31↑R), 4 0 8 2 ⍕X[3],EXT
PQR[2]←PQR[2]-X[3]
→(>/PQR[2 3])/L2
LOW←LOW,(~X[2]∈LOW)/X[2]
L2:R[31+⍳4]← 4 0 ⍕PQR[2]
R PUT X[2]
→(0≠1↑⍴N← 1 0 ↓N)/L1


Z←ORDER N
Z←(ORDLIST[;1]=N)⌿ORDLIST


Z←ADDRESS N;I
I←(CUSTLIST[;1;⍳6]∧.=,N)⍳1
Z← 1 0 ↓CUSTLIST[I;;]


Z←DATE N
Z←MTH[N[2];], 3 0 6 0 ⍕N[3 1]
Z[7]←','
```

```
NEWSTOCK;R
⎕←'STOCK NUMBER/ '
R← 4 0 ⍕⍎⎕
⎕←'DESCRIPTION/ '
R←R,21↑12↓⎕
⎕←'UNIT PRICE/ '
R←R, 6 2 ⍕⍎⎕
⎕←'INITIAL INVENTORY/ '
R←R, 4 0 ⍕⍎⎕
⎕←'REORDER LEVEL/ '
R←R, 4 0 ⍕⍎⎕
R PUT⍎4↑R


ORDERENTRY;L
'CUSTOMER  ITEM  QUANTITY'
L1:→(' '∧.=L←⎕)/0
ORDLIST←ORDLIST,[1]⍎L
→L1


RESTOCK;R;X;PQR
'STOCK NUMBER'
PQR←⍎25↓R←GET X←⎕
'STOCK INCREMENT'
R[31+⍳4]← 4 0 ⍕PQR[2]←PQR[2]+⎕
R PUT X


R PUT N;I
I←(STOCKLIST[;⍳4]∧.= 4 0 ⍕N)⍳1
→(I≤1↑⍴STOCKLIST)/L1
STOCKLIST←STOCKLIST,[1] ' '
L1:STOCKLIST[I;]←R


Z←GET N;I
I←(STOCKLIST[;⍳4]∧.= 4 0 ⍕N)⍳1
Z←STOCKLIST[I;]
```

Figure 1.1: Functions for Sales, Billing, and Inventory Control

The details of the programming will not be discussed here, but one point is worth noting. As they stand here the various data lists are very small. In a real use, one or more of them might have many thousands of entries. A good way to handle them in that case is illustrated by the treatment of *STOCKLIST*, which is accessed only by the functions *PUT* and *GET*, rather than the primitive APL functions which could have been used directly for this purpose. By thus isolating the access functions the application package design is automatically made more general; if a different representation were chosen for *STOCKLIST* - say, an external file- the required changes in the programming would be confined to the definitions of *PUT* and *GET*, and the other functions such as *INVOICE* or *NEWSTOCK*, which give the package its particular character, would be unaffected. In this way, an APL application can be expanded to data-bases of arbitrary size, well beyond the storage capacity of the workspace that establishes the environment for the work.

THE CHARACTERISTICS OF APL

The subsequent sections of this manual describe APL in detail, giving the meaning of each symbol and discussing the various features common to the APL systems currently available from IBM. It is well to view these details in light of the major characteristics of APL, which may be summarized as follows:

The primitive objects of the language are arrays (lists, tables, lists of tables, etc.). For example, $A+B$ is meaningful for any arrays A and B, the size of an array (ρA) is a primitive function, and arrays may be indexed by arrays, as in $A[3\ 1\ 4\ 2]$

The syntax is simple: there are only three statement types (name assignment, branch, or neither), there is no function precedence hierarchy, functions have either one, two, or no arguments, and primitive functions and defined functions (programs) are treated alike.

The semantic rules are few: the definitions of primitive functions are independent of the representations of data to which they apply, all scalar functions are extended to other arrays in the same way (that is, item-by-item), and primitive functions have no hidden effects (so-called side-effects).

The sequence control is simple: one statement type embraces all types of branches (conditional, unconditional, computed, etc.), and the termination of the execution of any function always returns control to the point of use.

External communication is established by means of variables which are shared between APL and other systems or subsystems. These shared variables are treated both syntactically and semantically like other variables. A subclass of shared variables, system variables, provides convenient communication between APL programs and their environment.

The utility of the primitive functions is vastly enhanced by operators which modify their behavior in a systematic manner. For example, reduction (denoted by /) modifies a function to apply over all elements of a list, as in $+/L$ for summation of the items of L. The remaining operators are scan (running totals, running maxima, etc.), the axis operator which, for example, allows reduction and scan to be applied over a specified axis (rows or columns) of a table, the outer product, which produces tables of values as in *RATE*$\circ.*$*YEARS* for an interest table, and the inner product, a simple generalization of matrix product which is exceedingly useful in data processing and other non-mathematical applications.

The number of primitive functions is small enough that each is represented by a single easily-read and easily-written symbol, yet the set of primitives embraces operations from simple addition to grading (sorting) and formatting. The complete set can be classified as follows:

Arithmetic: + - × ÷ * ⍟ ○ | ⌊ ⌈ ! ⌹
Boolean and Relational: ∨ ∧ ⍱ ⍲ ~ < ≤ = ≥ > ≠
Selection and Structural: / \ ⌿ ⍀ [;] ↑ ↓ ⍴ , ⍋ ⍒ ⊖
General: ∈ ⍳ ? ⊥ ⊤ ⍎ ⍕ ⍷ ⍉

A typical statement in APL is of the form:

```
      AREA←3×4
```

The effect of the statement is to assign to the name *AREA* the value of the expression 3×4 to the right of the specification arrow ← ; the statement may be read informally as "*AREA* is three times four."

The statement is the normal unit of execution. Two primitive types occur, the specification illustrated above, and the branch, which serves to control the sequence in which the statements in a defined function (discussed in Section 6) are executed. There is also a third type of statement which may invoke the use of a defined function without either a specification or a branch.

A variant of the specification statement produces display of a result on the medium (paper or screen) provided by the terminal device; if the leftmost part of a statement is not a name followed by a specification, the result of the expression is displayed. For example:

```
      3×4
12
      PERIMETER←2×(3+4)
      PERIMETER
14
```

Printing of the result of any part of a statement can be obtained by including the characters □← at the appropriate point in the statement. Moreover, any number of specification arrows may occur in a statement. For example:

```
      X←2+□←3×Y←4
12
      X
14
      Y
4
```

The terminal entry and display devices used with APL systems include a variety of typewriter-like and display-tube devices. Their characteristics vary, but the essential common characteristics are:

1. The ability to enter and display APL characters.
2. A means of signalling completion (and release to the system) of an entry.
3. Facilities for convenient revision of an entry before release.
4. Facilities to interrupt execution at the end of a statement (weak interrupt) and within a statement (strong interrupt).
5. A cursor, such as an arrowhead, to show where on the line the next character entered will appear.

All examples in this manual are presented as produced on a typewriter-like device on which the release signal is produced by the carrier return key, and revision is handled by backspacing to the point of revision, striking the attention button, and entering the revision. A caret supplied by the system marks the point of correction. For

example:

```
      3+4×5+
         ^
         +5+6
18
```

On terminals of this type the attention key is also used for interrupting execution. A single strike of this key while execution is in progress provides a weak interrupt, and a double strike provides a strong interrupt.

Entry of a statement which cannot be executed will invoke an error report which indicates the nature of the error and the point at which execution stopped. For example:

```
      X←5
      3+(Y×X)
VALUE ERROR
      3+(Y×X)
         ^
```

Error messages are listed in Figure 2.1, together with information on the cause and suggested corrective action.

CHARACTER SET

The characters which may occur in a statement fall in four main classes: alphabetic, numeric, special, and blank. The alphabetics comprise the roman alphabet in upper case italic font, and the same alphabet underscored. The entire set is shown in Figure 2.2 together with suggested names and the scheme for forming (as composites of other symbols) those which may not be directly available on the keys of certain terminal devices. A typical keyboard layout is shown in Figure 2.3.

The names suggested are for the symbols themselves and not necessarily for the functions they represent. For example, the downstile ⌊ represents both the minimum, a function of two arguments, and the floor (or integer part), a function of one argument. In general, most of the special characters (such as +, -, ×, and ÷) are used to denote primitive functions which are assigned fixed meanings, and the alphabetic characters are used to form names which may be assigned and re-assigned significance as variables, defined functions, and other objects. The blank serves as a separator to mark divisions between names (which are of arbitrary length).

For terminal devices which permit the display font to be changed without changing the behavior of the entry keyboard or communication with the system (as in changing the typing element on certain typewriters), any available display font may be used. For example, in textual work a font with normal upper- and lower-case roman is commonly employed.

SCALAR AND VECTOR CONSTANTS

All numbers entered or displayed are in decimal, either in conventional form (including a decimal point if appropriate) or in scaled form. The scaled form consists of an integer or decimal fraction called the multiplier followed immediately by the symbol *E* followed immediately by an integer (which must not include a decimal point) called the scale. The scale specifies the power of ten by which the multiplier is to be multiplied. Thus 1.44*E*2 is equivalent to 144.

TYPE	Cause; CORRECTIVE ACTION
CHARACTER	Illegitimate overstrike.
DOMAIN	Arguments not in the domain of the function.
DEFN	Misuse of ∇ or ⍞ symbols: 1. The function is pendent. DISPLAY STATE INDICATOR AND CLEAR AS REQUIRED. 2. Use of other than the function name alone in reopening a definition. 3. Improper request for a line edit or display.
⎕-- *IMPLICIT*	The system variable ⎕-- (for example, ⎕*IO*) has been set to an inappropriate value.
INDEX	Index value out of range.
INTERFACE QUOTA EXHAUSTED	Attempt to share more variables than allotted quota. REQUEST LARGER QUOTA FROM APL OPERATOR.
INTERRUPT	Execution was suspended within an APL statement. TO RESUME EXECUTION, ENTER A BRANCH TO THE STATEMENT INTERRUPTED.
LENGTH	Shapes not conformable.
NO SHARES	Shared variable facility not in operation.
RANK	Ranks not conformable.
RESEND	Transmission failure or more than the implementation-allowed number of characters entered in one line. RE-ENTER. IF CHRONIC, REDIAL OR HAVE TERMINAL OR PHONE REPAIRED.
SI DAMAGE	The state indicator (an internal list of suspended and pendent functions) has been damaged in editing a function or in performing a)*COPY* or)*ERASE*.
SYNTAX	Invalid syntax; e.g. two variables juxtaposed; function used without appropriate arguments as dictated by its header; unmatched parentheses.
SYMBOL TABLE FULL	Too many names used. SAVE, CLEAR, COPY or SAVE, CLEAR, SYMBOLS, COPY or ERASE, SAVE, CLEAR, COPY
SYSTEM	Fault in internal operation of the system. RELOAD. SEND TYPED RECORD, INCLUDING ALL WORK LEADING TO THE ERROR, TO THE SYSTEM MANAGER.
VALUE	Use of name which has not been assigned a value. ASSIGN A VALUE TO THE VARIABLE, OR DEFINE THE FUNCTION.
WS FULL	Workspace is filled (perhaps by temporary values produced in evaluating a compound expression, or by values of shared variables). CLEAR STATE INDICATOR, ERASE NEEDLESS OBJECTS, OR REVISE CALCULATIONS TO USE LESS SPACE.

Figure 2.1: Error Reports

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							
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							
0 1 2 3 4 5 6 7 8 9							
¨	dieresis	α	alpha	⍱	nor	~	∨
¯	overbar	⌈	upstile	⍲	nand	~	∧
<	less	⌊	downstile	⍒	del stile	∇	|
≤	not greater	_	underbar	⍋	delta stile	∆	|
=	equal	∇	del	⍉	circle stile	○	|
≥	not less	∆	delta	⌽	circle slope	○	\
>	greater	∘	null	⊖	circle bar	○	-
≠	not equal	'	quote	⍟	log	○	*
∨	or	⎕	quad	⌶	I-beam	⊥	⊤
∧	and	(	open paren	⍫	del tilde	∇	~
-	bar	)	close paren	⍬	base null	⊥	∘
÷	divide	[	open bracket	⍭	top null	⊤	∘
+	plus	]	close bracket	⍀	slope bar	\	-
×	times	⊂	open shoe	⌿	slash bar	/	-
?	query	⊃	close shoe	⍾	cap null	∩	∘
ω	omega	∩	cap	⍞	quote quad	'	⎕
∈	epsilon	∪	cup	!	quote dot	'	.
⍴	rho	⊥	base	⌸	domino	⎕	÷
~	tilde	⊤	top				
↑	up (arrow)	|	stile				
↓	down (arrow)	;	semicolon				
⍳	iota	:	colon				
○	circle	,	comma				
*	star	.	dot				
→	right (arrow)	\	slope				
←	left (arrow)	/	slash				
			space				

Figure 2.2: The APL Character Set

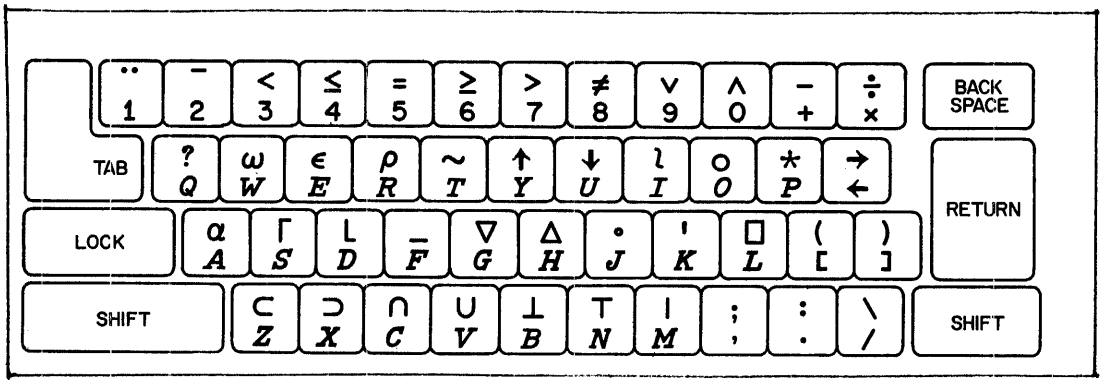


Figure 2.3: Typical APL Keyboard

Negative numbers are represented by an overbar immediately preceding the number, for example, ¯1.44 and ¯144E¯2 are equivalent negative numbers. The overbar can be used only as part of a constant and is to be distinguished from the bar which denotes negation, as in -X.

A single numerical constant entered by itself is accepted by the system as a scalar. A constant vector may be entered by listing the numerical components in order, separated by one or more spaces. A scalar character constant may be entered by placing the character between quotation marks, and a character vector may be entered by listing the characters between quotation marks. Such a vector is displayed by the system as the sequence of characters, with no enclosing quotes and with no separation of the successive elements. The quote character itself must be entered as a pair of quotes. Thus, the abbreviation of *CANNOT* is entered as `'CAN''T'` and prints as *CAN'T*.

SPACES

The blank character is used primarily as a separator. The spaces that one or more blank characters produce are needed to separate names of adjacent defined functions, constants, and variables. For example, if *F* is a defined function, then the expression `3 F 4` must be entered with the indicated spaces. The exact number of spaces used in succession is of no importance, and extra spaces may be used freely. Spaces are not required between primitive functions and constants or variables, or between a succession of primitive functions, but they may be used if desired. For example, the expression `3+4` may be entered with no spaces.

FUNCTIONS

The word "function" derives from a word which means to execute or to perform. A function executes some action on its argument (or arguments) to produce a result which may serve as an argument to another function. For example:

```
      3×4
12
      2+(3×4)
14
      (-6)÷3
¯2
```

A function (such as the negation used above) which takes one argument is said to be monadic, and a function (such as times) which takes two arguments is said to be dyadic. All APL functions are either monadic or dyadic or, in the case of defined functions only, niladic (taking no argument). Certain of the special symbols are used to denote two different functions, one monadic and the other dyadic. For example, *X-Y* denotes subtraction of *Y* from *X* (a dyadic function), and *-Y* denotes negation of *Y* (a monadic function). Other examples appear in Figures 3.1 and 3.5.

Each of the primitive functions is denoted by a single character or by an operator (discussed in Section 3) applied to such a character. For example, + and × are primitive functions as are +/ and ×/ (since / denotes an operator).

ORDER OF EXECUTION

Parentheses are used in the familiar way to control the order of execution in a statement. Any expression within matching parentheses is evaluated before applying to the result any function outside the matching pair.

In conventional notation, the order of execution of an unparenthesized sequence of monadic functions may be stated as follows: the (right-hand) argument of any function is the value of the entire expression to the right. For example, Log Sin Arctan x means the Log of Sin Arctan x, which means Log of Sin of Arctan x. In APL, the same rule applies to dyadic functions as well. Moreover, all functions, both primitive and defined, are treated alike; there is no hierarchy among functions (such as multiplication being done before addition or subtraction).

An equivalent statement of this rule is that an unparenthesized expression is evaluated in order from right to left. For example, the expression `3×8⌈3*|5-7` is equivalent to `3×(8⌈(3*(|(5-7))))`. Their result is `27`. A consequence of the rule is that the only substantive use of parentheses is to form the left argument of a function. For example, `(12÷3)×2` is `8` and `12÷3×2` is `2`. However, redundant pairs of parentheses can be used at will. Thus, `12÷(3×2)` is also `2`.

ARRAYS

APL functions apply to collections of individual items called arrays. Arrays range from scalars, which are dimensionless, to multi-dimensional arrays of arbitrary rank and size.

The vector is a simple form of array which may be formed by listing its elements as described in the discussion of constants. For example:

```
      V←2 3 5 7 11 13 17 19
      A←'ABCDEFGH'
```

The elements of a vector may be selected by indexing. For example:

```
      V[3 1 5]
5 2 11
      A[8 5 1 4]
HEAD
```

Arrays of more complex structure may be formed with the reshape function denoted by ρ:

```
      M←2 4ρV                    B←2 4ρA
      M                          B
 2  3  5  7              ABCD
11 13 17 19              EFGH
```

These results have two dimensions or axes and are called tables or matrices. A matrix has two axes and is said to be of rank 2; a vector has one axis and is of rank 1.

The left argument `2 4` in the foregoing examples specifies the shape of the resulting array. Arrays of arbitrary shape and rank may be

produced by the same scheme. For example:

```
      T←2 3 4ρ'ABCDEFGHIJKLMNOPQRSTUVWX'
      T
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
```

The shape of an array can be determined by the monadic function denoted by ρ:

```
      ρV              ρM              ρT
8               2 4             2 3 4
```

Elements may be selected from any array (other than a scalar) by indexing in the manner shown for vectors, except that indices must be provided for each axis:

```
      M[2;3]                          T[2;1;4]
17                              P
      M[2 1;2 3 4]                    T[2;1 2 3;1 2 3 4]
13 17 19                        MNOP
 3  5  7                        QRST
                                UVWX
```

The indexing used in the foregoing examples is called 1-origin because the first element along each axis is selected by the index 1. One may also use 0-origin indexing by setting the index origin to 0. The index origin is a system variable (See Section 4) denoted by ⎕IO. Thus:

```
      ⎕IO←1                           ⎕IO←0
      V[1 2 3]                        V[0 1 2]
2 3 5                           2 3 5
      B[2;3]                          B[1;2]
G                               G
```

All further examples assume 1-origin unless otherwise stated.

WORKSPACES AND LIBRARIES

The common organizational unit in an APL system is the workspace. When in use, a workspace is said to be active, and it occupies a block of working storage in the central computer. Part of each workspace is set aside to serve the internal workings of the system, and the remainder is used, as required, for storing items of information and for containing transient information generated in the course of a computation.

An active workspace is always associated with a terminal during a work session, and all transactions with the system are mediated by it. In particular, the names of variables (data items) and defined functions (programs) used in calculations always refer to objects known by those names in the active workspace; information on the progress of program execution is maintained in the state indicator of the active workspace; and control information affecting the form of output is held within the active workspace.

Inactive workspaces are stored in libraries, where they are identified by arbitrary names. They occupy space in secondary storage facilities of the central computer and cannot be worked with directly. When required, copies of stored workspaces can be made active, or selected information may be copied from them into an active workspace.

Libraries are either private or public. Private libraries are associated with individual users of the system, and are identified by the user's account number. Access to them by other users is restricted in that one user may not store workspaces in another person's library, nor obtain a listing of the workspaces already stored there. However, one user may activate a copy of another user's unlocked workspace if he knows the library number and workspace name. Workspaces and libraries are managed by system commands as described in Section 8.

NAMES

Names of workspaces, functions, variables, and groups (see Section 8) may be formed of any sequence of alphabetic and numeric characters that starts with an alphabetic and contains no blank.

The environment in which APL operations take place is bounded by the active workspace. Hence, the same name may be used to designate different objects (that is, groups, functions, or variables) in different workspaces, without interference. Also, since workspaces themselves are never the subject of APL operations, but only of system commands, it is possible for a workspace to have the same name as an object it holds.

Stored workspaces and the information they hold can be protected against unauthorized use by associating a lock, comprising a colon and a password of the user's choice, with the name of the workspace, when the workspace is stored. In order to activate a locked workspace or copy any information it contains, a colon and the password must again be used, as a key, in conjunction with the workspace name. Listings of workspace names, including those in public libraries, never give the keys, and do not overtly indicate the existence of a lock.

USER IDENTIFICATION

Account numbers can be similarly protected by locks and keys, thus maintaining the security of a user's private library and avoiding unauthorized charges against his account. Passwords for locks and keys may be formed of any sequence of alphabetic and numeric characters. In use as either a lock or key, a password follows the number or name it is protecting, from which it is set off by a colon.

APL systems differ in the procedure required to sign-on or begin a session. However, a numerical identification of the user is usually provided in some manner, so that it can be used to specify the first element of $\square AI$.

The primitive functions fall in two classes, scalar, and mixed. Scalar functions are defined on scalar arguments and are extended to other arrays item-by-item. Mixed functions are defined on arrays of various ranks and may yield results which differ from the arguments in both rank and shape. Five primitive operators apply to scalar dyadic functions and to certain mixed functions to produce a large number of new functions.

The definitions of certain functions depend on certain system variables whose names begin with the symbol ⎕ (as in ⎕*IO* and ⎕*CT*). These system variables are discussed further in Section 4.

SCALAR FUNCTIONS

A monadic scalar function extends to each item of an array argument; the result is an array of the same shape as the argument and each item of the result is obtained as the monadic function applied to the corresponding item of the argument.

A dyadic scalar function extends similarly to a pair of arguments of the same shape. To be conformable, the arguments must agree in shape, or at least one of them must be a scalar or one-element vector. If one of the arguments has only one item, that item is applied in determining each element of the result. For example:

```
      1 2 3×4 5 6
4 10 18
      3+4 5 6
7 8 9
      2 3+4 5 6
LENGTH ERROR
```

Each of the scalar functions is defined on all real numbers with two general exceptions: the four boolean functions are defined only on the numbers 0 and 1, and the functions = and ≠ are defined on characters as well as numbers. Specific exceptions (such as 4÷0) will be noted where appropriate.

The scalar functions are summarized in Figure 3.1 together with their symbols and brief definitions or examples which should make their use clear. The remainder of this section is devoted to more detailed definitions.

A dyadic function f may possess a left identity element *L* such that *L* f *X* equals *X* for any *X*, or a right identity element *R* such that *X* f *R* equals *X*. For example, one is a right identity element of ÷ since *X*÷1 is *X*, zero is a left or right identity of +, one is a left or right identity of ×, and the general logarithm function ⍟ has no identity element.

Identity elements become important as the appropriate result of applying a function over an empty vector; for example, the sum over an empty vector is 0, (the identity element of +), and the product over an empty vector is 1, the identity element of ×. These matters are discussed further in the treatment of the reduction operator which concerns such applications of dyadic functions over vectors.

Figure 3.2 lists the identity elements of the dyadic scalar functions. The relational functions <, ≤, =, ≥, > and ≠ possess no true identity elements, but do when considered as boolean functions, that is,

Monadic form fB		f	Dyadic form AfB	
Definition or example	Name		Name	Definition or example
$+B$ is B	Conjugate	+	Plus	2+3.2 is 5.2
$-B$ is 0-B	Negative	-	Minus	2-3.2 is ¯1.2
$\times B$ is (B>0)-B<0	Signum	×	Times	2×3.2 is 6.4
$\div B$ is 1÷B	Reciprocal	÷	Divide	2÷3.2 is 0.625
|¯3.14 is 3.14	Magnitude	|	Residue	A|B is B-A×⌊B÷A+A=0
B ⌊B ⌈B 3.14 3 4 ¯3.14 ¯4 ¯3	Floor	⌊	Minimum	3⌊7 is 3
	Ceiling	⌈	Maximum	3⌈7 is 7
?B is Random choice from ⍳B	Roll	?	Deal	A Mixed Function (See Figure 3.5)
$*B$ is (2.71828..)*B	Exponential	*	Power	2*3 is 8
⍟*N is N is *⍟N	Natural logarithm	⍟	General logarithm	A⍟B is Log B base A A⍟B is (⍟B)÷⍟A
○B is B×3.14159...	Pi times	○	Circular, Hyperbolic, Pythagorean (See table at left)	
!0 is 1 !B is B×!B-1 or !B is Gamma(B+1)	Factorial	!	Binomial	A!B is (!B)÷(!A)×!B-A 2!5 is 10 3!5 is 10
~1 is 0 ~0 is 1	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	

A	B	$A\wedge B$	$A\vee B$	$A⍲B$	$A⍱B$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

<	Less	Relations	
≤	Not greater	Result is 1 if the	
=	Equal	relation holds, 0	
≥	Not less	if it does not:	
>	Greater	3≤7 is 1	
≠	Not Equal	7≤3 is 0	

(-A)○B	A	A○B
(1-B*2)*.5	0	(1-B*2)*.5
Arcsin B	1	Sine B
Arccos B	2	Cosine B
Arctan B	3	Tangent B
(¯1+B*2)*.5	4	(1+B*2)*.5
Arsinh B	5	Sinh B
Arcosh B	6	Cosh B
Artanh B	7	Tanh B

Table of Dyadic ○ Functions

Figure 3.1: Primitive Scalar Functions

when restricted to the domain 0 and 1. These identity elements are included in the figure.

Dyadic Function		Identity Element	Left-Right
Plus	≤	0	L R
Minus	-	0	R
Times	×	1	L R
Divide	÷	1	R
Residue	|	0	L
Minimum	⌊	(note 1)	L R
Maximum	⌈	(note 2)	L R
Power	*	1	R
Logarithm	⍟		None
Circle	○		None
Binomial	!	1	L
And	∧	1	L R
Or	∨	0	L R
Nand	⍲		None
Nor	⍱		None
Less	<	0 Apply	L
Not greater	≤	1 for	L
Equal	=	1 boolean	L R
Not less	≥	1 arguments	R
Greater	>	0 only	R
Not equal	≠	0	L R

Note 1: the largest representable number

Note 2: the greatest in magnitude of representable negative numbers

Figure 3.2: Identity Elements of Primitive Scalar Dyadic Functions

PLUS, MINUS, TIMES, DIVIDE AND RESIDUE

The definitions of the first four of these functions agree with the familiar definitions except that the indeterminate case 0÷0 is defined to yield the value 1. For $X \neq 0$, the expression $X \div 0$ evokes a domain error.

If A and B are positive integers, then the residue $A|B$ is the remainder on dividing A into B. The following definition covers all values of A and B:

1. If $A=0$, then $A|B$ equals B
2. If $A \neq 0$, then $A|B$ lies between A and zero (being permitted to equal zero but not A) and is equal to $B-N \times A$ for some integer N.

For example:

```
      1|2.385          0|5.8          ¯3|¯3 ¯2 ¯1 0 1 2 3
0.385                  5.8          0 ¯2 ¯1 0 ¯2 ¯1 0
                                        3|¯3 ¯2 ¯1 0 1 2 3
                                    0 1 2 0 1 2 0
```

CONJUGATE, NEGATIVE, SIGNUM, RECIPROCAL, MAGNITUDE

The conjugate function $+X$ yields its argument unchanged, the negative function $-X$ yields the argument reversed in sign, and the reciprocal $\div X$ is equivalent to $1\div X$. For example, if $X\leftarrow 4\ {}^{-}5$, then:

```
      +X                    -X                  ÷X
4 ¯5                  ¯4 5              0.25 ¯0.2
```

The result of the signum function $\times X$ depends on the sign of its argument, being $^{-}1$ if $X<0$, and 0 if $X=0$, and 1 if $X>0$. The magnitude function $|X$ (also called absolute value) yields the greater of X and $-X$; in terms of the signum function it is equivalent to $X\times\times X$. For example:

```
      ×¯3 0 4                          |¯3 0 4
¯1 0 1                           3 0 4
```

BOOLEAN AND RELATIONAL FUNCTIONS

The boolean functions and, or, nand (not-and), and nor (not-or) apply only to boolean arguments, that is, 0 and 1; if 0 is interpreted as false, and 1 as true, then the definitions of these functions are evident from their names. For example, $A\wedge B$ (read as A and B) equals 1 (is true) only if A equals 1 (is true) and B equals 1. All cases are covered by the following examples:

```
      A←0 0 1 1
      B←0 1 0 1
      A∧B          A∨B          A⍲B          A⍱B
0 0 0 1      0 1 1 1      1 1 1 0      1 0 0 0
```

The monadic function not yields the logical complement of its argument, that is ~0 is 1, and ~1 is 0.

The relational functions apply to any numbers, but yield only boolean results, that is 0 or 1. The result is 1 if the indicated relation holds, and 0 otherwise. For example:

```
      3 5<5 3                          3 5 7≠7 5 3
1 0                              1 0 1
```

The comparisons involved in determining the results of the relational functions are not absolute, but are made to a certain tolerance specified by the comparison tolerance $\square CT$. Two scalar quantities A and B are considered to be equal if the magnitude of their difference does not exceed the value of $\square CT$ multiplied by the larger of the magnitudes of A and B, that is, if $(|A-B)$ is less than or equal to $\square CT\times(|A)\lceil|B$.

Similarly, $A\geq B$ is considered to be true if $(A-B)$ is greater than or equal to $-\square CT\times(|A)\lceil(|B)$, and $A>B$ is considered true if $A\geq B$ is true and $A=B$ is not.

The comparison tolerance $\square CT$ is typically set to the value $1E^{-}13$. The setting $\square CT\leftarrow 0$ is also useful since it yields absolute comparisons, but may lead to unexpected results due to the finite precision of the representation of numbers. For example, if the maximum precision is 16

decimal digits and all digits are displayed in printing, then:

```
      ⎕CT←0
      X←2÷3
      X
0.6666666666666667
      Y←3×X
      Y-2
¯2.220446049250313E¯16
      2=Y
0
      ⎕CT←1E¯13
      2=Y
1
```

When applied to boolean arguments only, the relations are, in effect, boolean functions, and denote functions which may be familiar from the study of logic, although referred to by different names and symbols. For example, $X \neq Y$ is the exclusive-or of X and Y, and $X \leq Y$ is material implication. This association should be clear from the following table, which lists in the first two columns the four possible sets of values of two boolean arguments, and in the remaining columns the values of the 16 possible boolean functions, with the symbols of the boolean and relational functions of APL appended to appropriate columns:

```
A B          A f B
0 0  0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 1  0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
1 0  0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1  0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
       ∧ >   <   ≠ ∨ ⍱ =   ≥   ≤ ⍲
```

The ten functions listed at the foot of this table embrace all non-trivial boolean functions of two arguments. Consequently, any boolean expression of two arguments X and Y can be replaced by a simple APL expression as follows: evaluate the expression for the four possible cases, locate the corresponding column in the table, then use the function symbol at the foot of the column, or, if none occurs, use X or Y or $\sim X$ or $\sim Y$ or 0 or 1 as appropriate.

MINIMUM AND MAXIMUM

The functions denoted by ⌊ and ⌈ perform as expected from their names. For example:

```
      X←¯3 ¯2 ¯1 0 1 2 3
      Y←3 2 1 0 ¯1 ¯2 ¯3
      X⌈Y
3 2 1 0 1 2 3
      X⌊Y
¯3 ¯2 ¯1 0 ¯1 ¯2 ¯3
```

FLOOR AND CEILING

The floor function ⌊ yields the integer part of its argument, that is, $\lfloor X$ yields the largest integer which does not exceed X. Similarly, $\lceil X$ yields the smallest integer which is not less than X. For example:

```
      X←¯3.14 2.718
      ⌊X
¯4 2
      -⌈-X
¯4 2
```

```
      ⌈X
¯3 3
      -⌊-X
¯3 3
```

The ceiling and floor functions are affected by the comparison tolerance $\square CT$ as follows: if there is an integer I for which $|X-I$ does not exceed the value of $\square CT$, then both $\lfloor X$ and $\lceil X$ equal I. For example, if results are represented and printed to 16 decimal digits, then:

```
      X←3×2÷3
      □CT←1E¯13
      ⌊X
2
      ⌈X
2
```

```
      □CT←0
      ⌊X
2
      ⌈X
3
```

ROLL (Random number function)

The roll is a monadic function named by analogy with the roll of a die; thus $?6$ yields a (pseudo-) random choice from $\iota 6$, that is the first six integers beginning with either 0 or 1 according to the value of the index origin $\square IO$. For example:

```
      □IO←1
      ?6
1
      ?6
5
      ?6
3
      ?6 6 6 6 6 6 6 6 6 6 6 6
4 2 1 5 5 6 3 4 5 1 1 4 5
      □IO←0
      ?6 6 6 6 6 6 6 6 6 6 6 6
0 2 0 2 4 3 5 5 3 0 3 2 4
```

The domain of the roll function is limited to positive integers.

The roll function employs an algorithm due to D.H. Lehmer; the result for each scalar argument X is a function of X and of the random link variable $\square RL$. The result of the roll function is system dependent but typically for $X<2*31$ is equal to $\square IO$ plus the integer part of $X*\square RL\div 1+2*31$.

POWER, EXPONENTIAL, GENERAL AND NATURAL LOGARITHM

For non-negative integer right arguments the power function $X*N$ is simply defined as the product over N repetitions of X. It is generalized to non-positive and non-integer arguments so as to preserve the relation that $X*A+B$ shall equal $(X*A)\times(X*B)$. Familiar consequences of this extension are that $X*-N$ is the reciprocal of $X*N$, and $X*\div N$ is the Nth root of X. For example:

```
      2*¯3 ¯2 ¯1 0 1 2 3
.125 .25 .5 1 2 4 8
      64*÷1 2 3 4 5 6
64 8 4 2.828 2.297 2
```

The indeterminate case $0*0$ is defined to have the value 1.

The domain of the power function $X*Y$ is restricted in two ways: if $X=0$, then Y must be non-negative; if $X<0$, then Y must be an integer or a (close approximation to a) rational number with an odd denominator. For example, $\overline{\ }8*.5$ yields a domain error, but $\overline{\ }8*1\div 3$ and $\overline{\ }8*2\div 3$ yield $\overline{\ }2$ and 4, respectively.

The exponential function $*X$ is equivalent to the expression $e*X$, where e is the base of the natural logarithms, approximately 2.71828.

For example:

```
      *¯2 ¯1 0 1 2
0.1353352832 0.3678794412 1 2.718281828 7.389056099
```

The natural logarithm function ⍟*X* is the inverse of the exponential, that is, *⍟X* and ⍟**X* both equal *X*. For example:

```
      ⍟1 2 3 4
0 0.6931471806 1.098612289 1.386294361
      *⍟1 2 3 4
1 2 3 4
      ⍟*1 2 3 4
1 2 3 4
```

The domain of the natural logarithm function is limited to positive numbers.

The general logarithm function *B*⍟*X* is defined as (⍟*X*)÷⍟*B*. It is inverse to the power function in the following sense: *B***B*⍟*X* and *B*⍟*B***X* both equal *X*. Limitations on the domain follow directly from the defining expression.

CIRCULAR, HYPERBOLIC, AND PYTHAGOREAN FUNCTIONS

The symbol ○ denotes a monadic function whose result equals pi times its argument. For example:

```
      ○1 2 .5
3.141592654 6.283185307 1.570796327
```

The symbol ○ is also used dyadically to denote a family of fifteen related functions as follows: the expression *I*○*X* is defined for integer values of *I* from ¯7 to 7, and is in each case equivalent to one of the circular, hyperbolic, or pythagorean functions as indicated in Figure 3.1.

The arguments for the circular functions Sin, Cos, and Tan (1○*X*, 2○*X*, and 3○*X*) are in radians. For example:

```
      PI←○1
      1○PI÷2 3 4
1 0.8660254038 0.7071067812
```

The hyperbolic functions sinh and cosh are the odd and even components of the exponential function; that is, 5○*X* is odd, 6○*X* is even, and the sum (5○*X*)+(6○*X*) is equivalent to **X*. Consequently:

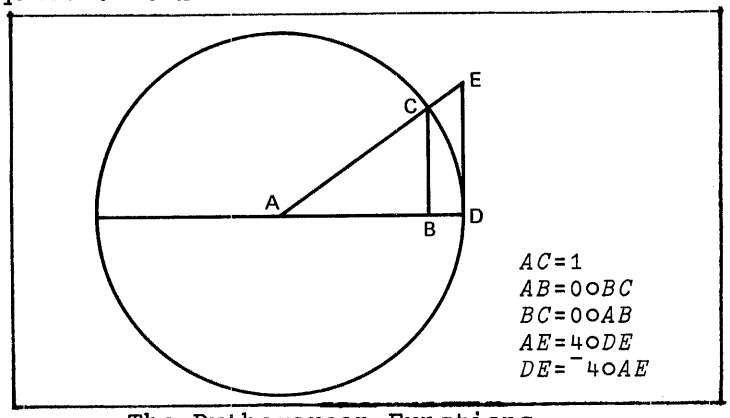
5○*X* equals .5×(**X*)-(*-*X*)
6○*X* equals .5×(**X*)+(*-*X*)

The definition of the hyperbolic tangent is analogous to that of the tangent, that is, 7○*X* equals (5○*X*)÷6○*X*.

The three pythagorean functions 0○*X*, 4○*X*, and ¯4○*X* are defined as in Figure 3.1, and are related to the properties of a right triangle as

indicated in Figure 3.3. They may also be defined as follows:

```
¯4○X equals 5○¯6○X
 0○X equals 2○¯1○X or 1○¯2○X
 4○X equals 6○¯5○X
```



The Pythagorean Functions

Figure 3.3

Each of the family of functions $I○X$ has an inverse in the family, that is, $(-I)○X$ is the inverse of $I○X$. Certain of the functions are not monotonic, and their inverses are therefore many-valued. The principal values are chosen in the following intervals:

Arcosh	$R←¯6○X$	$R≥0$
	$R←¯4○X$	$R≥0$
Arccos	$R←¯2○X$	$(R≥0)∧(R≤○1)$
Arcsin	$R←¯1○X$	$(\|R)≤○.5$
	$R←\ 0○X$	$R≥0$
	$R←\ 4○X$	$R≥0$

FACTORIAL AND BINOMIAL FUNCTIONS

For positive integer arguments, the factorial function $!N$ is defined as the product of all positive integers up to N. An important consequence of this definition is that $!N$ equals $N×!N-1$, or, equivalently $!N-1$ equals $(!N)÷N$. This relation is used to extend the function to non-integer and negative arguments. For example:

```
      N←1 2 3 4 5
      !N
1 2 6 24 120
      (!N)÷N
1 1 2 6 24
      !0 1 2 3 4
1 1 2 6 24
      F←.5 1 1.5 2 2.5
      !F
0.8862269255 1 1.329340388 2 3.32335097
      (!F)÷F
1.772453851 1 0.8862269255 1 1.329340388
      !¯.5 0 .5 1 1.5
1.772453851 1 0.8862269255 1 1.329340388
```

This extension leads to the expression $(!0)÷0$, or $1÷0$ for $!¯1$, and $¯1$ is therefore excluded from the domain of factorial, as are all negative integers.

The binomial function *M*!*N* is defined, for non-negative integer arguments, as the number of distinct ways in which *M* things can be chosen from *N* things. The expression (!*N*)÷(!*M*)×(!*N*-*M*) yields an equivalent definition which is used to extend the definition to all numbers. Although the domain of factorial excludes negative integers, the domain of the binomial does not, since any implied division by zero in the numerator !*N* is invariably accompanied by a corresponding division by zero in the denominator; the function therefore extends smoothly to all numbers.

The result of *I*!*N* is equivalent to coefficient *I* in the binomial expansion (*X*+1)**N* . For example:

```
      0 1 2 3!3
1 3 3 1
```

OPERATORS

An operator may be applied to a function to yield a different function. For example, the outer product operator denoted by the symbols ∘. may be applied to any of the primitive scalar dyadic functions to produce a corresponding "table function" as illustrated below for times and power:

```
      A←1 2 3 4
      A∘.×A
1  2  3  4
2  4  6  8
3  6  9 12
4  8 12 16
      A∘.*A
1   1   1   1
2   4   8  16
3   9  27  81
4  16  64 256
```

Four of the APL operators - reduction, scan, inner product, and outer product - apply to any primitive scalar dyadic function. The axis operator applies to reduction and scan , and also to certain of the mixed functions.

REDUCTION

Reduction is denoted by the symbol / and applies to the function which precedes it. For example, if *V*←1 2 3 4 5, then +/*V* yields the sum of the items of *V*, and ×/*V* yields their product:

```
      +/V
15
      ×/V
120
```

In general, an expression of the form f/*V* is equivalent to the expression obtained by placing the function symbol f between adjacent pairs of items of the vector *V*:

```
      ⌈/V
5
      1⌈2⌈3⌈4⌈5
5
      -/V
3
      1-2-3-4-5
3
```

The last example points up the fact that the general rule for the order of execution is applied, and that as a consequence the expression -/*V* yields the alternating sum of the items of *V*. The alternating sum is defined as the sum obtained after first weighting the items by

multiplying alternate elements by 1 and ¯1. Thus:

```
      A←1 ¯1 1 ¯1 1
      V×A
1 ¯2 3 ¯4 5
      +/V×A
3
      -/V
3
```

Similarly, ÷/*V* yields the alternating product:

```
      V*A
1 .5 3 .25 5
      ×/V*A
1.875
      ÷/V
1.875
```

The result of applying reduction to any scalar or vector is a scalar; the value in the case of a scalar or one-element vector argument is the single item itself. The application of reduction to other arrays is treated in the discussion of the axis operator.

Reduction of an empty vector by any function is defined as the identity element of the function if one exists, as a domain error if not. Thus if *V* is an empty vector, +/*V* equals 0, and ∧/*V* equals 1.

The reason for this definition is the extension to empty vectors of an important relation between the reductions of two vectors *P* and *Q* and the reduction of the vector *V*←*P*,*Q* obtained by chaining them together. For example:

```
      +/V   equals  (+/P)+(+/Q)
      ×/V   equals  (×/P)×(×/Q)
```

If *P* is an empty vector it is clear that +/*P* must equal 0 (the identity element of +), and that ×/*P* must equal 1.

SCAN

The scan operator is denoted by the symbol \ and applies to the function which precedes it. When the resulting function is applied to a vector *V* it yields a vector of the same shape whose *K*th element is equal to the corresponding reduction over the first *K* elements of *V*. For example:

```
      +\1 2 3 4 5
1 3 6 10 15
      ×\1 2 3 4 5
1 2 6 24 120
      ∨\0 0 1 0 1
0 0 1 1 1
      ∧\1 1 0 1 0
1 1 0 0 0
      <\0 0 1 0 1 1 0
0 0 1 0 0 0 0
```

The extension of scan to arrays other than vectors is treated in the discussion of the axis operator.

AXIS

Since reduction is defined on vectors, and since a matrix can be conceived as a collection of row vectors, the result of reduction on a matrix could be defined as the vector of results obtained by reduction of each of the row vectors. However, a matrix can also be viewed as a collection of column vectors, and reduction could be defined accordingly. It is therefore important to be able to specify which set of vectors is to be reduced.

The row vectors extend along the second axis of the matrix, and are each of length equal to the length of that axis. Similarly, the column vectors extend along the first axis and, in general, the index of an axis identifies a set of vectors in an array of any non-zero rank.

The axis operator is denoted by brackets enclosing an expression which yields the index of an axis as a one-element vector or a scalar. It determines the direction of application of any scan or reduction operator whose symbols immediately precede it. For example:

```
      M←3 4⍴⍳12
 1  2  3  4
 5  6  7  8
 9 10 11 12

      +\[1]M
 1  2  3  4
 6  8 10 12
15 18 21 24

      +\[2]M
 1  3  6 10
 5 11 18 26
 9 19 30 42

      +/[1]M
15 18 21 24

      +/[2]M
10 26 42
```

The shape of the result of scan equals the shape of the argument. The shape of the result of reduction equals the shape of the argument except that the indicated axis of reduction is removed. The indexing of axes depends on the index origin ⎕*IO*.

If no axis operator occurs, reduction and scan apply along the last axis. The symbols ⌿ and ⍀ also denote reduction and scan, but in the absence of axis operators they apply along the first axis.

The axis operator is also used to specify the axis of application of the mixed functions reverse, rotate, compress, and expand.

INNER PRODUCT

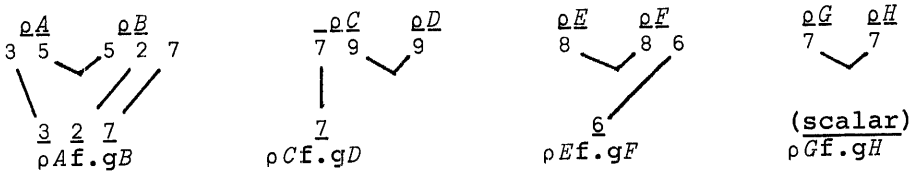
If *P* and *Q* are vectors of the same shape, then the expression +/*P*×*Q* has a variety of useful interpretations. For example, if *P* is a list of prices and *Q* a list of corresponding order quantities, then +/*P*×*Q* is the total cost. Expressions of the same form employing functions other than + and × are equally useful, as suggested by the following examples (where *B* is used to denote a boolean vector):

∧/*P*=*Q*	Comparison of *P* and *Q*.
+/*P*=*Q*	Count of agreements between *P* and *Q*.
⌊/*P*+*Q*	Minimum distance for shipment, where *P* represents the distances from source to transhipment points and *Q* the distances from these points to the destination.
+/*P*×*B*	Sum over a subset of *P* specified by *B*.
×/*P**B*	Product over a subset of *P* specified by *B*.

The inner product produces functions equivalent to expressions in this form; it is denoted by a dot and applies to the two functions which surround it. Thus *P*+.×*Q* is equivalent to +/*P*×*Q*, and *P*×.**B* is equivalent

to $\times/P*B$, and, in general, $Pf.gQ$ is equivalent to f/PgQ, if P and Q are vectors.

The inner product is extended to arrays other than vectors along certain fixed axes, namely the last axis of the first argument and the first axis of the last argument. The lengths of these axes must agree. The shape of the result is obtained by deleting these axes and chaining the remaining shape vectors together. The consequence for matrix arguments is illustrated in Figure 3.4. The consequences for the shape of inner products on some other arrays are illustrated below:



Formally, $\rho Af.gB$ equals $(^{-}1\downarrow\rho A),1\downarrow\rho B$.

The inner product $M+.\times N$ is commonly called the matrix product. Examples of it and other inner products follow:

```
      P←2 3 5 7
      M←(⍳4)∘.≤⍳4
      M                M+.×M            M∧.=M            M-.×M
1 1 1 1          1 2 3 4          0 0 0 1          1 ¯0 1 ¯0
0 1 1 1          0 1 2 3          0 0 0 0          0 ¯1 0 ¯1
0 0 1 1          0 0 1 2          0 0 0 0          0  0 1 ¯0
0 0 0 1          0 0 0 1          0 0 0 0          0  0 0 ¯1

      M+.×P            P+.×M            P×.*M            M∧.=0 0 1 1
17 15 12 7       2 5 10 17        2 6 30 210       0 0 1 0
```

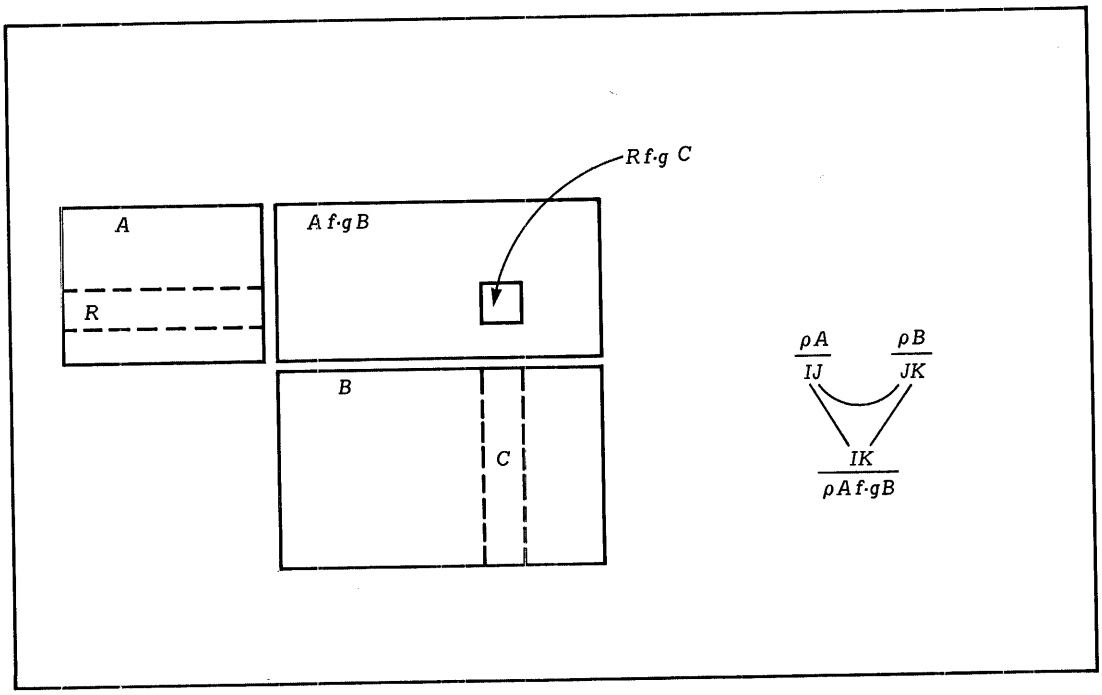


Figure 3.4: Inner Product

Either argument of an inner product may be a scalar; it is extended in the usual manner. For example, `A+.×1` is equivalent to `+/A`, and `1+.×A` is equivalent to `+⌿A`.

OUTER PRODUCT

The outer product operator, denoted by the symbols `∘.` preceding the function symbol, applies to any dyadic primitive scalar function, so that the function is evaluated for each member of the left argument paired with each member of the right argument. For example, if `A←1 2 3` and `B←1 2 3 4 5`:

```
      A∘.×B                 A∘.<B
1  2  3  4  5          0 1 1 1 1
2  4  6  8 10          0 0 1 1 1
3  6  9 12 15          0 0 0 1 1
```

Such tables may be better understood if labelled in a way widely used in elementary arithmetic texts: values of the arguments are placed beside and above the table, and the function whose outer product is being computed is shown at the corner. Thus:

```
        B                          B
  ×  | 1  2  3  4  5        <  | 1  2  3  4  5
  1  | 1  2  3  4  5           1 | 0  1  1  1  1
A 2  | 2  4  6  8 10         A 2 | 0  0  1  1  1
  3  | 3  6  9 12 15           3 | 0  0  0  1  1
```

In the foregoing example the shape of the result `A∘.×B` is clearly equal to `(⍴A),(⍴B)`. This expression yields the shape for any arguments `A` and `B`. Thus if `R←A∘.+B` and `A` is a matrix of shape `3 4` and `B` is a three-dimensional array of shape `5 6 7`, then `R` is a five-dimensional array of shape `3 4 5 6 7`. Moreover, `R[I;J;K;L;M]` equals `A[I;J]+B[K;L;M]` for all possible values of the indices.

MIXED FUNCTIONS

The mixed functions are grouped in five classes according to whether they concern the structure of arrays, selection from arrays, the generation of selector information for use by selection functions, numerical calculations, or transformations of data such as that between characters and numbers. All are listed in Figure 3.5 together with brief definitions or examples.

Those functions which may be modified by an axis operator may also be used without an axis operator, in which case the axis concerned is the last, or, in the case of the functions denoted by `⌽` and by `⌿`, the first.

Name	Sign[1]	Definition or example
Shape	⍴*A*	⍴*P* is 4 ⍴*E* is 3 4 ⍴5 is ⍳0
Reshape	*V*⍴*A*	Reshape *A* to dimension *V* 3 4⍴⍳12 is *E* 12⍴*E* is ⍳12 0⍴*E* is ⍳0
Ravel	,*A*	,*A* is (×/⍴*A*)⍴*A* ,*E* is ⍳12 ⍴,5 is 1
Reverse[5]	⌽*A*	*DCBA* *IJKL* ⌽*X* is *HGFE* ⌽[1]*X* is ⊖*X* is *EFGH* *LKJI* ⌽*P* is 7 5 3 2 *ABCD*
Rotate[5]	*A*⌽*A*	*BCDA* 3⌽*P* is 7 2 3 5 is ⍣¯1⌽*P* 1 0 ⍣¯1⌽*X* is *EFGH* *LIJK*
Catenate	*A*,*A*	*P*,⍳2 is 2 3 5 7 1 2 'T','HIS' is 'THIS'
Transpose[3]	*V*⍉*A*	*AEI* Coordinate *I* of *A* 2 1⍉*X* is *BFJ* becomes coordinate *CGK* *V*[*I*] of result 1 1⍉*E* is 1 6 11 *DHL*
	⍉*A*	Reverse order of coordinates ⍉*E* is 2 1⍉*E*
Take	*V*↑*A*	Take or drop |*V*[*I*] first 2 3↑*X* is *ABC* (*V*[*I*]≥0) or last (*V*[*I*]<0) *EFG*
Drop	*V*↓*A*	elements of coordinate *I* ⍣¯2↑*P* is 5 7
Compress[5]	*V*/*A*	1 3 1 0 1 0/*P* is 2 5 1 0 1 0/*E* is 5 7 9 11
		1 0 1/[1]*E* is 1 2 3 4 is 1 0 1⌿*E* 9 10 11 12
Expand[5]	*V**A*	*A BCD* 1 0 1\⍳2 is 1 0 2 1 0 1 1 1*X* is *E FGH* *I JKL*
Indexing[4]	*V*[*A*]	*P*[2] is 3 *P*[4 3 2 1] is 7 5 3 2
	M[*A*;*A*]	*E*[1 3;3 2 1] is 3 2 1 11 10 9
	A[*A*;.. ..;*A*]	*E*[1;] is 1 2 3 4 *ABCD* *E*[;1] is 1 5 9 'ABCDEFGHIJKL'[*E*] is *EFGH* *IJKL*
Index generator[3]	⍳*S*	First *S* integers ⍳4 is 1 2 3 4 ⍳0 is an empty vector
Index of[3]	*V*⍳*A*	Least index of *A* *P*⍳3 is 2 5 1 2 5 in *V*, or 1+⍴*V* *P*⍳*E* is 3 5 4 5 4 4⍳4 is 1 5 5 5 5
Membership	*A*∈*A*	0 1 1 0 ⍴*W*∈*Y* is ⍴*W* *E*∈*P* is 1 0 1 0 *P*∈⍳4 is 1 1 0 0 0 0 0 0
Grade up[3]	⍋*V*	The permutation which ⍋3 5 3 2 is 4 1 3 2
Grade down[3]	⍒*V*	would order *V* (ascend- ing or descending) ⍒3 5 3 2 is 2 1 3 4
Deal[3]	*S*?*S*	*W*?*Y* is Random deal of *W* elements from ⍳*Y*
Matrix inverse	⌹*M*	⌹2 2⍴1 1 0 1 is 1 ¯1 arguments may 0 1 be scalars,
Matrix division	*M*⌹*M*	(2 2⍴*P*)⌹2 2⍴1 1 0 1 is ¯3 ¯4 vectors, or 5 7 matrices
Decode	*A*⊥*A*	10⊥1 7 7 6 is 1776 24 60 60⊥1 2 3 is 3723
Encode	*A*⊤*A*	24 60 60⊤3723 is 1 2 3 60 60⊤3723 is 2 3
Execute	⍎*V*	⍎'1+2' is 3 ⍎'*P*' is 2 3 5 7
Format, monadic	⍕*A*	'¯1.5'∧.=⍕¯1.5 is 1 ⍴⍕*E* is 3 12 *X* is ⍕*X*
Format, dyadic	*V*⍕*A*	1⍕*P* is 4 1⍕*P* is 2.0 3.0 5.0 7.0 ¯1⍕*P* is 6 ¯1⍕*P* is 2*E*00 3*E*00 5*E*00 7*E*00

Notes to Figure 3.5

1. Restrictions on argument ranks are indicated by: *S* for scalar, *V* for vector, *M* for matrix, and *A* for any array. Except as the left argument of *S*⍳*A* or *S*[*A*], or the argument of ⍋ and ⍒, a scalar may be used in place of a vector. A one-element vector may replace any scalar.

2. Arrays used in examples:

```
      P
2 3 5 7

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

      X
ABCD
EFGH
IJKL
```

3. Function depends on index origin.

4. Elision of any index selects all along that axis.

5. The function is applied along the last axis; the symbols ⌿, ⍀, and ⊖ are equivalent to /, \, and ⌽, respectively, except that the function is applied along the first axis. In general, the relevant axis is determined by [*V*] after the function symbol.

Figure 3.5: Primitive Mixed Functions

1. A scalar may be used in place of a one-element vector:

   a. as left argument of

```
reshape          3⍴4     ↔   (,3)⍴4
take             3↑⍳5    ↔   (,3)↑⍳5
drop             3↓⍳5    ↔   (,3)↓⍳5
expand           1\,5    ↔   (,1)\,5
transpose        1⍉,5    ↔   (,1)⍉,5
format     5⍕3.2  ↔  (,5)⍕3.2  ↔  0 5 ⍕3.2
```

   b. as right argument of

```
execute          ⍎'P'    ↔   ⍎,'P'
branch           →4      ↔   →,4
```

2. A scalar is extended to conform to a vector:

   a. as left argument of

```
compress         1/ ⍳3 ↔     1 1 1 / ⍳3
rotate           1⌽2 2 ⍴⍳4   ↔  1 1 ⌽ 2 2⍴⍳4
```

   b. as right argument of

```
compress         1 0 1 / 2   ↔  1 0 1 / 2 2 2
expand           1 0 1 \ 2   ↔  1 0 1 \ 2 2
```

3. A one-element vector is permitted in place of a scalar:

   a. as left argument of

```
deal             (,3)?5 ↔    3?5
```

   b. as right argument of

```
index generator  ⍳,5     ↔   ⍳5
deal             3?,5    ↔   3?5
```

Figure 3.6: Scalar-Vector Substitutions for Mixed Functions

Figure 3.5 gives restrictions, in summary form, on the ranks of arguments which may be used with each mixed function. Figure 3.6 shows for what mixed functions and under what conditions scalar and vector arguments may be substituted for each other.

STRUCTURAL FUNCTIONS

In the monadic structure functions the argument may be of any type, numeric or character. In the dyadic selection and structure functions one argument may be of any type and the other (which serves as an index or other selection indicator) must be numeric, and in two cases (compression and expansion) is further restricted to be boolean.

Shape Reshape and Ravel

The monadic function ⍴ applied to an array *A* yields the shape of *A*, that is, a vector whose components are the dimensions of *A*. For

example, if A is the matrix

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

of three rows and four columns, then ρA is the vector `3 4`.

Since ρA contains one component for each axis of A, the expression $\rho\rho A$ is the rank of A. Figure 3.7 illustrates the values of ρA and $\rho\rho A$ for arrays of rank `0` (scalars) up to rank 3. In particular, the function ρ applied to a scalar yields an empty vector.

Type of Array	ρA	$\rho\rho A$	$\rho\rho\rho A$
Scalar		0	1
Vector	N	1	1
Matrix	$M\ N$	2	1
3-Dimensional	$L\ M\ N$	3	1

Figure 3.7 SHAPE AND RANK VECTORS

The monadic function ravel is denoted by a comma; when applied to any array A it produces a vector whose elements are the elements of A in row order. For example, if A is the matrix:

```
 2   4   6   8
10  12  14  16
18  20  22  24
```

and if $V\leftarrow,A$, then V is a vector of dimension `12` whose elements are the integers `2 4 6 8 10 12 ... 24`. If A is a vector, then $,A$ is equivalent to A; if A is a scalar, then $,A$ is a vector of length `1`.

The dyadic function ρ reshapes its right argument to the shape specified by its left argument. If $M\leftarrow D\rho V$, then M is an array of dimension D whose elements are the elements of V. For example, `2 3ρ1 2 3 4 5 6` is the matrix

```
1  2  3
4  5  6
```

If N, the total number of elements required in the array $D\rho V$, is equal to the dimension of the vector V, then the ravel of $D\rho V$ is equal to V. If N is less than ρV, then only the first N elements of V are used; if N is greater than ρV, then the elements of V are repeated cyclically. For example:

```
      2 3ρ1 2
1  2  1
2  1  2
```

```
      3 3ρ1 0 0 0
1  0  0
0  1  0
0  0  1
```

More generally, if A is any array, then $D\rho A$ is equivalent to $D\rho,A$. For example:

```
      A
1  2  3
4  5  6
```

```
      3 5ρA
1  2  3  4  5
6  1  2  3  4
5  6  1  2  3
```

The expressions `0ρX` and `0 3ρX` and `0 0ρX` are all valid; any one or more of the axes of an array may have zero length. Such an array is called an empty array. If D is an empty vector, then $D\rho A$ is a scalar.

Reverse and Rotate

The monadic function reverse is denoted by ⌽; if *X* is a vector and *K*←⌽*X*, then *K* is equal to *X* except that the items appear in reverse order. The axis operator applies to reversal and determines the axis along which the vectors are to be reversed. For example:

```
      A              ⌽[1]A          ⌽[2]A
1 2 3          4 5 6          3 2 1
4 5 6          1 2 3          6 5 4
```

The expression ⌽*A* denotes reversal along the last coordinate of *A*, and ⊖*A* denotes reversal along the first coordinate. For example, if *A* is of rank 3, then ⌽*A* is equivalent to ⌽[3]*A*, and ⊖*A* is equivalent to ⌽[1]*A*.

The dyadic function rotate is also denoted by ⌽. If *K* is a scalar or one-element vector and *X* is a vector, then *K*⌽*X* is a cyclic rotation of *X* defined as follows: *K*⌽*X* is equal to *X*[1+(⍴*X*)|¯1+*K*+⍳⍴*X*]. For example, if *X*←2 3 5 7 11, then 2⌽*X* is equal to 5 7 11 2 3, and ¯2⌽*X* is equal to 7 11 2 3 5. In 0-origin indexing, the definition for *K*⌽*X* becomes *X*[(⍴*X*)|*K*+⍳⍴*X*].

If the rank of *X* exceeds 1, then the coordinate *J* along which roatation is to be performed may be specified by the axis operator in the form *Z*←*K*⌽[*J*]*X*. Moreover, the shape of *K* must equal the remaining dimensions of *X*, and each vector along the *J*th axis of *X* is rotated as specified by the corresponding element of *K*. A scalar *K* is extended in the usual manner.

For example, if ⍴*X* is 3 4 and *J* is 2, then the shape of *K* must be 3 and *Z*[*I*;] is equal to *K*[*I*]⌽*X*[*I*;]. If *J* is 1, then ⍴*K* must be 4, and *Z*[;*I*] is equal to *K*[*I*]⌽*X*[;*I*]. For example:

```
      M                    0 1 2 3⌽[1]M          1 2 3⌽[2]M
1  2  3  4          1  6 11  4          2  3  4  1
5  6  7  8          5 10  3  8          7  8  5  6
9 10 11 12          9  2  7 12         12  9 10 11
```

The expression *K*⊖*X* denotes rotation along the first axis of *X*.

Catenate and Laminate

Catenate, denoted by a comma, chains vectors (or scalars) together to form a vector. For example:

```
      X←2 3 5 7 11
      X,X
2 3 5 7 11 2 3 5 7 11
```

In general, the dimension of *X*,*Y* is equal to the total number of elements in *X* and *Y*. A numeric vector cannot be catenated with a character vector.

The axis operator applies to catenate and determines the axis along which vectors are to be catenated. For example;

```
      M←3 3⍴'ABCDEFGHI'
      M                M,[1]M          M,[2]M          M,M
ABC              ABC             ABCABC          ABCABC
DEF              DEF             DEFDEF          DEFDEF
GHI              GHI             GHIGHI          GHIGHI
                 ABC
                 DEF
                 GHI
```

Two arrays are conformable for catenate along axis *I* if all other elements of their shapes agree. Moreover, two arrays may be catenated along axis *I* if they differ in rank by 1 and if the shape of the array of lower rank equals the shape of the array of higher rank after dropping item *I* from it. For example:

```
      V←'PQR'
      M,[1]V          M,[2]V          M,V
ABC             ABCP            ABCP
DEF             DEFQ            DEFQ
GHI             GHIR            GHIR
PQR
```

A scalar argument of catenate will be replicated to form a vector as required. For example:

```
      C←'⊡'
      C,(C,[1] M,[1] C),C
⊡⊡⊡⊡⊡
⊡ABC⊡
⊡DEF⊡
⊡GHI⊡
⊡⊡⊡⊡⊡
```

Laminate joins two arrays of identical rank and shape along a new axis. The position of the new axis relative to the existing axes is indicated by a fractional axis number. For example, if the new axis is to be inserted between the existing axes 1 and 2, the axis number must have a value between 1 and 2. If the new axis is to be inserted ahead of the present first axis, the axis number must be between 0 and 1 (or, if zero-origin indexing is used, between ¯1 and 0). Similarly, if the new axis is to be after the last of the present axes, the axis number must exceed the index of the present last axis by a fraction between 0 and 1.

The result of lamination has rank 1 greater than the rank of the arguments, and has the same shape except for the interpolation of the new axis, along which it has length 2. The comma, which normally denotes catenation, followed by an axis operator associated with a non-integral index produces lamination. For example:

```
      M←3 3⍴'ABCDEFGHI'
      N←3 3⍴'123456789'
      M             M,[.5]N          M,[1.5]N          M,[2.5]N
ABC             ABC             ABC             A1
DEF             DEF             123             B2
GHI             GHI                             C3
      N                         DEF
123             123             456             D4
456             456                             E5
789             789             GHI             F6

                                789             G7
                                                H8
                                                I9
```

The shapes of the foregoing laminations are 2 3 3 and 3 2 3 and 3 3 2; the position of the 2 shows the point of new axis insertion in each case.

A scalar argument of laminate is extended as required. For example:

```
      B←2 2ρ'1234'
      B,[2.5]'+'
1+
2+

3+
4+
      ,B,[2.5]'+'
1+2+3+4+
```

Transpose

The expression 2 1⍉*M* yields the transpose of the matrix *M*; that is, if *R*←2 1⍉*M*, then each element *R*[*I*;*J*] is equal to *M*[*J*;*I*]. For example:

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

If *P* is any permutation of the indices of the axes of an array *A*, then *P*⍉*A* is an array similar to *A* except that the axes are permuted: the *I*th axis becomes the *P*[*I*]th axis of the result. Hence, if *R*←*P*⍉*A*, then (ρ*R*)[*P*] is equal to ρ*A*. For example:

```
      A←2 3 5 7ρ⍳210
      ρA
2 3 5 7
      P←2 3 4 1
      ρP⍉A
7 2 3 5
```

More generally, *Q*⍉*A* is a valid expression if *Q* is any vector equal in length to the rank of *A* which is "complete" in the sense that if its items include any integer *N* they also include all positive integers less than *N*. For example, if ρρ*A* is 3, then 1 1 2 and 2 1 1 and 1 1 1 are suitable values for *Q* but 1 3 1 is not. Just as for the case *P*⍉*A* where *P* is a permutation, the *I*th axis becomes the *Q*[*I*]th axis of *Q*⍉*A*. However, in this case two or more of the axes of *A* may map into a single axis of the result, thus producing a diagonal section of *A* as illustrated below:

```
      A←3 3ρ⍳9              B←3 5ρ⍳15
      A                     B
1 2 3                 1  2  3  4  5
4 5 6                 6  7  8  9 10
7 8 9                11 12 13 14 15
      1 1⍉A                 1 1⍉B
1 5 9                 1  7 13
```

The monadic transpose ⍉*A* reverses the order of the axes of its argument. Formally, ⍉*A* is equivalent to (⌽⍳ρρ*A*)⍉*A*. In particular, for a matrix *A* this reduces to 2 1 ⍉*A* and is what is commonly called the transpose of a matrix.

SELECTION FUNCTIONS

The selection functions are all dyadic; one of the arguments may be an array of any type, and the other, which will be referred to as the selector because it serves to specify the selection to be made, must be numeric, and, in the case of expand and compress, is further restricted to boolean.

Take and Drop

If S is a non-negative scalar integer and V is a vector, then $S\uparrow V$ is a vector of dimension S obtained by taking the first S elements of V followed (if $S>\rho V$) by zeros if V is numeric and by spaces if it is not. For example:

```
      3↑2 3 5 7          7↑2 3 5 7
2 3 5                2 3 5 7 0 0 0
      3↑'ABCDE'          (7↑'ABCDE'),'⍈'
ABC                  ABCDE  ⍈
```

If S is a negative integer, then $S\uparrow V$ takes elements as above but takes the last elements of V and fills as needed on the left. For example:

```
      ¯3↑2 3 5 7         ¯7↑2 3 5 7
3 5 7                0 0 0 2 3 5 7
```

If A is any array, then $W\uparrow A$ is valid only if the vector W has one element for each axis of A, and $W[I]$ determines what is to be taken along the Ith axis of A. For example:

```
      A←3 4⍴⍳12
      A                2 3↑A          2 ¯3↑A          ¯2 6↑A
 1  2  3  4        1 2 3          2 3 4          5  6  7  8  0  0
 5  6  7  8        5 6 7          6 7 8          9 10 11 12  0  0
 9 10 11 12
```

The function drop (↓) is defined analogously, except that the indicated number of elements are dropped rather than taken. For example, ¯1 1↓A is the same matrix as the result of 2 ¯3↑A displayed in the preceding paragraph. If the number of elements to be dropped along any axis equals or exceeds the length of that axis, the resulting shape has a zero length for the axis.

The rank of the result of the take and drop functions is the same as the rank of the right argument.

Compress and Expand

The expression U/X denotes compression of X by U. If U is a boolean vector and X is a vector of the same dimension, then U/X produces a vector of $+/U$ elements chosen from those elements of X corresponding to non-zero elements of U. For example, if X←2 3 5 7 11 and U←1 0 1 1 0 then U/X is 2 5 7 and $(\sim U)/X$ is 3 11.

```
      C←'THIS IS AN EXAMPLE'
      D←C≠' '
      C1←D/C
      C1
THISISANEXAMPLE
```

To be conformable, the dimensions of the arguments must agree, except that a scalar argument is extended. Hence 1/X is equal to X and 0/X is an empty vector.

Expansion is the converse of compression and is denoted by $U\backslash X$. If $Y\leftarrow U\backslash X$, then U/Y is equal to X and $(\sim U)/Y$ is an array of zeros or spaces according as X is numeric or character. In other words, $U\backslash X$ expands X to the format indicated by the ones in U and fills in zeros or spaces. To be conformable, $+/U$ must equal ρX.

```
      D\C1
THIS IS AN EXAMPLE
```

The axis operator applies to both compress and expand and determines the axis along which they apply. If the axis operator is omitted, the last axis is used. The symbols $\neq$ and $\not\backslash$ also denote compression, but when used without an axis operator apply along the first axis.

```
      Q←3 4ρ'ABCDEFGHIJKL'
      Q
ABCD
EFGH
IJKL
      1 1 0 1\[1]Q
ABCD
EFGH

IJKL
      1 1 0 1⌿Q
ABCD
EFGH

IJKL
      0 1 1 0/Q
BC
FG
JK
      1 0 1/[1]Q
ABCD
IJKL
      1 0 1≠Q
ABCD
IJKL
```

The rank of the result of compress or expand equals the rank of the right argument.

Indexing

Indexing may be either 0-origin or 1-origin as discussed in Section 2. The following discussion assumes 1-origin. If X is a vector and I is a scalar, then $X[I]$ denotes the Ith element of X. For example, if $X\leftarrow 2\ 3\ 5\ 7\ 11$ then $X[2]$ is 3.

If the index I is a vector, then $X[I]$ is the vector obtained by selecting from X the elements indicated by successive components of I. For example, $X[1\ 3\ 5]$ is $2\ 5\ 11$ and $X[5\ 4\ 3\ 2\ 1]$ is $11\ 7\ 5\ 3\ 2$. If the elements of I do not belong to the set of indices of X, then the expression $X[I]$ evokes an index error report.

In general $\rho X[I]$ equals ρI. In particular, if I is a scalar, then $X[I]$ is a scalar, and if I is a matrix then $X[I]$ is a matrix. For example:

```
      A←'ABCDEFG'
      I←4 3ρ3 1 4 2 1 4 4 1 2 4 1 4

      I          A[I]
3 1 4        CAD
2 1 4        BAD
4 1 2        DAB
4 1 4        DAD
```

If M is a matrix, then M is indexed by a two-part list of the form $I;J$ where I selects the row (or rows) and J selects the column (or

columns). For example:

```
      M                    M[2;3]        M[1 3;2 3 4]
 1  2  3  4        7              2  3  4
 5  6  7  8                   10  11  12
 9 10 11 12
```

In general, $\rho M[I;J]$ is equal to $(\rho I),\rho J$. Hence if I and J are both vectors, then $M[I;J]$ is a matrix; if both I and J are scalars, $M[I;J]$ is a scalar; if I is a vector and J is a scalar (or vice versa), $M[I;J]$ is a vector. The indices are not limited to vectors, but may be of higher rank. For example, if I is a 3 by 4 matrix, and J is a vector of dimension 6, then $M[I;J]$ is of dimension 3 4 6, and $M[J;I]$ is of dimension 6 3 4. In particular, if T and P and Q are matrices, and if $R\leftarrow T[P;Q]$, then R is an array of rank 4 and $R[I;J;K;L]$ is equal to $T[P[I;J];Q[K;L]]$.

The form $M[I;]$ indicates that all columns are selected, and the form $M[;J]$ indicates that all rows are selected. For example, $M[2;]$ is 5 6 7 8 and $M[;2\ 1]$ is the matrix with rows 2 1 and 6 5 and 10 9.

The following example illustrates the use of a matrix indexing a matrix to obtain a three-dimensional array:

```
      M←2 4ρ3 1 4 2 1 4 4 1
      M                          M[;M]
3 1 4 2                    4 3 2 1
1 4 4 1                    3 2 2 3

                           4 1 1 4
                           1 1 1 1
```

An indexed variable may appear to the left of a specification arrow if (1) the expression is executable in the environment and (2) denoting the values of the expression on the left and right by L and R, then $1=\times/\rho R$ or $(1\neq\rho L)/\rho L$ must equal $(1\neq\rho R)/\rho R$. For example:

```
      X←2 3 5 7 11
      X[1 3]←6 8
      X
6  3  8  7  11
```

SELECTOR GENERATORS

All of the functions in this group have integer results which, although they are commonly useful as the selector argument in selection functions, are often used in other ways as well. For example, the grade function ⍋ is commonly used to produce indices needed to reorder a vector into ascending order (as in $X[⍋X]$), but may also be used in the treatment of permutations as the inverse function, that is, $⍋P$ yields the permutation inverse to P. Similarly, ιN generates a vector of N successive indices, but $.1\times\iota N$ generates a grid of values with an interval of .1.

Index Generator and Index of

The index generator ι applies to a non-negative scalar integer N to produce a vector of length N containing the first N integers in order, beginning with the value of the index origin $\square IO$. For example, $\iota 5$ yields 1 2 3 4 5 (in 1-origin) or 0 1 2 3 4 (in 0-origin), and $\iota 0$ yields an empty vector. A one-element vector argument is treated as a scalar.

If V is a vector and S is a scalar, then $V\iota S$ yields the index (in

the origin in force) of the earliest occurence of *S* in *V*, that is, the index of *S* in *V*. If *S* differs from all items of *V*, then *V*⍳*S* yields the first index outside the range of *V*, that is, ⎕*IO*+⍴*V*.

If *S* is any array, then *V*⍳*S* yields an array of the shape of *S*, each item being determined as the index in *V* of the corresponding item of *S*. For example:

```
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
      J←A⍳'HEAD CHIEF'
      J
8 5 1 4 27 3 8 9 5 6
      A[J]
HEAD CHIEF
      A[⍋J]
FEIHC DAEH
      A⍳'VAR3'
22 1 18 28
```

```
      M←2 5⍴'HEAD CHIEF'
      M
HEAD
CHIEF
      A⍳M
8  5  1  4 27
3  8  9  5  6
```

Membership

The function *X*∈*Y* yields a boolean array of the same shape as *X*. Any particular element of *X*∈*Y* has the value 1 if the corresponding element of *X* belongs to *Y*, that is, if it occurs as some element of *Y*. For example, (⍳7)∈3 5 is equal to 0 0 1 0 1 0 0 and 'ABCDEFGH'∈'COFFEE' equals 0 0 1 0 1 1 0 0. The right argument *Y* may be of any rank.

The selector argument of compression is commonly provided by application of the membership function, alone or in combination with the scalar boolean and relational functions.

Grade Functions

The grade up function ⍋*V* grades the items of vector *V* in ascending order, that is, it yields a result of the same dimension as *V* whose first item is the index (in the origin in force) of the smallest item of *V*, whose second item is the index of the next smallest item, and so on. Consequently, the expression *V*[⍋*V*] yields the elements of *V* in ascending order. For example, if *V*←8 3 7 5, then ⍋*V* is 2 4 3 1 and *V*[⍋*V*] is 3 5 7 8.

If the items of *V* are not all distinct, the ranking among any set of equal elements is determined by their position. For example, ⍋4 3 1 3 4 2 yields 3 6 2 4 1 5.

The grade down function ⍒*V* grades the items of *V* in descending order; among equal elements the ranking is determined by position just as for grade up. Consequently, ⍒*V* equals the reversal of ⍋*V* only if the items of *V* are distinct. For example:

```
      A←7 2 5 11 3
      ⍋A
2 5 3 1 4
      ⍒A
4 1 3 5 2
```

```
      B←4 3 1 3 4 2
      ⍋B
3 6 2 4 1 5
      ⍒B
1 5 2 4 6 3
```

The grade functions apply only to vectors.

Deal

The function *M*?*N* produces a vector of length *M* obtained by making *M* (pseudo-) random selections, without replacement, from the population

ιN. Both arguments are limited to scalars or one-element vectors. Each selection is made by appropriate application of the scheme described for the function roll.

The expression $N?N$ yields a random permutation of the items of ιN. The expression $P[M?\rho P]$ selects M distinct elements from the population defined by the items of a vector P. For example:

```
      P←'ABCDEFGH'
      P[3?⍴P]                          P[(⍴P)?⍴P]
GBD                                CEADHFGB
```

NUMERICAL FUNCTIONS

The numerical mixed functions apply only to numerical arguments and produce numerical results.

Matrix Inverse and Matrix Divide

The domino (⌹) represents two functions which are useful in a variety of problems including the solution of systems of linear equations, determining the projection of a vector on the subspace spanned by the columns of a matrix, and determining the coefficients of a polynomial which best fits a set of points in the least-squares sense.

When applied to a non-singular matrix A the expression $⌹A$ (matrix inverse) yields the inverse of A, and the expression $X←B⌹A$ (matrix divide) yields a value of X which satisfies the relation $\wedge/,B=A+.\times X$ and is therefore the solution of the system of linear equations conventionally represented as Ax=b:

```
      A←(ι4)∘.≥ι4
      A                ⌹A                A+.×⌹A
1 0 0 0          ¯1  0  0  0        1 0 0 0
1 1 0 0           1 ¯1  0  0        0 1 0 0
1 1 1 0           0  1 ¯1  0        0 0 1 0
1 1 1 1           0  0  1  1        0 0 0 1

      B←1 3 6 10
      X←B⌹A
      B                X                A+.×X          (⌹A)+.×B
1 3 6 10         1 2 3 4          1 3 6 10         1 2 3 4

      C←4 2⍴1 2 3 5 6 9 10 14
      Y←C⌹A
      C                Y                A+.×Y          (⌹A)+.×C
 1  2             1 2              1  2             1 2
 3  5             2 3              3  5             2 3
 6  9             3 4              6  9             3 4
10 14             4 5             10 14             4 5
```

The final example above shows that if the left argument is a matrix C, then $C⌹A$ yields a solution of the system of equations for each column of C.

If A is non-singular and if I is an identity matrix of the same dimension, then the matrix inverse $⌹A$ is equivalent to the matrix divide $I⌹A$. More generally, for any matrix P, the expression $⌹P$ is equivalent to the expression $((\iota R)\circ.=\iota R)⌹P$, where R is the number of rows in P.

The domino functions apply more generally to singular and non-square matrices, and to vectors and scalars; any argument of rank greater than 2 is rejected (*RANK ERROR*). For matrix arguments A and B

the expression $X\leftarrow B⌹A$ is executed only if

1. A and B have the same number of rows, and

2. the columns of A are linearly independent.

If the expression $X\leftarrow B⌹A$ is executable, then ρX is equal to $(1\downarrow\rho A),1\downarrow\rho B$ and X is determined so as to minimize the value of the expression $+/,(B-A+.\times X)*2$.

The domino functions apply to vector and scalar arguments as follows: except that the shape of the result is determined as specified above, a vector is treated as a one-column matrix and a scalar is treated as a one-by-one matrix. The rationale for this interpretation of a vector as a one-column (rather than one-row) matrix is that the right argument is treated geometrically (as will be seen in a subsequent example) as defining a space spanned by its column vectors, and the left argument was seen (in an earlier example) to be treated so as to yield a solution for each of its column vectors. Indeed, a one-rowed matrix right argument (unless 1-by-1) would be rejected under condition 2 above.

In the case of scalar arguments X and Y, the expression $⌹Y$ is equivalent to $\div Y$ and, except that it yields a domain error for the case $0⌹0$, the expression $X⌹Y$ is equivalent to $X\div Y$.

The use of ⌹ for a singular right argument can be illustrated as follows: if X is a vector and $Y\leftarrow F\ X$, then the expression $Y⌹X\circ.*0,\iota D$ yields the coefficients of the polynomial of degree D which best fits (in the least squares sense) the function F at the points X.

The definition of $B⌹A$ has certain useful geometric interpretations. If B is a vector and A is a matrix, then saying that $+/(B-A+.\times B⌹A)*2$ is a minimum is equivalent to saying that the length of the vector $B-A+.\times B⌹A$ is a minimum. But $A+.\times B⌹A$ is a point in the space spanned by the column vectors of A and is therefore the point in this space which is closest to B. In other words, $P\leftarrow A+.\times B⌹A$ is the projection of B on the space spanned by the columns of A. Moreover, the vector $B-P$ must be normal to every vector in the space; in particular, $(B-P)+.\times A$ is a zero vector.

If A and B are single-column matrices, then $B⌹A$ is a 1 by 1 matrix and $A+.\times B⌹A$ is equivalent to $A\times S$, where S is the scalar $''\rho B⌹A$. If A and B are vectors, then $B⌹A$ is a scalar and the projection of B on A is therefore given by the simpler expression $A\times B⌹A$. For example:

```
      A←4.5 1.7
      B←2 5
      P←A×B⌹A
      P
3.403197926 1.28565255
      N←B-P
      N
¯1.403197926 3.71434745
      N+.×A
2.442490654E¯15
```

Similar analysis shows that if A is a vector then $⌹A$ is a vector in the direction of A; that is, $⌹A$ is equal to $S\times A$ for some scalar S. Moreover, $A+.\times⌹A$ is equal to 1. In other words, $⌹A$ is the "image" of

the vector A obtained by inversion in the unit circle (or sphere).

Decode and Encode

For vectors R and X, the decode (or base-value) function $R\perp X$ yields the value of the vector X evaluated in a number system with radices $R[1],R[2],\ldots,R[\rho R]$. For example, if $R\leftarrow$`24 60 60` and $X\leftarrow$`1 2 3` is a vector of elapsed time in hours, minutes, and seconds, then $R\perp X$ has the value `3723`, and is the corresponding elapsed time in seconds. Similarly, `10 10 10 10 ⊥ 1 7 7 6` is equal to `1776`, and `2 2 2 ⊥ 1 0 1` is equal to `5`. Formally, $R\perp X$ is equal to $+/W\times X$, where W is the weighting vector determined as follows: $W[\rho W]$ is equal to `1` and $W[I-1]$ is equal to $R[I]\times W[I]$. For example, if R is `24 60 60`, then W is `3600 60 1`.

Scalar (or one-element vector) arguments are treated as usual. For example, `10 ⊥ 1 7 7 6` yields `1776`. The arguments are not restricted to integers; for example, if X is a scalar then $X\perp C$ is the value of the polynomial with coefficients C, arranged in descending order on the powers of X.

The decode function is extended to arrays in the manner of the inner product operator: each of the radix vectors along the last axis of the first argument is applied to each of the vectors along the first axis of the second argument. There is one difference; if either of these distinguished axes is of length `1`, it will be extended (by replication of the element) as necessary to match the length of the other argument. Except for this different treatment of unit axes, the shape of the result of $A\perp B$ is determined like the shape of the inner product, namely $(^{-}1\downarrow\rho A),1\downarrow\rho B$.

The encode (or representation) function $R\top X$ is, for certain arguments, inverse to the decode function. For example:

```
      R←10 10 10 10
      R⊥1 7 7 6
1776
      R⊤1776
1 7 7 6
```

More generally, $R\perp(R\top X)$ equals $(\times/R)|X$ rather than X. For example:

```
      10 10 10 10⊤123          10 10 10⊤123
0 1 2 3                  1 2 3

      10 10⊤123                10⊤123
2 3                      3
```

More precisely, the definition of the encode function is based on the definition of the residue function; for a vector left argument and scalar right argument, decode is equivalent to the E whose canonical representation is shown at the left below:

```
Z←A E B                        2 2 2⊤13
Z←0×A                    1 0 1
I←ρA                           ¯2 ¯2 ¯2⊤13
L:→(I=0)/0               ¯1 ¯1 ¯1
Z[I]←A[I]|B                    2 0 2⊤13
→(A[I]=0)/0              0 6 1
B←(B-Z[I])÷A[I]                2 2 2⊤¯13
I←I-1                    0 1 1
→L                             ¯2 2 ¯2⊤13
                         0 1 ¯1
```

The basic definition of $R\top X$ concerns a vector R and a scalar X, and produces a result of the shape of R. It is extended to arrays as follows: each radix vector along the first axis of R is applied to obtain the representation of each item of X, the resulting representations being arrayed along the first axis of the result. For example:

```
      10 10 10⊤215 486 72 219 3
2 4 0 2 0
1 8 7 1 0
5 6 2 9 3
```

```
      R←10 10 10,[1.5]8 8 8
      R
10 8
10 8
10 8
      R⊤123
1 1
2 7
3 3
```

The expression for the shape of the result of $R\top X$ is the same as for the shape of the outer product, namely $(\rho R),\rho X$.

DATA TRANSFORMATIONS

Of the two functions in this class, the format is a true type transformation, being designed to produce a character array which represents the data in its numeric argument. Over a certain class of arguments the execute function is inverse to the format and is therefore considered as a type transformation as well, although its applicability is, in fact, much broader.

Execute and Format

Any character vector or scalar can be regarded as a representation of an APL statement (which may or may not be well-formed). The monadic function denoted by ⍎ takes as its argument a character vector or scalar and evaluates or executes the APL statement it represents. When applied to a character array that might be construed as a system command or the opening of function definition, an error will necessarily result when evaluation is attempted, because neither of these is a well-formed APL statement.

The execute function may appear anywhere in a statement, but it will successfully evaluate only valid (complete) expressions, and its result must be at least syntactically acceptable to its context. Thus, execute applied to a vector that is empty, contains only spaces, or starts with → (branch symbol) or ⍝ (comment symbol) produces no explicit result and therefore can be used only on the extreme left. For example:

```
      ⍎''
      Z←⍎''
VALUE ERROR
      Z←⍎''
      ∧
```

The domain of ⍎ is any character array of rank less than two, and *RANK* and *DOMAIN* errors are reported in the usual way:

```
      C←'3 4'
      +/⍎C
7
      ⍎1 3ρC
RANK ERROR
      ⍎ 1 3 ρC
      ∧
```

```
      ⍎3 4
DOMAIN ERROR
      ⍎3 4
      ∧
```

An error can also occur in the attempted execution of the APL expression represented by the argument of ⍎; such an indirect error is reported by the error type prefaced by the symbol ⍎ and followed by the character string and the caret marking the point of difficulty. For example:

```
      ⍎'4÷0'
⍎ DOMAIN ERROR
      4÷0
       ^
      ⍎')WSID'
⍎ VALUE ERROR
      )WSID
       ^
```

The symbol ⍕ denotes two format functions which convert numerical arrays to character arrays. There are several significant uses of these functions in addition to the obvious one for composing tabular output. For example, the use of format is complementary to the use of execute in treating bulk input and output, and in the management of combined alphabetic and numeric data.

The monadic format function produces a character array which will display identically to the display normally produced by its argument, but makes this character array explicitly available. For example:

```
      M←2=?4 4⍴2
      R←⍕M
      M                    R                    R[;2×⍳4]
 0 1 0 1              0 1 0 1              0101
 0 0 1 1              0 0 1 1              0011
 1 0 1 1              1 0 1 1              1011
 0 0 1 1              0 0 1 1              0011
      ⍴M                   ⍴R
4 4                  4 8

      ⍴⍕2 5                X←34
3                          'THE VALUE OF X IS ',⍕X
                     THE VALUE OF X IS 34
      ∧/,R=⍕R
1
      ⍕'ABCD'
ABCD
```

The format function applied to a character array yields the array unchanged, as illustrated by the last two examples above. For a numerical array, the shape of the result is the same as the shape of the argument except for the required expansion along the last coordinate, each number going, in general, to several characters. The format of a scalar number is always a vector.

The printing normally produced by APL systems may vary slightly from system to system, but the result produced by the monadic format will have no final column of all spaces, and no initial spaces in the case of a vector or scalar argument.

The dyadic format function accepts only numerical arrays as its right argument, and uses variations in the left argument to provide progressively more detailed control over the result. Thus, for *F*⍕*A*, the argument *F* may be a single number, a pair of numbers, or a vector of length 2×¯1↑1,⍴*A*.

In general, a pair of numbers is used to control the result: the first determines the total width of a number field, and the second sets the precision. For decimal form the precision is specified as the

number of digits to the right of the decimal point, and for scaled form it is specified as the number of digits in the multiplier. The form to be used is determined by the sign of the precision indicator, negative numbers indicating scaled form. Thus:

```
      ⍴⍞←A
 12.34      ¯34.567
  0          12
 ¯0.26      ¯123.45
3 2
      R←9 2⍕A
      S←9 ¯2⍕A
      ⍴⍞←R
 12.34    ¯34.57
  .00      12.00
 ¯.26    ¯123.45
3 18
      ⍴⍞←S
  1.2E01  ¯3.5E01
 0.0E00    1.2E01
¯2.6E¯01  ¯1.2E02
3 18
```

```
      ⍴⍞←12 3⍕A
 12.340      ¯34.567
  .000        12.000
 ¯.260      ¯123.450
3 24
      ⍴⍞←6 0⍕A
 12    ¯35
  0     12
  0   ¯123
3 12
      ⍴⍞←7 ¯1⍕A
  1E01   ¯3E01
 0E00    1E01
¯3E¯01  ¯1E02
3 14
```

If the width indicator of the control pair is zero, a field width is chosen such that at least one space will be left between adjacent numbers. If only a single control number is used, it is treated like a number pair with a width indicator of zero:

```
      ⍴⍞←2⍕A
 12.34   ¯34.57
  .00     12.00
 ¯.26   ¯123.45
3 16
      ⍴⍞←0 2⍕A
 12.34   ¯34.57
  .00     12.00
 ¯.26   ¯123.45
3 16
```

```
      ⍴⍞←¯2⍕A
  1.2E01   ¯3.5E01
 0.0E00     1.2E01
¯2.6E¯01  ¯1.2E02
3 18
      ⍴⍞←0 ¯2⍕A
  1.2E01   ¯3.5E01
 0.0E00     1.2E01
¯2.6E¯01  ¯1.2E02
3 18
```

Each column of an array can be individually composed by a left argument that has a control pair for each:

```
      ⍴⍞←0 2 0 2⍕A
 12.34   ¯34.57
  .00     12.00
 ¯.26   ¯123.45
3 15
      ⍴⍞←6 2 12 ¯3⍕A
 12.34    ¯3.46E01
  .00      1.20E01
 ¯.26     ¯1.23E02
3 18
```

```
      ⍴⍞←8 3 0 2⍕A
 12.340   ¯34.57
  .000     12.00
 ¯.260   ¯123.45
3 16
      ⍴⍞←8 0 0 ¯2⍕A
  12  ¯3.5E01
   0   1.2E01
   0  ¯1.2E02
3 17
```

```
      6 2 8 3 3 0 4 0 5 0 12 4⍕,A
 12.34 ¯34.567  0  12    0   ¯123.4500
```

The format function applied to an array of rank greater than two applies to each of the planes defined by the last two axes. For example:

```
      L←2=?2 2 5ρ2
      L
1 1 0 0 1
1 1 1 0 1

1 0 0 1 0
0 0 0 0 0

      4 1⍉L
1.0 1.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0

1.0 0.0 0.0 1.0 0.0
0.0 0.0 0.0 0.0 0.0
```

Tabular displays incorporating row and column headings, or other information between columns or rows, are easily configured using the format function together with catenation. For example:

```
      ROWHEADS←4 3ρ'JANAPRJULOCT'
      YEARS←71+ι5
      TABLE←.001×¯4E5+?4 5ρ8E5
      (' ',[1]ROWHEADS),(2ϕ9 0⍉YEARS),[1]9 2⍉TABLE
       72      73      74      75      76
JAN   318.13  ¯351.55    3.62  ¯144.77  ¯4.82
APR   ¯327.41 ¯341.00  ¯92.69   331.05  ¯28.44
JUL   ¯359.93  216.16  ¯299.71  150.77  103.64
OCT    180.33  310.86  ¯154.94   10.62  276.79
```

There are obvious restrictions on the left argument of format, since the width of a field must be large enough to hold the requested form, and if the specified width is inadequate the result will be a *DOMAIN* error. However, the width need not provide open spaces between adjacent numbers. For example, boolean arrays can be tightly packed:

```
      1 0⍉2=?4 4ρ2
1001
0000
1101
0111
```

The following formal characteristics of the format function need not concern the general user, but may be of interest in certain applications:

The least width needed for a column of numbers *C* with precision *P* is `W←(∨/R<0)+(~Pϵ0 ¯1)+(|P)+(4,⌈/0,(R≠0)+⌊10⍟|R+R=0)[1+P≥0]`, where *R* is the rounded value of *C* given by `R←(⌊.5+C×10*|P)÷10*|P`.

The expressions `(M⍉A),N⍉B` and `(M,N)⍉A,B` are equivalent if *M* and *N* are full control vectors, that is, if `((ρM)=2×¯1↑ρA)∧(ρN)=2×¯1↑ρB`. If `2=ρM`, then `(M⍉A),M⍉B` and `M⍉A,B` are equivalent.

Although the primitive functions of APL deal only with abstract objects (arrays of numbers and characters), it is often desirable to bring the power of the language to bear on the management of the concrete resources or the environment of the system in which APL operates. This can be done within the language by identifying certain variables as elements of the interface between APL and its host system, and using these variables for communication between them. While still abstract objects to APL, the values of such system variables may have any required concrete significance to the host system.

In principle all necessary interaction between APL and its environment could be managed by use of a complete set of system variables, but there are situations where it is more convenient, or otherwise more desirable, to use functions based on the use of system variables which may not themselves be made explicitly available. Such functions are called, by analogy, system functions.

System variables and system functions are denoted by distinguished names that begin with a quad (⎕). The use of such names is reserved for the system and cannot be applied to user-defined objects. They cannot be copied, grouped, or erased; those that denote system variables can appear in function headers, but only to be localized (see Section 6). Within APL statements, distinguished names are subject to all the normal rules of syntax.

SYSTEM FUNCTIONS

Like the primitive abstract functions of APL, the system functions are available throughout the system, and can be used in defined functions. They are monadic or dyadic, as appropriate, and have explicit results. In most cases they also have implicit results, in that their execution causes a change in the environment. The explicit result always indicates the status of the environment relevant to the possible implicit result.

Altogether, 13 system functions are provided. Six of these are concerned with the management of the shared-variable facility and are described in Section 5. The other seven are given in Figure 4.1, and are described here.

CANONICAL REPRESENTATION

The canonical representation of a defined function as defined in Section 6 is obtained as a result of applying the system function ⎕*CR* to the character array representing the name of the function. Applied to any argument which does not represent the name of an unlocked defined function it yields a matrix of dimension 0 by 0. Possible error reports for ⎕*CR* are *RANK* error if the argument is not a vector or a scalar, or *DOMAIN* error if the argument is not a character array. The use of ⎕*CR* is further described in Section 6.

FUNCTION ESTABLISHMENT

The definition of a function can be established or fixed by applying the system function ⎕*FX* to its character representation. The function ⎕*FX* produces as an explicit result the array of characters which represents the name of the function being fixed, while replacing any existing definition of a function with the same name.

FUNCTION	REQUIREMENTS			EFFECT ON ENVIRONMENT	EXPLICIT RESULT
	RANK	LENGTH	DOMAIN		
$\square CR\ A$	$1\geq\rho\rho A$		Array of characters.	None.	Canonical representation of object named by *A*. The result for anything other than an unlocked defined function is of size 0 0.
$\square FX\ M$	$2=\rho\rho M$		Matrix of characters.	Fix (establish) definition of the function represented by *M*, unless its name is already in use for an object other than function which is not halted.	A vector representing the name of the function established, or the scalar row index of the fault which prevented establishment.
$\square EX\ A$	$2\geq\rho\rho A$		Array of characters.	Expunge (erase) objects named by rows of *A*, except groups, labels, or halted functions.	A boolean vector whose *I*th element is 1 if the *I*th name is now free.
$\square NL\ N$	$1\geq\rho\rho N$	$1\geq\rho, S$	$\wedge/N\epsilon 1\ 2\ 3$	None.	A matrix of rows (in accidental order) representing names of designated kinds in the dynamic environment: 1, 2, 3 for labels, variables, functions.
$A\ \square NL\ N$	$1\geq\rho\rho N$		$\wedge/N\epsilon 1\ 2\ 3$ Elements of *A* must be alphabetic.	None.	As for the monadic form, except that only names beginning with letters in *A* will be included.
$\square NC\ A$	$2\geq\rho\rho M$		Array of characters.	None.	A vector giving the usage of the name in each row of *A*: 0 name is available 1 label 2 variable 3 function 4 other
$\square DL\ S$	$1\geq\rho\rho S$		Numeric value.	None, but requires *S* seconds to complete.	Scalar value of actual delay.

Figure 4.1: System Functions

An expression of the form $\square FX\ M$ will establish a function if both the following conditions are met:

1. *M* is a valid representation of a function. Any matrix which differs from a canonical matrix only in the addition of non-significant spaces (other than rows consisting of spaces only) is a valid representation.

2. The name of the function to be established does not conflict with an existing use of the name for a halted function (defined in Section 7) or for a label, group, or variable.

If the expression fails to establish a function then no change occurs in the workspace and the expression returns a scalar index of the row in the matrix argument where the fault was found. If the argument of $\square FX$ is not a matrix a *RANK* error will be reported, and if it is not a character array a *DOMAIN* error will result.

EXPUNGE

Certain name conflicts can be avoided by using the expunge function $\square EX$ to dynamically eliminate an existing use of a name. Thus $\square EX$ *'PQR'* will erase the object *PQR* unless it is a label, a group, or a

halted function. The function returns an explicit result of 1 if the name is now unencumbered, and a result of 0 if it is not, or if the argument does not represent a well-formed name. The expunge function applies to a matrix of names and then produces a logical vector result. ⎕*EX* will report a *RANK* error if its argument is of higher rank than a matrix, or a *DOMAIN* error if the argument is not a character array. A single name may also be presented as a vector or scalar.

NAME LIST

The dyadic function ⎕*NL* yields a character matrix, each row of which represents the name of an object in the dynamic environment. The right argument is an integer scalar or vector which determines the class of names produced as follows: 1, 2, and 3 respectively invoke the names of labels, variables, and functions. The left argument is a scalar or vector of alphabetic characters which restricts the names produced to those with an initial letter occurring in the argument. The ordering of the rows of the result is fortuitous.

The monadic function ⎕*NL* behaves analogously with no restriction on initial letters. For example, ⎕*NL* 2 produces a matrix of all variable names, and either of ⎕*NL* 2 3 or ⎕*NL* 3 2 produces a matrix of all variable and function names.

The uses of ⎕*NL* include the following:

In conjunction with ⎕*EX*, all the objects of a certain class can be dynamically erased; or a function can be readily defined that will clear a workspace of all but a preselected set of objects.

In conjunction with ⎕*CR*, functions can be written to automatically display the definitions of all or certain functions in the workspace, or to analyze the interactions among functions and variables.

The dyadic form of ⎕*NL* can be used as a convenient guide in the choice of names while designing or experimenting with a workspace.

NAME CLASSIFICATION

The monadic function ⎕*NC* accepts a matrix of characters and returns a numerical indication of the class of the name represented by each row of the argument. A single name may also be presented as a vector or scalar.

The result of ⎕*NL* is a suitable argument for ⎕*NC*, but other character arrays may also be used, in which case the possible results are integers ranging from 0 to 4. The significance of 1, 2, and 3 are as for ⎕*NL*; a result of 0 signifies that the corresponding name is available for any use; a result of 4 signifies that the argument is not available for use as a name. The latter case may arise because the name is in use for denoting a group, or because the argument is a distinguished name or not a valid name at all.

DELAY

The delay function, denoted by ⎕*DL*, evokes a pause in the execution of the statement in which it appears. The argument of the function determines the duration of the pause, in seconds, but the accuracy is limited by possible contending demands on the system at the moment of release. Moreover, the delay can be aborted by a strong

interrupt. The explicit result of the delay function is a scalar value equal to the actual delay. If the argument of ⎕*DL* is not a scalar with a numerical value, a *RANK* or *DOMAIN* error will be reported.

Generally speaking, the delay function uses only a negligible amount of computer time (as opposed to connect time). It can therefore be used freely in situations where repeated tests may be required at intervals to determine whether an expected event has taken place. This is useful in work with shared variables (as in the function *OFFER* given as an example in Section 5), as well as in certain kinds of interactions between users and programs.

SYSTEM VARIABLES

System variables are instances of shared variables, which are treated in Section 5. The characteristics of shared variables that are most significant here are these:

1. If a variable is shared between two processors, the value of the variable when used by one of them may well be different from what that processor last specified, and

2. each processor is free to use or not use a value specified by the other, according to its own internal workings.

System variables are shared between a workspace and the APL processor. Sharing takes place automatically each time a workspace is activated and, when a system variable is localized in a function, each time the function is used.

The system variables are listed in Figure 4.2, which gives their significance and use. Two classes can be discerned:

1. Comparison tolerance, index origin, latent expression, random link, printing precision, and printing width. In these cases the value specified by the user (or available in a clear workspace) is used by the APL processor during the execution of operations to which they relate. If this value is inappropriate, or if no value has been specified after localization, an *IMPLICIT* error will be evoked at the time of execution.

2. Account information, atomic vector, line counter, time stamp, terminal type, user load, and work area. In these cases localization or setting by the user are immaterial. The APL processor will always reset the variable before it can be used again.

The APL statement represented by the latent expression is automatically executed whenever the workspace is activated. Formally, ⎕*LX* is used as an argument to the execute function (⍎⎕*LX*), and any error message will be appropriate to the use of that function.

Common uses of the latent expression include the form ⎕*LX*←'*G*' used to invoke an arbitrary function *G*, the form:

⎕*LX*←'''*FOR NEW FEATURES IN THIS WS ENTER*: *NEW*'''

used to print a message upon activation of the workspace, and the form ⎕*LX*←'→⎕*LC*' used to automatically restart a suspended function. The variable ⎕*LX* may also be localized within a function and respecified therein to furnish a different latent expression when the function is

NAME	PURPOSE	VALUE IN CLEAR WS	MEANINGFUL RANGE
□*CT*	Comparison tolerance: used in monadic ⌈ ⌊, dyadic < ≤ = ≥ > ≠ ∈ ⍳	1E¯13	0-1
□*IO*	Index origin: used in indexing and in ? ⍳ ⍋ ⍒ ⌽ □*FX*	1	0 1
□*LX*	Latent expression executed on activation of workspace	''	characters
□*PP*	Printing precision: affects numeric output and monadic ⍕	10	⍳16
□*PW*	Printing width: affects all but bare output and error reports	(system dependent)	
□*RL*	Random link: used in ?	7*5	⍳¯1+2*31
□*AI*	Account information: identification, computer time, connect time, keying time (all times in milliseconds and cumulative during session)		
□*AV*	Atomic vector		
□*LC*	Line counter: statement numbers of functions in execution or halted, most recently activated first	⍳0	
□*TS*	Time stamp: year, month, day (of month), hour (on 24-hour clock), minute, second, millisecond		
□*TT*	Terminal type: 1 for Selectric, 2 for PTTC/BCD, 3 for 1050		
□*UL*	User Load		
□*WA*	Working area available (in bytes)		

Figure 4.2: System Variables

suspended. For example:

```
      □LX←'F'

      ∇ F;□LX
[1]   □LX←'→□LC,ρ□←''WE CONTINUE FROM WHERE WE LEFT OFF'''
[2]   'WE NOW BEGIN LESSON 2'
[3]   DRILLFUNCTION ∇

      )SAVE ABC
```

On the first activation of workspace *ABC*, the function *F* would be automatically invoked; if it were later saved with *F* halted, subsequent activation of the workspace would automatically continue execution from the point of interruption.

The atomic vector □*AV* is a 256-element character vector, containing all possible characters. Certain elements of □*AV* may be terminal control characters, such as carrier return or linefeed, but other elements of □*AV* may neither print nor exercise control. The indices of any known characters can be determined by an expression such as □*AV*⍳'*ABCABC*'.

SECTION 5: SHARED VARIABLES

Two otherwise independent concurrently operating processors can communicate, and thereby be made to cooperate, if they share one or more variables. Such shared variables constitute an interface between the processors, through which information may be passed to be used by each for its own purposes. In particular, variables may be shared between two active APL workspaces, or between an APL workspace and some other processor that is part of the overall APL system, to achieve a variety of effects including the control and utilization of devices such as printers, card readers, magnetic tape units, and magnetic disk storage units.

In use in an APL workspace, a shared variable may be either global or local, and is syntactically indistinguishable from ordinary variables. It may appear to the left of an assignment, in which case its value is said to be set, or elsewhere in a statement, where its value is said to be used. Either form of reference is an access.

At any instant a shared variable has only one value, that last assigned to it by one of its owners. Characteristically, however, a processor using a shared variable will find its value different from what it might have set earlier.

A given processor can simultaneously share variables with any number of other processors. However, each sharing is bilateral; that is, each shared variable has only two owners. This restriction does not represent a loss of generality in the systems that can be constructed, and commonly useful arrangements are easily designed. For example, a shared file can be made directly accessible to a single control processor which communicates bilaterally with (or is integral with) the file processor itself. In turn, the central processor shares variables bilaterally with each of the using processors, controlling their individual access to the data, as required.

It was noted in Section 4 that system variables are instances of shared variables in which the sharing is automatic. It was not pointed out, however, that access sequence disciplines are also imposed on certain of these variables, although one effect of this was noted; namely, variables like the time stamp accept any value specified, but continue to provide the proper information when used. The discipline that accomplishes this effect is an inhibition against two successive accesses to the variable unless the sharing processor (the system) has set it in the interim.

When ordinary, "undistinguished", variables are to be shared, explicit actions are necessary to effect the sharing and establish a desired access discipline. Six system functions are provided for these purposes: three for the actual management and three to provide related information. These are summarized in Figure 5.1.

OFFERS

A single offer to share is of the form *P* □*SVO N*, where *P* is the identification of another processor and *N* is a character vector representing a pair of names. The first of this pair is the name of the variable to be shared, and the second is a surrogate name which is offered to match a name offered by the other processor. The name of the variable may be its own surrogate, in which case only the one name need be used, rather than two. For example, the three sets of actions shown below all have the same effect, which is to share one variable between two processors 1234 and 5678, the variable being known to the former as

FUNCTION	REQUIREMENTS [1] RANK	LENGTH	DOMAIN	EFFECT ON ENVIRONMENT	EXPLICIT RESULT
P'⎕*SVO N*	2≥ρρ*N*	(×/ρ*P*)∈1,¯1↓ρ*N*	*P*∈¯1+⍳2*31 [2]	Tenders offer to processor *P* if first (or only) name of a pair is not previously offered and not already in use as the name of an object other than a variable. The offer is general (to anyone) if 0=*P*.	Degree of coupling now in effect for the name pair. Dimension: ×/¯1↓ρ*N*.
⎕*SVO N*	2≥ρρ*N*	None.	[2]	None.	Degree of coupling now in effect for the name pair. Dimension: ×/¯1↓ρ*N*.
C ⎕*SVC N*	2≥ρρ*N* 2≥ρρ*C*	(1≥ρρ*C*)∧1=×/ρ*C* or (ρ*C*)=(¯1↓ρ*N*),4	∧/*C*∈0 1 [2]	Sets access control.	New setting of access control. Dimension: (¯1↓ρ*N*),4.
⎕*SVC N*	2≥ρρ*N*	None.	[2]		Existing access control.
⎕*SVR N*	2≥ρρ*N*	None.	[2]	Retracts offer (ends sharing).	Degree of coupling before this retraction. Dimension: ×/¯1↓ρ*N*.
⎕*SVQ P*	1≥ρρ*P*	1≥ρ,*P*	*P*∈¯1+⍳2*31	None.	If 0=ρ*P*: Vector of identifications of processors making offers to this user. If 1=×/ρ*P*: Matrix of names offered by processor *P* but not yet shared.

NOTES:
1. If a requirement is not met the function is not executed and a corresponding error report is printed.
2. Each row of *N* (or *N* itself if 2≥ρρ*N*) must represent a name or pair of names. If a pair of names is used for an offer (dyadic ⎕*SVO*), either the pair, or the first name only, can be used for the other functions.

Figure 5.1: Functions for the Management of Sharing

ABC, and to the latter as *Q*.

```
          User 1234                                  User 5678

          5678 ⎕SVO 'ABC Y'
1                                                                 1234 ⎕SVO 'Q Y'
                                                                2
          5678 ⎕SVO 'ABC Q'
1                                                                 1234 ⎕SVO 'Q'
                                                                2
          5678 ⎕SVO 'ABC'
1                                                                 1234 ⎕SVO 'Q ABC'
                                                                2
```

The surrogate names have no effect other than to control the matching, making it possible for one processor to operate with no direct knowledge of, or concern with, the variable name used by the other. The same surrogate can be used in a succession of offers to the same processor, in which case they are matched in sequence by appropriate counter-offers. The same surrogate may also be used for offers to any number of other processors at the same time. However, since a variable may be offered to (or shared with) only one other processor at a time, each coincident use of a particular surrogate name must be associated with a different variable name.

The explicit result of the expression *P* ⎕*SVO N* is the degree of coupling of the name or name pair in *N*: zero if no offer has been made, one if an offer has been made but not matched, two if sharing is completed. An offer to any processor (other than the offering processor itself) increases the coupling of the name offered if the name has zero coupling and is not the name of a label, function, or group. An offer never decreases the coupling.

The monadic function ⎕*SVO* does not affect the coupling of the name represented by its argument, but does report the degree of coupling as its explicit result. If the degree of coupling is one or two, a repeated offer has no further implicit result and either monadic or dyadic ⎕*SVO* may be used for inquiry. Advantage is taken of this in the following example of a defined function for offering a name (to be entered on request) to a processor *P* for a period of *T* seconds:

```
Z←P OFFER T;I;Q
⍞←'NAME: '
→(' '∧.=Q←⍞)/Z←I←0
L1:→(2=Z←P ⎕SVO Q)/L2
→(T≥I←I+⌶+0×⎕DL 1)/L1
'NO DEAL'
→0
L2:'ACCEPTED' ∇
```

If the arguments of ⎕*SVO* fail to meet any of the basic requirements listed in Figure 5.1, the appropriate error report is evoked and the function is not executed. If a user attempts to share more variables than the quota allotted to him by those responsible for the general management of the system the error report will be *INTERFACE QUOTA EXHAUSTED*, and if, for any reason, the shared variable facility itself is not available the report will be *NO SHARES*. An offer to a processor will be tendered, whether or not the processor happens to be available.

The value of a shared variable when sharing is first completed is determined thus: if both owners had assigned values beforehand, the value is that assigned by the first to have offered; if only one owner had, that value obtains; if neither had, the variable has no value. Names used in sharing are subject to the usual rules of localization.

A set of offers can be made by using a vector left argument (or a scalar or one-element vector which is automatically extended) and a matrix right argument, each of whose rows represents a name or name pair. The offers are then treated in sequence and the explicit result is the vector of the resulting degrees of coupling. If the quota of shared variables is exhausted in the course of such a multiple offer, none of the offers will be tendered.

An offer made with zero as left argument is a general offer, that is, an offer to any processor. A general offer will be matched only with a counter-offer which is not general, that is, one that explicitly identifies the processor making the general offer. The processor identification associated with a user is the user's sign-on number. Auxiliary processors are usually identified by numbers between 1 and 1000.

ACCESS CONTROL

Consider the following simple example of sharing the variable *V* between two users 1234 and 5678:

```
      User 1234                          User 5678

      5678 □SVO 'V'
1                                              1234 □SVO 'V'
                                       2
      V←5
                                               V←3×V*2
      V
75
```

The relative sequence of events in the two workspaces, after sharing, is significant; for example, had the use of *V* by 1234 in the foregoing example preceded the setting by 5678, the resulting value would have been 5 rather than 75.

In most practical applications it is important to know that a new value has been assigned between successive uses of a shared variable, or that use has been made of an assigned value before a new one is set. Since, as a practical matter, this cannot be left to chance, an access control mechanism is embodied in the shared variable facility.

The access control operates by inhibiting the setting or use of a shared variable by one owner or the other, depending upon the access state of the variable, and the value of an access control matrix which is set jointly by the two owners, using the dyadic form of the system function □*SVC*. If, in the example above, one user (say 5678) had followed his offer to share *V* by the expression `1 1 1 1 □SVC 'V'`, then the desired sequence would have been enforced. That is, the use of *V* by 5678 would be automatically delayed until *V* is set by 1234, and the use by 1234 would be delayed until *V* is set by 5678.

The delay occasioned by the inhibition of any access uses only a negligible amount of computer time. Interruption by a strong interrupt signal during the period of delay aborts the access and unlocks the keyboard.

Figure 5.2 shows the three access states possible for a shared variable, the possible transitions between states, and the potential inhibitions imposed by the access control matrix, *ACM*. The first row of *ACM* is associated with setting of the variable by each owner, and the second with its use. The permissible operations for any state are indicated by the zeros in *ACM*∧*ASM*, where *ASM* is the representation of the access state shown in the figure. This can be confirmed by using Figure 5.2 to validate each of the following statements:

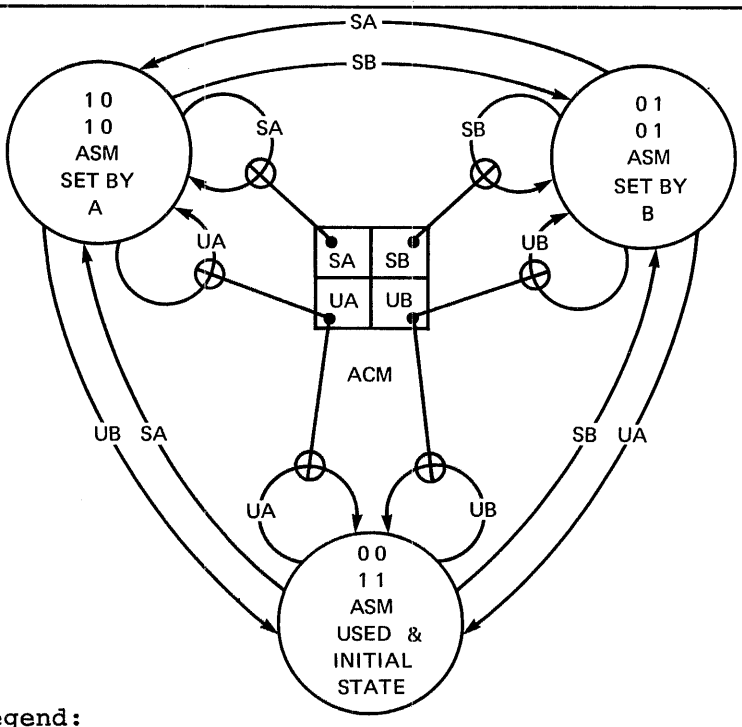
If *ACM*[1;1]=1, then two successive sets by A require an intervening access (set or use) by B.

If *ACM*[1;2]=1, then two successive sets by B require an intervening access by A.

If *ACM*[2;1]=1, then two successive uses by A require an intervening set by B.

If *ACM*[2;2]=1, then two successive uses by B require an intervening set by A.

The value of the access state representation is not directly available to a user, but the value of the access control matrix is given by the monadic function □*SVC*. For a shared variable *V* the result of the



Legend:
SA SB UA UB: Denote set or use by A or B.
ACM: Access Control Matrix
ASM: Access State Matrix

A one in an element of ACM inhibits the associated access. Allowable accesses are given by the zeroes in $ACM \wedge ASM$. Access control vectors as seen by A and B, respectively, are $,ACM$ and $,\phi ACM$.

The access state matrix represents the last access: ones occur in the last row if it is not a set, and in a column if it is, the first column if set by A and the last if set by B.

Figure 5.2: Access Control of a Shared Variable

Access Control Vector as seen by		Comments
A	B	
0 0 0 0	0 0 0 0	No constraints.
0 0 1 1	0 0 1 1	Half-duplex. Ensures that each use is preceded by a set by partner.
1 1 0 0	1 1 0 0	Half-duplex. Ensures that each set is preceded by an access by partner.
1 1 1 1	1 1 1 1	Reversing half-duplex. Maximum constraint.
0 1 1 0	1 0 0 1	Simplex. Controlled communication from B to A. (For card reader, etc.)

Figure 5.3: Some Useful Settings for the Access Control Vector

expression $\square SVC$ *'V'* executed by user A is the access control vector $,ACM$ (the four-element ravel of *ACM*). However, if user *B* executed the same expression he would obtain the result $,\phi ACM$. The reason for the reversal is that sharing is symmetric: neither owner has precedence over the other, and each sees a control vector in which the first one of each pair of control settings applies to his own accesses. This symmetry is evident in Figure 5.2; if it were redrawn to interchange the roles of A and B the control matrix would be the row-reversal of the matrix shown.

The setting of the access control matrix for a shared variable is determined in a manner which maintains the functional symmetry. An expression of the form *L* $\square SVC$ *'V'* executed by user A assigns the value of the logical left argument *L* to a four-element vector which, for the purposes of the present discussion, will be called *QA*. Similar action by user B sets *QB*. The value of the access control matrix is determined as follows:

$$ACM \leftarrow (2\ 2 \rho QA) \vee \phi 2\ 2 \rho QB$$

Since ones in *ACM* inhibit the corresponding actions, it is clear from this expression that one user can only increase the degree of control imposed by the other (although he can, by using $\square SVC$ with a left argument of zeros, restore the control to that minimum level at any time).

Access control can be imposed only after a variable is offered, either before or after the degree of coupling reaches two. The initial values of *QA* and *QB* when sharing is first offered are zero.

The access state when a variable is first offered (degree of coupling is one) is always the initial state shown in Figure 5.2. If the variable is set or used before the offer is accepted, the state changes accordingly. Completion of sharing does not change the access state.

Figure 5.3 lists a number of settings of the access control vector which are of common practical interest. Any one of them can be represented by a simplification of Figure 5.2 obtained by omitting the control matrix and deleting the lines representing those accesses which are inhibited in the particular case. For example, with maximum constraints all the inner paths would be removed from the figure.

A group of *N* access control matrices can be set at once by applying the function □*SVC* to an *N* by 4 matrix left argument and an *N*-rowed matrix right argument of names. The explicit result is an *N* by 4 matrix giving the current values of the (ravels of) control matrices. When control is being set for a single variable the left argument may be a single 1 or 0 if all inhibits or none are intended.

RETRACTION

Sharing offers can be retracted by the monadic function □*SVR* applied to a name or a matrix of names. The explicit result is the degree (or degrees) of coupling prior to the retraction. The implicit result is to reduce the degree of coupling to zero.

Retraction of sharing is automatic if the connection to the computer is interrupted or if the user signs off or loads a new workspace. Sharing of a variable is also retracted by its erasure or, if it is a local variable, upon completion of the function in which it appeared.

The nature of the shared-variable implementation is often such that the current value of a variable set by a partner will not be represented within a user's workspace until actually required to be there. This requirement obtains when the variable is to be used, when sharing is terminated, or when a *SAVE* command is issued (since the current value of the variable must be stored). Under any of these conditions it is possible for a *WS FULL* error to be reported. In all cases the prior access state remains in effect and the operation can be retried after corrective action.

INQUIRIES

There are three monadic inquiry functions which produce information concerning the shared variable environment but do not alter it; the functions □*SVO*and □*SVC* already discussed, and the function □*SVQ*. A user who applies the latter function to an empty vector obtains a vector result containing the identification of each user making any sharing offer to him. A user who applies the function □*SVQ* to a non-empty argument obtains a matrix of the names offered to him by the processor identified in the argument. This matrix includes only those names which have not been accepted by counter-offers.

The expression `(0≠□SVO M)/[1] M←□NL 2` can be used to produce a character matrix whose rows represent the names of all shared variables in the dynamic environment.

There are three ways in which a defined function can be established in an APL workspace:

1. It can be copied from a stored workspace using a system command, as described in Section 8.

2. It can be established in execution mode, using the system function ⎕*FX*, either in direct keyboard entry or in the course of execution of another defined function.

3. It can be established in function definition mode.

Regardless of which facility has been used for establishing a function, its definition can be displayed or modified in either the function definition mode, in which certain editing capabilities are built-in, or by the combined use of the system functions ⎕*CR* and ⎕*FX*.

CANONICAL REPRESENTATION AND FUNCTION ESTABLISHMENT

The character representation of a function is a character matrix satisfying certain constraints: the first row of the matrix represents the function header and must be one of the forms specified below in the section on function headers. The remaining rows of the matrix, if any, constitute the function body, and may comprise any sequence of characters. If the character representation satisfies additional constraints such as no redundant spaces and left justification of the non-blank character in each row, then it is said to be a canonical representation.

Applying ⎕*CR* to the character array representing the name of an already established function will produce its canonical representation. For example, if *OVERTIME* is an available function:

```
      DEF←⎕CR 'OVERTIME'
      DEF
PAY←R OVERTIME H;TIME
TIME←0⌈H-40
PAY←R×1.5×TIME

      ⍴DEF
3 21
```

The function ⎕*CR* applied to any argument which does not represent the name of an unlocked defined function yields a matrix of shape 0 0. Error reports for ⎕*CR* are *RANK* error if the argument is not a vector or a scalar, or *DOMAIN* error if the argument is not a character array.

The use of ⎕*CR* does not change the status of the function *OVERTIME*, which remains established and can be used for calculations. Thus:

```
      7 5 8 OVERTIME 35 40 45
0 0 60
```

If *OVERTIME* should be expunged:

```
      ⎕EX 'OVERTIME'
1
```

it is no longer available for use:

```
      7 5 8 OVERTIME 35 40 45
SYNTAX ERROR
      7 5 8 OVERTIME 35 40 45
            ^
```

The function can be re-established by □*FX*:

```
      □FX DEF
OVERTIME
```

The function □*FX* produces as its explicit result the vector of characters which represents the name of the function being fixed, while replacing any existing definition of a function with the same name. The function *OVERTIME* can now be used again:

```
      7 5 8 OVERTIME 35 40 45
0 0 60
```

As noted in Section 4, an expression of the form □*FX M* will establish a function if the following conditions are met:

1. *M* is a valid representation of a function. Any matrix which differs from a canonical matrix only in the addition of non-significant spaces (other than rows consisting of spaces only) is a valid representation.

2. The name of the function to be established does not conflict with an existing use of the name for an executing or halted function or for a label, group, or variable.

If the expression fails to establish a function then no change occurs in the workspace and the expression returns a scalar index of the row in the matrix argument where the fault was found. If the argument of □*FX* is not a matrix a *RANK* error will be reported, and if it is not a character array a *DOMAIN* error will result.

THE FUNCTION HEADER

The valence of a function is defined as the number of explicit arguments which it takes. A defined function may have a valence of zero, one, or two, and may yield an explicit result or not. These cases are represented by six forms of header as follows:

Type	Valence	Result	No Result
Dyadic	2	*R←A F B*	*A F B*
Monadic	1	*R←F B*	*F B*
Niladic	0	*R←F*	*F*

The names used for the arguments of a function become local to the function, and additional local names may be designated by listing them after the function name and argument, separated from them and from each other by semicolons; the name of the function is global. The significance of these distinctions is explained below.

Except that the function name itself may be repeated in the list of local names, a name may not be repeated in the header. It is not obligatory either for the arguments of a defined function to be used within the body, or for the result variable to be specified in the course of function execution.

LOCAL AND GLOBAL NAMES

In the execution of a defined function it is often necessary to work with intermediate results or temporary functions which have no significance either before or after the function is used. The use of local names for these purposes, so designated by their appearance in the function header, avoids cluttering the workspace with a multitude of objects introduced for such transient purposes, and allows greater freedom in the choice of names. Names used in the function body, and not so designated, are said to be global to that function.

A local name may be the same as that for a global object, and any number of names local to different functions may be the same. During the execution of a defined function, a local name will temporarily exclude from use a global object of the same name. If the execution of a function is interrupted (leaving it either suspended, or pendent (See Section 7), the local objects retain their dominant position, during the execution of subsequent APL operations, until such time as the halted function is completed. However, system commands and the del form of function definition (see below) continue to reference global objects under these circumstances.

The localization of names is dynamic, in the sense that it has no effect except when the defined function is being executed. Furthermore, when a defined function uses another defined function during its execution, a name localized in the first (or outer) function continues to exclude global objects of the same name from the purview of the second (or inner) function. This means that a name localized in an outer function has the significance assigned to it in that function when used without further localization in an inner function. The same name localized in a sequence of nested functions has the significance assigned to it at the innermost level of execution.

The shadowing of a name by localization is complete, in the sense that once a name has been localized its global and outer values are nullified, even if no significance is assigned to it during execution of the function in which it is localized.

BRANCHING AND STATEMENT NUMBERS

Statements in a function are normally executed successively, from top to bottom, and execution terminates at the end of the last statement in the sequence. This normal order can be modified by branches. Branches are used in the construction of iterative procedures, in choosing one out of a number of possible continuations, or in other situations where decisions are made during the course of function execution.

To facilitate branching, the successive statements in a function definition have reference numbers associated with them, starting with the number one for the first statement in the function body and continuing with successive integers, as required. Thus, the expression →4 denotes a branch to the fourth statement in the function body, and when executed causes statement 4 to be executed next, regardless of where the branch statement itself occurs. (In particular →4 may be statement 4, in which case the system will simply execute this "tight loop" indefinitely, until interrupted by an action from the keyboard. This is a trap to be avoided.)

A branch statement always starts with the branch arrow (or right arrow) on the left, and this can be followed by any expression. For the statement to be effective, however, the expression must evaluate to an integer, to a vector whose first element is an integer, or to an empty vector; any other value results in a *DOMAIN* or *RANK* error. If the

expression evaluates to a valid result, then the following rules apply:

1. If the result is an empty vector the branch is vacuous and execution continues with the next statement in the function if there is one, or else the function terminates.

2. If the result is the number of a statement in the function then that statement is the next to be executed.

3. If the result is a number out of the range of statement numbers in the function, then the function terminates. The number 0 and all negative integers are outside of the range of statement numbers for any function.

Since zero is often a convenient result to compute, and it is not the number of a statement in the body of any function, it is often used as a standard value for a branch intended to end the execution of a function. It should be noted that in the function definition mode described below, zero is used to refer to the header. This has no bearing on its use as a target for a branch.

An example of the use of a branch statement is shown in the following function, which computes the greatest common divisor of two scalars:

```
Z←M GCD N
Z←M
M←M|N
N←Z
→(0≠M)/1
```

The compression function in the form *U/V* gives *V* if *U* is equal to one, and an empty vector if *U* is equal to zero. Thus, the fourth statement in *GCD* is a branch statement which causes a branch to the first statement when the condition *0≠M* is true, and a branch with an empty vector argument, that is, normal sequence, when the condition is false. In this case, there is no next statement and so execution of the function ends.

LABELS

If a statement occurring in the body of a function definition is prefaced by a name and a colon, then the name is assigned a value equal to the statement number. A name used in this way is called a label. Labels are used to advantage when it is expected that a function definition may be changed for one reason or another, since a label automatically assumes the new value of the statement number of its associated statement as other statements are inserted or deleted.

The name of a label is local to the function in which it appears, and must be distinct from other label names and from the local names in the header.

A label name may not appear immediately to the left of a specification arrow. In effect it acts like a (local) constant.

COMMENTS

The lamp symbol ⍝ (the cap-null) signifies that what follows it is a comment, for illumination only and not to be executed; it may occur only as the first character in a statement.

FUNCTION EDITING - THE ∇ FORM

The functions □*CR* and □*FX* together form a basis for establishing and revising functions. Convenient definition editing with them, however, requires the use of prepared editing functions which must be defined, stored in a library, and explicitly activated when needed. The del form described here provides an alternative facility for function entry and revision which is always present for use.

When the user enters the del character (∇) followed by the name of a defined function, the system responds by displaying [*N*+1] , where *N* is the number of statements in the function. It is now possible to:

add, insert, or replace statements
replace the header
modify the header or a statement
delete a statement
display all or part of the definition

A new function is started by entering the desired header on the same line as the opening ∇. Once the function definition mode has thus been entered, the treatment of a new function is identical to that for a function already defined.

ADDING A STATEMENT

If the response to the display of statement number [*N*+1] is a statement, it is accepted as a line added at the end of the definition. The system response is the display [*N*+2] . Additional statements may continue to be added to the definition in this manner. If an empty statement is entered, the system will re-display the line number in brackets.

INSERTING OR REPLACING A STATEMENT

If the response to the statement number displayed by the system is [*N*] , where *N* is any positive number with or without a fractional part, the system will display [*N*] . A statement entered will replace an existing statement *N*.

The system continues by displaying the next appropriate number. For example, if the statement number entered was [3] , the next number displayed will be [4] ; if [3.02] , then [3.03] ; if [3.29] then [3.3] , and so forth.

A statement may be submitted with line number [*N*]; it will be inserted or will replace an existing statement in the manner described. The response of the system in this case is to display the next statement number.

REPLACING THE HEADER

If the user enters [0] , the system responds by displaying [0] . The user may now enter any legal header, which will replace the existing header. Following this, the system displays [1] . The entire operation may be carried out by entering [0] and, on the same line, the header.

DELETING A STATEMENT

One or more statements may be deleted by entering in brackets a delta followed by one or more statement numbers, for example, [Δ2 7 3]. The response of the system is to display the statement number that follows the last number in the list. In the example, the response will be [4].

EDITING A STATEMENT OR HEADER

Statement *N* can be modified by the following mechanism:

1. Enter [*N*□*M*] where *M* is an integer. The header is referred to by using 0 for *N*.
2. Statement *N* or the header is automatically displayed and the cursor stops under position *M*, which is counted from the left margin.
3. A letter or decimal digit or the symbol / may be typed under any of the positions in the display. Any other characters typed in this mode are ignored. The ordinary rules for editing before entry (such as backspace and linefeed) apply.
4. The line is re-displayed. Each character understruck by a / is deleted, each character understruck by a digit *K* is preceded by *K* added spaces, and each character understruck by a letter is preceded by 5×*R* spaces, where *R* is the position of the letter in the alphabet. For example, the letter *C* will produce 15 spaces. Finally, the cursor moves to the first injected space and awaits the typing of modifications in the usual manner. The final effect is as if the entry had been made entirely from the keyboard; in particular, a completely blank sequence leaves the line unchanged.

If a statement number itself is changed from *N* to *P* during the editing procedure the statement affected is determined by the new statement number *P*; hence statement *N* remains unchanged. This permits statements to be moved, with or without modification.

If an attempt to display a line would exceed the printing width (determined by □*PW*) this form of editing is not applicable. Such a condition may arise if the line is too long to start with, or if too many spaces have been requested for insertion.

ADDING TO A STATEMENT OR HEADER

One or more characters can be added to the end of statement *N*, or statement *N* can be corrected, by entering [*N*□0]. In response, the system displays statement *N*; the cursor is positioned at the end of the statement, and the keyboard unlocks. The statement may be extended, or modified by using the normal revision procedures for entry. In response, the system displays the next statement number and awaits entry.

The header may be modified in this way by entering [0□0].

FUNCTION DISPLAY

The canonical representation of a function includes the header and body displayed as a character matrix. The ∇ form permits display of a canonical representation modified as follows:

1. Labelled lines and comments are offset one space to the left.
2. Statement numbers in brackets are appended to the left of the statements.
3. A del character (∇) is prefixed to the header, separated by one space.
4. A final line is added, consisting of spaces and a del character, aligned with the del character which prefixes the header.

Shown are the canonical representation and function display of a function for computing the determinant of a matrix:

```
      ⎕CR'DET'
Z←DET A;B;P;I
I←⎕IO
Z←1
L:P←(|A[;I])⍳⌈/|A[;I]
→(P=I)/LL
A[I,P;]←A[P,I;]
Z←-Z
LL:Z←Z×B←A[I;I]
→(0 1 ∨.=Z,1↑⍴A)/0
A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
→L
⍝EVALUATES A DETERMINANT
```

```
      ∇DET[⎕]∇
∇ Z←DET A;B;P;I
[1]   I←⎕IO
[2]   Z←1
[3]  L:P←(|A[;I])⍳⌈/|A[;I]
[4]   →(P=I)/LL
[5]   A[I,P;]←A[P,I;]
[6]   Z←-Z
[7]  LL:Z←Z×B←A[I;I]
[8]   →(0 1 ∨.=Z,1↑⍴A)/0
[9]   A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
[10]  →L
[11] ⍝EVALUATES A DETERMINANT
∇
```

The display of a function can be obtained by typing [⎕]. After the display, the system displays [*N*+1] where *N* is the number of statements in the function.

Statement *N* can be displayed by typing [*N*⎕]. After display the system types [*N*]. Entering [⎕*K*] causes display of all statements from *K* onwards. After the display the system displays [*N*+1] where *N* is the number of statements in the function.

LEAVING THE ∇ FORM

The del form may be left by typing a ∇ on a line by itself, or as the last character on any entry except one which enters a comment statement.

In particular, it can follow a request for display or a function statement, and either can be incorporated in the same entry that both opens and closes the definition mode, as in ∇*DET*[⎕]∇ or ∇*DET*[10]→*L*,*L*∇. On leaving the del form, the statements are reordered according to their statement numbers and the statement numbers are replaced by the integers 1, 2, 3, and so on.

A function definition can be locked by either opening or closing the definition mode with a del-tilde, ⍫. The use of this is explained in Section 7.

SECTION 7: FUNCTION EXECUTION

A defined function may be used like a primitive function, except that it may not be the argument of a primitive operator. In particular, a defined function may be used within its own definition or that of another defined function. When a function is called, or put into use, its execution begins with the first statement, and continues with successive statements except as this sequence is altered by branch instructions.

Consider the function *OVERTIME*:

```
PAY←R OVERTIME H;TIME
TIME←0⌈40-H
PAY← R×1.5×TIME
```

If this function is invoked by a statement such as *X OVERTIME Y* the effect is to assign to the local name *R* the value of *X* and to *H* the value of *Y*, and then execute the body of the function *OVERTIME*. Except for having a value assigned initially, the argument variable is treated as any other local variable and, in particular, may be respecified within the function.

A function like *OVERTIME*, which produces an explicit result, may properly be used in compound expressions. In the *OVERTIME* function the last value received by *PAY* in the course of execution is the explicit result of the function. For example:

```
      YTDAT
100   200   150
      YTDAT←YTDAT+OT←5 7 6 OVERTIME 35 40 45
      OT
0     0     45
      YTDAT
100    200    195
```

PAY, itself, is a local variable and therefore has no significance after the function is executed:

```
      PAY
VALUE ERROR
      PAY
      ^
```

HALTED EXECUTION

The execution of a function *F* may be stopped before completion in a variety of ways: by an error report, by an attention signal, or by the stop control treated below. When this happens the function is said to be suspended, and its progress can be resumed by entering a branch statement from the keyboard. Whatever the reason for suspension, the name of the function is displayed, with a statement number beside it. In the case of an error stop or an interrupt the statement itself is also displayed, with an appropriate message and an indication of the point of interruption. Unless a specification appears in the statement to the right of this point, the state of the computation has been restored to the condition obtaining before the statement started to execute.

In general, therefore, the displayed number is that of the statement that should be executed next if the function is to continue normally. Resumption of execution at that point can be accomplished by entering a branch to that number specifically, a branch to an empty vector, or a branch to □*LC*. Entering →0, or a branch to another number

outside the range of statement numbers, causes an immediate exit from the function and it is no longer suspended.

In the suspended state all normal activities are possible, but names used refer to their local significance, if any. The system is in a condition to execute statements or system commands, resume execution of the function at an arbitrary point, or enter definition mode to work on any function which is not pendent (see below). In particular, the suspended function can be edited at this point (if it is not itself pendent from a previous use).

STATE INDICATOR

Entering the system command)*SI* causes a display of the state indicator; a typical display has the following form:

```
      )SI
H[7]  *
F[2]
G[3]
```

This display indicates that execution was halted before completing (perhaps before starting) execution of statement 7 of function *H*, that the current use of function *H* was invoked in statement 2 of function *F*, and that the use of function *F* was in turn invoked in statement 3 of *G*. The * appearing to the right of *H*[7] indicates that the function *H* is suspended. The functions *G* and *F* are said to be pendent, because their execution cannot be restarted directly, but only as a consequence of function *H* resuming its course of execution. The term halted is used to describe a function which is either pendent or suspended.

Further functions can be invoked when in the suspended state. Thus, if *G* were now invoked and a further suspension occurred in statement 5 of *Q*, itself invoked in statement 8 of *G*, a subsequent display of the state indicator would appear as follows:

```
      )SI
Q[5]   *
G[8]
H[7]   *
G[2]
F[3]
```

Since the line counter, ⎕*LC*, holds the current statement numbers of functions that are in process of execution, its value at this point would be the vector 5 8 7 2 3. The sequence from the last to the preceding suspension can be cleared by entering a right arrow (→). This behavior is illustrated by continuing the foregoing example as follows!

```
      →
      )SI
H[7]   *
G[2]
F[3]
      ⎕LC
7 2 3
```

Repeated use of → will clear the state indicator completely, and restore ⎕*LC* to an empty vector. The cleared state indicator displays as a blank line.

STATE INDICATOR DAMAGE

If the name of a function occurs in the state indicator list, then erasure of the function or replacement of the function by copying a function with the same name (even another instance of the same function) will make it impossible for the original course of execution to be resumed. In such an event an *SI DAMAGE* report is given. In addition, some APL systems will give an *SI DAMAGE* report if a suspended function is edited to change the order of its labels or to modify its header.

If an *SI DAMAGE* report is given for a suspended function, it will not be possible to resume its execution by entering a branch statement, but the function can be invoked anew, with or without prior clearance of the state indicator.

In case of *SI DAMAGE*, display of the state indicator will show the damage, possibly by a blank where the function name should have appeared or by some explicit indicator or both.

TRACE CONTROL

A trace is an automatic display of information generated by the execution of a function as it progresses. In a complete trace of a function, the number of each statement executed is displayed in brackets, preceded by the function name and followed by the final value produced by the statement. The trace of a branch statement shows a branch arrow followed by the number of the next statement to be executed. The trace is useful in analyzing the behavior of a defined function, particularly during its design.

The tracing of a function *PROFIT* is controlled by the trace control for *PROFIT*, denoted by *T∆PROFIT*. If one sets *T∆PROFIT*←2 3 5 then statements 2, 3, and 5 will be traced in any subsequent execution of *PROFIT*. *T∆PROFIT*←⍳0 discontinues tracing of *PROFIT*. A complete trace of *PROFIT* is obtained by *T∆PROFIT*←⍳*N*, where *N* is the number of statements in *PROFIT*. In general, the trace control for any function is designated by prefixing *T∆* to the function name.

STOP CONTROL

A function can be caused to execute up to a certain statement and then stop in the suspended state. This is frequently useful in analyzing a function, for example by experimenting with local variables or intermediate results. The stops are set by the stop control in the same manner as the trace. For a function *PROFIT*, stops before lines 4 and 12 are executed would be set by entering *S∆PROFIT*←4 12.

At each stop, the function name and line number are displayed as described above for suspended functions. To go to the next stopping point after the first, execution must be explicitly restarted by entering an appropriate branch statement.

Trace control and stop control can be used in conjunction. Moreover, either of the controls may be set within functions. In particular, they may be set by expressions which initiate tracing or stops as a result of certain conditions that may develop in the course of function execution, such as a particular variable taking on a particular value. They may only be used as the left argument of specification. They may not be used by themselves or as the argument to a function.

LOCKED FUNCTIONS

If the symbol ⍫ (called del-tilde) is used instead of ∇ to open or close a function definition, the function becomes locked. A locked function cannot be revised or displayed in any way. Any associated stop control or trace control is nullified after the function is locked.

A locked function is treated essentially as a primitive, and its inner workings are concealed as much as possible. Execution of a locked function is terminated by any error ocurring within it, or by a strong interrupt. If execution stops the function is never suspended but is immediately abandoned. The message displayed for a stop is *DOMAIN* error if an error of any kind occured, *WS FULL* and the like if the stop resulted from a system limitation, or *INTERRUPT*.

Moreover, a locked function is never pendent, and if an error occurs in any function invoked either directly or indirectly by a locked function, the execution of the entire sequence of nested functions is abandoned. If the outermost locked function was invoked by an unlocked function, that function will be suspended; if it was invoked by a keyboard entry the error message will be displayed with a copy of that statement.

Similarly, when a weak interrupt is encountered in a locked function, or in any function that was ultimately invoked by a locked function, execution continues normally up to the first interruptable point: either the next statement in an unlocked function that invoked the outermost locked function, or the completion of the keyboard entry that used this locked function. In the latter case, the weak interrupt has no net effect.

Locked functions may be used to keep a function definition proprietary, or as part of a security scheme for protecting other proprietary information. They are also used to force the kind of behavior just described, which sometimes simplifies the use of applications.

RECURSIVE FUNCTIONS

A defined function whose name has not been made local and is used in the body of the function definition is said to be defined recursively. For example, one definition of the greatest common divisor function states that the greatest common divisor of zero and any number *N* is *N*; for any other pair of numbers it is the greatest common divisor of the residue of the second number by the first, and the first number. The words "greatest common divisor" are used in the definition. This suggests that a greatest common divisor function *GCDR* can be written whose canonical representation is:

```
      ⎕CR'GCDR'
R←A GCDR B
R←B
→(0=A)/0
R←(A|B) GCDR A

      18 GCDR 45
9
```

This can be compared to the equivalent function defined iteratively in Section 6.

Executing an erroneously defined recursive function will often result in a *WS FULL* report. The non-trivial execution of a properly defined recursive function may also have this effect because of the very deep nesting of function calls that is often required.

TERMINAL INPUT AND OUTPUT

In many significant applications, such as text processing, for example, it is necessary that the user supply information as the execution of the application programs progresses. It is also often convenient, even in the use of an isolated function, to supply information in response to a request, rather then as arguments to the function as part of the original entry. This is illustrated by considering the use of the function *CI*, which determines the growth of a unit amount invested at periodic interest rate *R* for number of periods *T*:

```
      ⎕CR'CI'
A←R CI T
A←(1+R)*T
```

For example, the value of 1000 dollars at 5 per cent for 7 years, compounded quarterly, might be found by:

```
      1000 × (.05÷4) CI 7×4
1415.992304
```

The casual user of such a function might, however, find it difficult to remember which argument of *CI* is which, how to adjust the rate and period stated in years for the frequency of compounding, and whether the interest rate is to be entered as the actual rate (for example, .05) or as a percentage (for example, 5). An exchange of the following form might be more suitable:

```
      INVEST
ENTER CAPITAL AMOUNT IN DOLLARS
⎕:
      1000
ENTER NUMBER OF TIMES COMPOUNDED IN ONE YEAR
⎕:
      4
ENTER ANNUAL INTEREST RATE IN PERCENT
⎕:
      5
ENTER PERIOD IN YEARS
⎕:
      7
VALUE IS 1415.992304
```

It is necessary that each of the entries (1000, 4, 5, and 7) occurring in such an exchange be accepted not as an ordinary entry (which would only evoke the response 1000, etc.), but as data to be used within the function *INVEST*. Facilities for this are provided in two ways, termed evaluated input, and character input. A definition of the function *INVEST*, which uses evaluated input, is as follows:

```
      ⎕CR'INVEST'
INVEST;C;R;T;F
'ENTER CAPITAL AMOUNT IN DOLLARS'
C←⎕
'ENTER NUMBER OF TIMES COMPOUNDED IN ONE YEAR'
F←⎕
'ENTER ANNUAL INTEREST RATE IN PER CENT'
R←⎕÷F×100
'ENTER PERIOD IN YEARS'
T←F×⎕
'VALUE IS ',▼C×R CI T
```

The function *NEWSTOCK* in the Introduction (Section 1) is an example of the use of both character input and bare output (discussed below).

EVALUATED INPUT

The quad symbol ⎕ appearing anywhere other than immediately to the left of a specification arrow denotes request for keyboard input as follows: the two symbols ⎕: are displayed, and the keyboard is unlocked on the next line, indented from the left margin. Any valid expression entered at this point is evaluated, and the result substituted for the quad. Suppose *F* is a function whose definition includes a quad symbol:

```
      ⎕CR'F'
Z←F
Z←4×⎕
      F
⎕:
      3+2
20
```

An invalid entry in response to a request for quad input causes an appropriate error report, after which input is again awaited. For example, entering an expression which has no result produces a value error. Function definition mode (the editing or display of functions, or creation of new functions) is not permitted during ⎕ entry. In general, a system command entered during ⎕ input is executed, but the system's response to the command is not treated as a response to ⎕. After execution of a command, valid input is again awaited (unless the command was one which replaced the contents of the active workspace). An empty input (one containing nothing other than zero or more spaces) is rejected and the system again awaits input.

INTERRUPTING EVALUATED INPUT

Execution of a defined function containing a request for ⎕ input can be interrupted at the statement containing the ⎕ by entering a statement which has no value (for example ⍎'' or a branch statement, or any defined function which does not return a result). This suspends execution of the statement containing the ⎕, with the report *VALUE ERROR*. A command that requires saving the workspace (either *SAVE* or *CONTINUE*, described in Section 8), issued during a request for ⎕ input, also interrupts the statement containing the ⎕ before carrying out the command.

The response → entered in response to a ⎕ abandons execution of the function and any pendent functions leading up to it.

CHARACTER INPUT

The quote-quad symbol ⍞ (that is, a quad overstruck with a quote) is a request for character input: entry is permitted at the left margin and data entered are accepted as characters. For example:

```
      X←⍞
CAN'T                    (Quote-quad input, not indented)
      X
CAN'T
```

INTERRUPTING CHARACTER INPUT

Request for ⍞ input can be interrupted, in the case of certain typewriter terminals, by entering the three letters *OUT*, in that order, but with a backspace between each pair so that they all overstrike. This interrupts execution, but does not cause an exit from a defined function.

NORMAL OUTPUT

The quad symbol appearing immediately to the left of a specification arrow indicates that the value of the expression to the right of the arrow is to be displayed in the standard format (subject to the printing precision ⎕*PP* and the printing width ⎕*PW*). Hence, ⎕←*X* is equivalent to the statement *X*. The longer form ⎕←*X* is useful when employing multiple specification. For example, ⎕←*Q*←*X**2 assigns to *Q* the value *X**2 and then prints the value of *X**2.

The maximum length of a line of normal display (measured in characters) is called the printing width, and is given by the value of the system variable ⎕*PW*. The maximum and minimum effective values of the printing width depend upon the system being used, and the appropriate user's guide should be consulted. A display whose lines exceed the printing width is ended at or before the maximum length, and continued on subsequent lines.

BARE OUTPUT

Normal output includes a concluding new line signal so that the succeeding display (either input or output) will begin at a standard position on the following line. Bare output, denoted by expressions of the form ⍞←*X* does not include this signal if it is followed either by another bare output or by character input (of the form *X*←⍞).

Character input following a bare output is treated as though the user had spaced over to the position occupied at the conclusion of the bare output, so that the characters received in response will normally be prefixed by a number of space characters. This allows for the possibility that, after the keyboard is unlocked, the user backspaces into the area occupied by the preceding output. The following function prompts the user with whatever message is supplied as its argument, and evaluates the response:

```
      ⎕CR'PROMPT'
Z←PROMPT MSG
⍞←MSG
Z←⍞
```

Using such a function, the expression:

```
PROMPT 'ENTER CAPITAL: '
```

would have the following effect:

```
displayed by system
        /
ENTER CAPITAL: 1000
                \
          entered by user
```

The value of *Z* is as many blank characters as there are characters in *MSG*, followed by the characters entered by the user. In this case *Z* will have fifteen blanks followed by the character '1000'.

The new line signals that would be supplied by the system in order to break lines that exceed the printing width are not supplied with bare output. However, since an expression of the form ⍞←*X* entered directly from the keyboard (rather than being executed as part of a defined function) must necessarily be followed by another keyboard entry, the output it causes is concluded with a new line signal. The effect is in this case indistinguishable from normal output, except for the possibility of exceeding the printing width limitation.

An APL system recognizes two broad classes of instructions, statements and system commands. System commands control the initiation and termination of a work session, saving and reactivating copies of a workspace, and transferring data from one workspace to another.

System commands can be invoked only by individual entries from the keyboard and cannot be executed dynamically as part of a defined function. They are prefixed by a right parenthesis.

The system commands are summarized in Figure 8.1. They will be discussed under three main headings:

1. The active workspace.
   a. Action.
   b. Inquiry.

2. Workspace storage and retrieval.
   a. Action.
   b. Inquiry.

3. Access to the system.

A command that is not recognizable, or is improperly formed, is rejected with the report *INCORRECT COMMAND*. Certain commands may also result in more specific trouble reports; these are discussed in the appropriate context and are summarized in Figure 8.2.

Once its execution has started, a system command cannot be interrupted, although display of the system's response to the command can be suppressed by an interrupt signal.

In the text that follows, each system command is shown in a sample form. In use, the appropriate names or numbers should, of course, be substituted for those shown.

A	A letter of the alphabet.
N	A number.
USERNO	A user account number.
LIBNO	A library number (that is, either a user account number or a public library number).
WSNAME	A workspace name.
GRPNAME	A group name.
OBJNAME	A name of an object within a workspace (that is, a function, a variable, or a group).
PASSWD	A password, which must match the previously set value.
NEWPASS	A new password, which need not match the previously set value.
[ ]	Items enclosed in brackets may in some circumstances be omitted.

FORM	PURPOSE	NORMAL RESPONSE	TROUBLE REPORTS
TERMINAL CONTROL COMMANDS			
)number [pass]	Identify user and start use of APL	[public-address] header system [*SAVED* time date]	1 5 12 13 14
)*CONTINUE* [newpass]	Replace ws *CONTINUE* by copy of active ws and end use of APL	[time date *CONTINUE*] header account	4 18
)*CONTINUE HOLD* [newpass]	Replace ws *CONTINUE* by copy of active ws, end use of APL, but hold connection	[time date *CONTINUE*] header account	4 18
)*OFF* [newpass]	End use of APL	header account	4
)*OFF HOLD* [newpass]	End use of APL, but hold connection	header account	4
ACTIVE WORKSPACE CONTROL COMMANDS			
)*CLEAR*	Activate a clear ws	*CLEAR WS*	4
)*COPY* wsid [pass]	Copy all global objects from named ws into active ws	*SAVED* time date	2 3 4 16 17 18 19 20
)*COPY* wsid [pass] names	Copy global objects named from designated ws into active ws	*SAVED* time date	2 3 4 7 16 17 18 19 20
)*ERASE* [names]	Erase global objects named from active ws		4 7 16
)*GROUP* names	Gather objects into (or disperse) a group (first name designates group)		4 8 17 18
)*LOAD* wsid [pass]	Activate copy of named ws	*SAVED* time date	3 4 18 19
)*PCOPY* wsid [pass]	Copy all objects from designated ws not named in active ws	*SAVED* time date	2 3 4 6 16 17 18 19 20
)*PCOPY* wsid [pass] names	Copy objects designated that are not named in active ws	*SAVED* time date	2 3 4 6 7 16 17 18 19 20
)*SYMBOLS* number	Change maximum number of symbols in ws to number	*WAS* number	4
)*WSID* wsid [newpass]	Change identification of active ws	*WAS* wsid	4
LIBRARY CONTROL COMMANDS			
)*DROP* wsid	Drop ws from library	time date	3 4 20
)*SAVE*	Place copy of active ws in library	time date wsid	3 4 9 10 11 18
)*SAVE* wsid [newpass]	Replace named ws by copy of active ws	time date	3 4 9 10 11 18
INQUIRY COMMANDS			
)*FNS* [alphabetic]	List defined functions (whose initials follow given character in alphabet)	[names]	4
)*GRP* name	List members of named group	[names]	4 17
)*GRPS* [alphabetic]	List groups (whose initials follow given character of alphabet)	[names]	4
)*LIB* [number] [alphabetic]	List workspaces in designated library (whose initials follow given character in alphabet)	[names]	3 4
)*SI*	List halted functions	state-indicator	4
)*SIV*	List halted functions and associated local names	state-indicator and names	4
)*SYMBOLS*	Give maximum number of names in ws	*IS* number	4
)*VARS* [alphabetic]	List global variables (whose initials follow given character in alphabet)	[names]	4
)*WSID*	Give wsid of active ws	[number] name	4

Notes: 1. Items in brackets are optional
2. Abbreviations and meanings:

ws	workspace
pass	a colon possibly followed by a password
newpass	a pass that does not have to match a previous password
header	a port number, time of day, date, and user-code
account	gives the connect time and compute time since last start and since beginning of the accounting period
wsid	a ws name possibly preceded by a library number
public-address	a message to users of the system

3. The commands)*ERASE*,)*FNS*, and)*VARS* have variants that are system functions (see □*EX* and □*NC*).

Figure 8.1: System Commands

No.	*TROUBLE REPORT*	Meaning	Remedy
1	*ALREADY SIGNED ON*	APL already in use at terminal	1) To display account number, use □*AI* 2) To change account number, execute *)OFF HOLD* or *)CONTINUE HOLD* and sign on
2	*DEFN ERROR*	Attempted copy or protected copy of function definition as response to □ input request (on some systems)	
3	*IMPROPER LIBRARY REFERENCE*	1) Number is not a library number, or 2) Attempted save into alien library, or 3) Attempted reference to alien *CONTINUE* ws	
4	*INCORRECT COMMAND*		
5	*INCORRECT SIGN-ON*		
6	*NOT COPIED*: names	Global homonyms in active ws are protected	
7	*NOT FOUND*: purported names	Ws does not contain global objects with purported names	
8	*NOT GROUPED, NAME IN USE*	First name is name of a global function or variable	1) Use different name for group, or 2) Erase global object if not needed
9	*NOT SAVED, THIS WS IS CLEAR WS*	A clear ws has no name and cannot be stored	
10	*NOT SAVED, THIS WS IS* wsid	Attempted replacement of a stored ws whose identification does not match that of the active ws	Remove active ws, then store
11	*NOT SAVED, WS QUOTA USED UP*	Allotted number of stored wss previously reached	1) Drop an unneeded ws, or 2) Ask APL operator to increase quota
12	*NUMBER IN USE*		Consult APL operator
13	*NUMBER LOCKED OUT*	Authorization for use of number has been withdrawn	Consult APL operator
14	*NUMBER NOT IN SYSTEM*	1) Number entered is not an account number, or 2) Password missing, or 3) Wrong password used	Consult APL operator
15	*RESEND*	Transmission failure or more than the implementation allowed number of characters entered in one line	If chronic, redial or have terminal or phone repaired
16	*SI DAMAGE*	State indicator damaged while performing an *)ERASE* or *)COPY* command	
17	*SYMBOL TABLE FULL*	Too many names used	Erase objects not needed, save ws, clear active ws, and perhaps change limit, using *)SYMBOLS*, copy the saved ws and rename the active ws
18	*WS FULL*	Workspace full, possibly because of 1) Temporary values produced during evaluation of an expression, or 2) Value assigned to shared variable by partner.	1) Erase objects not needed, or 2) Clear state indicator, or 3) Revise method of calculation
19	*WS LOCKED*	1) Password missing, or 2) Wrong password used	
20	*WS NOT FOUND*	No stored ws with given identification	

Figure 8.2: Trouble Reports

< >	Items enclosed in angles may in some circumstances be omitted, and, where omitted, the system supplies values from the existing user identification, workspace identification, or previously established password.

COMMANDS THAT MODIFY THE WORKSPACE ENVIRONMENT

The following system commands affect the active workspace, the environment in which computation takes place and in which names have meaning. In particular, the active workspace contains the settings of the state indicator (discussed in Section 7) and other elements of the computing environment, mediated by several of the system variables (discussed in Section 4).

)*CLEAR*

This command is used to make a fresh start, discarding the contents of the active workspace, and resetting the environment to standard initial values (see Figure 8.3). At sign-on, the user receives a clear workspace characterized by these same initial values, unless the workspace *CONTINUE* was automatically loaded.

Symbol table size	* 256
Index origin, ⎕*IO*	1
Latent expression, ⎕*LX*	Empty
Line counter, ⎕*LC*	Empty
State indicator	Cleared
Workspace name	None (*CLEAR WS*)
Workspace password	None
Printing precision, ⎕*PP*	* 10
Printing width, ⎕*PW*	* 120
Comparison tolerance, ⎕*CT*	* $1E^{-}13$
Random link, ⎕*RL*	* 16807
Work area available, ⎕*WA*	* Depends on the local installation, and, in some systems, upon options selected by the user.

* Items marked with an asterisk have values which may vary from system to system. The values shown are widely used.

Figure 8.3. Environment Within a Clear Workspace

)SYMBOLS N

Sets the size of the symbol table, that is, the maximum number of names that may occur in the workspace. (Note that the occurrence of a name includes not only the names of functions, variables, or groups themselves, but also any names occurring within their definitions). New values of the maximum may be set only in a clear workspace. An attempt to change the maximum once the workspace is no longer clear, or to set it outside the range permitted by the system, is rejected with the report *INCORRECT COMMAND*. Valid use of the command results in the report *WAS ...*, showing the former limit.

)ERASE [OBJNAME1 OBJNAME2 OBJNAME3 ...]

The global objects named are expunged from the workspace; shared variable offers with respect to any of them are retracted. If a name is the name of a group (see below), the group and all of its members present in the workspace are erased. (If a member of a group is itself a group, its definition is erased but not its members.)

If a halted function is erased, the report *SI DAMAGE* is displayed, and the name of that function is replaced in the state indicator by blanks. It is not possible to resume the execution of an erased function, and the user should enter → one or more times to clear the state indicator of indications of damage.

If an object named in the command cannot be found, the report *NOT FOUND*: is emitted, followed by a list of the objects not found.

)COPY <LIBNO> WSNAME [:PASSWD] < OBJNAME1 [OBJNAME2 ...] >

The indicated global objects (but not system variables) are copied from the indicated workspace (the source workspace) into the active workspace. The system reports the date and time at which the source workspace was saved.

If the list of objects to be copied is omitted, all global objects other than system variables are copied from the source workspace.

If the indicated source workspace is for some reason unavailable, copying cannot take place. The possible errors in gaining access are the same as those discussed under the *LOAD* command, namely *IMPROPER LIBRARY REFERENCE*, *WS NOT FOUND*, *WS LOCKED*.

When an object to be copied is a group, the group membership list is copied as well as all those of its members which exist as global objects in the source workspace.

When an object to be copied has the same name as a global object in the active workspace, the copied object replaces it. If there was a shared variable offer with respect to the variable thus replaced, the offer is retracted.

If names explicitly mentioned in the copy command are not the names of global objects in the source workspace, the system reports *NOT FOUND*: followed by a list of the objects not found.

Copy During Evaluated Input. A copy command can be entered while a request for evaluated input (⎕:) is pending. The objects copied will then exist in the workspace but do not become the value of ⎕. The request for evaluated input is then repeated. However, definition mode may not be entered during evaluated input, and hence the definition of a function may not be copied during evaluated input. An attempt to do so is rejected with the report *DEFN ERROR*. The following trouble reports may arise during copying:

WS FULL There is insufficient space to accommodate all the material to be copied. However, those objects copied before space was exhausted remain in the active workspace.

SYMBOL TABLE FULL New names occurring in the copied material exhaust the capacity of the symbol table. Those objects copied before the symbol table was exhausted remain in the active workspace.

SI DAMAGE A copied object has replaced the definition of a halted function, so that its halted execution cannot be resumed. The user should then enter → enough times to clear the state indicator of indications of damage.

)*PCOPY* <*LIBNO*> *WSNAME* [:*PASSWD*] < *OBJNAME* [*OBJNAME2* ...] >

The protecting copy command works in the same way as the unprotecting copy, described above, except that a global object having the same name as one already present in the active workspace is not copied. If any object is not copied for this reason, the system reports *NOT COPIED*: followed by a list of the objects not copied.

GROUPING TO FACILITATE COPYING OR ERASURE

It is frequently convenient to copy into the active workspace several related functions and variables, and to erase them when they are no longer needed. To facilitate such transfers, a group may be defined by supplying a list of names that are to be copied or erased together. Since the group's definition will exist in the workspace, it must have a name distinct from that of any global object in the workspace.

The definition of a group consists of a list of names. It is not necessary that objects having those names exist in the workspace. When a copy command or an erase command mentions the name of a group, then not only the definition of that group but also all global objects whose names appear in the membership list are copied (or erased, as the case may be).

If the membership of a group *A* includes the name *B* of another group, the act of copying *A* causes the membership list of *B* to be copied, but copying does not extend further. That is, the objects referred to in the membership list of *B* are not copied. The same applies to erasure. If an object named in the membership list of a group does not exist in the workspace, it cannot be copied or erased, but no report is generated.

)*GROUP GRPNAME* [*GRPNAME*] [*OBJNAME1 OBJNAME2* ...]

A group having the name *GRPNAME* and consisting of the membership list shown is formed in the active workspace. If

the name is already in use as the name of a group, the new membership list (possibly empty) supersedes the old.

If a group is given an empty membership list, it is dispersed. Dispersal of a group has no effects on its members (whereas the command *)ERASE GRPNAME* applied to a group expunges both the group definition and the members).

If the name of the group appears also as one of the members of the group, the former membership list is merged with the list provided in the command, thereby appending new names to the membership list.

The system makes no acknowledgment of a successful grouping.

An attempt to form a group having a name already in use as the name of a global variable or function in the workspace is rejected with the report *NOT GROUPED, NAME IN USE*.

Since the membership list of a group requires space within the workspace, the attempt to form a group may find insufficient space in the workspace or in the symbol table, resulting in the trouble reports *WS FULL* or *SYMBOL TABLE FULL*.

COMMANDS THAT MONITOR THE ACTIVE WORKSPACE

The following commands report aspects of the workspace environment, but produce no change in it.

)SYMBOLS

Elicits the report *IS ...* showing the current maximum number of symbols.

)FNS [A]

Reports a list of the global functions in the active workspace, in alphabetical order [starting with the letter indicated] .

)VARS [A]

Reports a list of the global variables in the active workspace, in alphabetical order [starting with the letter indicated] .

)GRPS [A]

Reports a list of the groups in the active workspace, in alphabetical order [starting with the letter indicated] .

)GRP GRPNAME

Reports the membership of the group named *GRPNAME*.

)SI

Displays the state indicator, showing the status of halted functions, with the most recently halted first. The list

shows the name of the function and the number of the statement at which work is halted. The actions that a user can take with respect to a halted function are described in Section 7.

Suspended functions are marked in the state indicator by an asterisk, while pendent functions appear on the state indicator list without an asterisk. The names of functions whose definitions have been damaged (by editing, copying, or erasing them while they are halted) are replaced by blanks in the state indicator.

)SIV

Displays the state indicator in the same way as *)SI*, but in addition, with each function listed, lists names that are local to its execution.

COMMANDS FOR WORKSPACE STORAGE AND RETRIEVAL

The user may request that a duplicate of the currently active workspace be saved for later use. When a duplicate of a saved workspace is subsequently re-activated, the entire environment of computation is restored as it was, except that variables which were shared in the active workspace are not automatically shared again when the workspace is reactivated.

LIBRARIES OF SAVED WORKSPACES

The set of workspaces saved for a particular user is called that user's library. Each workspace is identified by the user's account number and the name the user assigns to it. However, in referring to workspaces in one's own library, the account number may be omitted; the user's own number is supplied automatically.

In systems with multiple users, it is often convenient to use functions or variables contributed by others. One user may activate an entire workspace saved by another user, or may copy from it selected items. To do so both the library number and the name of the desired workspace must be supplied. However, the system provides no way of learning either the account numbers or the names of workspaces belonging to other users. Thus a user may make use of material from the libraries of others only if they supply that information. In no case may a user add, change, or delete material from the library of another user.

Certain libraries (usually identified by a particular group of library numbers) are not assigned to individual users, but are designated as public libraries. Any user may obtain a list of workspaces in a public library, and may use public workspaces. However, there may be restrictions on who can save, drop, or modify a workspace in a public library. In general, a public workspace can be re-saved or deleted only by the user who first saved it.

NAMES AND PASSWORDS FOR WORKSPACES

A saved workspace must be named. The name of a workspace may duplicate a name used for an APL object within the workspace. A password may be paired with the name of a workspace. Once this is done, any reference to the saved workspace (other than dropping it) must be accompanied by the password.

The rules governing what workspace names are permissible and what passwords are permissible may differ from each other and from the rules governing names within a workspace. In particular, the number of significant characters in a workspace name or in a password may differ, and may be less than the number of significant characters in a name used within a workspace. Workspace names and passwords may be composed of alphabetic and numeric characters, but not spaces or special symbols; workspace names must begin with an alphabetic, but this restriction does not apply to passwords.

)WSID <LIBNO> WSNAME [:NEWPASS]

Assigns to the active workspace the name indicated, and (optionally) the library number or the password indicated. Use of the colon with nothing following it assigns an empty password; that is, it removes a former password if there was one. If the active workspace is subsequently saved, future use of the saved workspace will require use of the password set here.

Setting of the active workspace's identifcation is acknowledged by the report *WAS ...* followed by the former name, but not the former password.

)SAVE < <LIBNO> WSNAME <:NEWPASS> >

A duplicate of the active workspace is saved (optionally, in the indicated public library, otherwise in the user's own library) under the indicated name, and (optionally) with the new password indicated. If the user number, workspace name, or password are omitted, they are supplied from the workspace identification. After saving, the active workspace has the same indentification (including account number, name, and password) as the saved workspace.

Although saving does not affect the state of sharing in the active workspace, current values of the shared variables are saved in the stored copy.

Saving is acknowledged by a report showing the date and time at which the workspace was saved.

The command to save the active workspace may be rejected, with trouble reports as follows:

IMPROPER LIBRARY REFERENCE The system does not permit a workspace to be saved in the private library of another user, or in a non-existent public-library, nor does it permit a workspace named *CONTINUE* to be saved in a public library.

NOT SAVED, THIS WS IS ... Saving is not permitted when the name given in the command matches the identification of an existing saved workspace but does not match the identification of the active workspace. This restriction prevents the user from inadvertently overwriting one workspace with another.

WS QUOTA USED UP Saving is permitted only while the number of workspaces currently saved or the space used by the saved workspaces is less than the user's allocation. Quotas are set by the system operator.

WS FULL The workspace contains a shared variable whose value, when brought in to the workspace, would require for its storage more work area than is available in the workspace.

NO SPACE There is insufficient space in the system's auxiliary storage to accomodate the workspace.

LIBRARY TABLE FULL The system's directory of saved workspaces lacks space in which to record this one.

The last two problems cannot be remedied by the user's action but require intervention from the system manager to expand the system facilities.

)CONTINUE [HOLD] <:NEWPASS>

The active workspace is saved under the name *CONTINUE*, and (unless a different password is specified) with the password previously assigned to the active workspace. Then the terminal is signed off in the same way as with the command *)OFF*, described below. Unless the *CONTINUE* workspace is password-protected, it will be automatically loaded at the next sign-on.

The system operator has the power to force a sign off ("bounce") some or all users. If a user is bounced, the system behaves as though the user had entered the command *)CONTINUE*. A *)CONTINUE* is also executed automatically if the user's connection to the system is broken without a normal sign-off; see the discussion of "Automatic Saving After Line Drop," below.

The command *)CONTINUE* may be rejected for the same reasons that apply to *)SAVE*, listed above.

)LOAD <LIBNO> WSNAME [:PASSWD]

A duplicate of the indicated workspace (including its entire computing environment) becomes the user's active workspace. Shared variable offers in the former active workspace are retracted. Following a successful *)LOAD*, the system reports the date and time at which the loaded workspace was last saved. The system then immediately executes the latent expression (*□LX*).

Invalid requests to load a workspace may result in the following trouble reports:

WS NOT FOUND The indicated workspace cannot be found.

WS LOCKED The password supplied in the command does not match the password of the saved workspace, or is missing from the command when required.

IMPROPER LIBRARY REFERENCE The user is ineligible to use the indicated library.

)DROP <LIBNO> WSNAME

The named workspace is removed from the user's library or from the indicated public library. The system reports the date and time. The password is not required to drop a saved workspace. Dropping a workspace has no effect on the active workspace.

An attempt to drop a workspace saved by someone other than the user is rejected with the report *IMPROPER LIBRARY REFERENCE*. Reference to a non-existent workspace is rejected with the report *WS NOT FOUND*.

AUTOMATIC SAVING AFTER LINE-DROP AND SPECIAL PROPERTIES OF THE *CONTINUE* WORKSPACE

If the connection to the computer or host system is broken before a normal sign off, and the active workspace is not empty, the system acts as though the user had entered the command *)CONTINUE*, thereby saving the contents of the active workspace under the name *CONTINUE*. The *CONTINUE* workspace is intended solely for temporary storage. Regardless of the name of the active workspace, it can under these conditions be saved under the name *CONTINUE*. That is, a previously saved workspace named *CONTINUE* may be replaced by the active workspace without the protection against overwriting provided for other workspaces.

Since any account may have a workspace named *CONTINUE*, to assure privacy the *CONTINUE* workspace of one user can not be loaded by another user, nor can it be saved in a public library. The workspace *CONTINUE* does not count as part of the user's quota of saved workspaces.

COMMANDS REGARDING WORKSPACE STORAGE AND RETRIEVAL

)WSID

Reports the identification of the active workspace, showing the library number if other than the user's own, and the workspace name, but not the password.

)LIB <LIBNO> [A]

Displays the names of the workspaces in the user's private library or in the indicated public library. In systems in which the optional letter of the alphabet can be used the display is alphabetized. If the command is followed by a letter of the alphabet, the display begins with names starting with that letter.

An attempt to display the list of workspaces in another user's private library or in a non-existent public library is rejected with the report *IMPROPER LIBRARY REFERENCE*.

ACCESS TO THE SYSTEM

Each user of the system is assigned by the system manager an account identification used to identify data storage and charges for use of the system. The account identification is required in order to sign on.

Each user is also assigned a quota indicating the maximum number or capacity of saved workspaces, and an interface quota indicating the maximum number of variables that may be shared simultaneously.

The system manager may also establish a cpu limit for an account, limiting the amount of central processor time per keyboard entry. When the limit is reached, an interruption is made in the same way as if a strong interrupt had been signalled from the keyboard.

SIGN ON

Before work can be started, a physical connection to the computer must first be established. This may require as little as turning a switch, or may require establishing a link from a dial-up terminal to a central computer, possibly passing through intermediary computing systems which are host to APL, depending on the system employed and the type of terminal device employed.

Once communication is established, a numerical identification of the user must be provided to the system. The precise manner of providing this varies from system to system.

At sign on, either a clear workspace or the *CONTINUE* workspace is activated, depending on the condition which terminated the preceding session, and the system in use.

Successful sign on is acknowledged by a message showing the port number, the date and time, the user's name, and the system identification. This may be preceded by a broadcast message from the system operator. If *CONTINUE* was activated, the system reports the date and time at which it was saved.

An attempt to sign on may be rejected, with one of the following reports:

ALREADY SIGNED ON The terminal is in use, and must first be signed off (or bounced by the system operator) before a new sign-on can be accepted.

NUMBER IN USE The indicated account number is in use at another port. This may arise either because another user is in fact using the account, or occasionally (perhaps because of an equipment problem) because the system did not complete disconnection at an earlier work session.

NUMBER NOT IN SYSTEM This report means either that the indicated account has not been enrolled, or that the password supplied does not match the password last set.

NUMBER LOCKED OUT Authorization for use of the account has been withdrawn. Workspaces saved in this library cannot be loaded or copied by anyone.

SIGN OFF

)OFF [HOLD] <:NEWPASS>

An accounting report is displayed, showing the port number, date and time, and user code, followed by the connect time and central processor time used at the current session and cumulatively since the last accounting.

If the word *HOLD* is used, the connection to the computer or host system is held for a brief period to accommodate sign on by another user.

If a colon is used, the word following it (or none) will be the password for subsequent sign ons.

)CONTINUE [HOLD] <:NEWPASS>

Because this command has the dual effects of saving the active workspace and signing the user off, it is described here and also in the section on workspace storage and retrieval.

The user is signed off in the same manner as with *)OFF*, but the active workspace is first saved under the name *CONTINUE*. If a password is set, it will apply at subsequent sign-ons.

The command may be rejected if it is impossible to save the *CONTINUE* workspace, either because the physical resources of the system are unable to accomodate it, or because of a full workspace produced when the value of a shared variable is transferred to the workspace. See the preceding discussion of the commands *)SAVE* and *)CONTINUE*.

D

E

F

G

H

I

T

U

APL Language

GC26-3847-0

Reader's Comment Form

Your comments about this publication will help us to improve it for you. Comment in the space below, giving specific page and paragraph references whenever possible. All comments become the property of IBM.

Please do not use this form to ask technical questions about IBM systems and programs or to request copies of publications. Rather, direct such questions or requests to your local IBM representative.

If you would like a reply, please provide your name, job title, and business address (including ZIP code).

Fold on two lines, staple, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

GC26-3847-0

Fold and Staple

First Class Permit
Number 439
Palo Alto, California

Business Reply Mail

No postage necessary if mailed in the U.S.A.

Postage will be paid by:

IBM Corporation
System Development Division
LDF Publishing—Department J04
1501 California Avenue
Palo Alto, California 94304

Fold and Staple

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

APL Language (File No. S370-22) Printed in U.S.A. GC26-3847-0

APL Language (File No. S370-22) Printed in U.S.A. GC26-3847-0

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)