Burroughs

B 6700/B 7700

APL/700

USER REFERENCE MANUAL

(RELATIVE TO MARK II.7 RELEASE)

$6.00

Printed in U.S. America

17 February 1975

5000813

Total pages in this manual is 232.

Page	Issue	Page	Issue
Title	Original	5-70 Blank	Original
A	Original	6-1 thru 6-22	Original
i thru vii	Original	7-1 thru 7-11	Original
viii Blank	Original	7-12 Blank	Original
1-1 thru 1-9	Original	8-1 thru 8-21	Original
1-10 Blank	Original	8-22 Blank	Original
2-1 thru 2-12	Original	9-1 thru 9-8	Original
3-1 thru 3-12	Original	A-1 thru A-7	Original
4-1 thru 4-13	Original	A-8 Blank	Original
4-14 Blank	Original	B-1 thru B-4	Original
5-1 thru 5-69	Original	C-1 thru C-16	Original
		Index-1 thru Index-12 .	Original

Burroughs believes that the information described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of the use of this material. The information contained herein is subject to change. Revisions may be issued to advise of such changes and/or additions.

Table of Contents

Section Title Page

INTRODUCTION vi

OVERVIEW vii

1 APL/700 SYSTEM DESCRIPTION 1-1

General 1-1
Properties and Features 1-2
Use Requirements 1-2
APL/700 Interactive Environment 1-3
Data Elements and Objects 1-4
Constituents of APL Language 1-5
    Constants and Variables 1-5
    Functions 1-5
    Primitive Functions and Operators 1-6
    Defined Functions 1-6
    Control Structures 1-7
    Expressions 1-7
User Account 1-7
Workspaces, Library and Files 1-8
    Self Protection 1-8
    Security and Sharing 1-9

2 INTERACTING WITH APL/700 2-1

General 2-1
APL Terminal Keyboard Configurations 2-2
    APL Character Set 2-2
    Typing Conventions 2-5
Connection with the APL/700 System 2-6
    Sign-On 2-6
    Transaction Entries 2-8
    Transaction Editing 2-8
    Correcting Typing Errors within Entry 2-8
    Editing Prior Transaction 2-9
    Sign-Off 2-11
    Recovery Operations 2-12

Table of Contents (Cont)

Section	Title	Page
3	SYSTEM COMMANDS	3-1
	General	3-1
	System Command Categories	3-1
	Session Controls	3-2
	Terminal Controls	3-3
	Clear Workspace Controls	3-4
	Library Controls	3-6
	Name Displays	3-9
	Erase Names	3-9
	Group Commands	3-10
	Run State	3-12
4	THE APL/700 LANGUAGE	4-1
	General	4-1
	Data Objects	4-2
	Names	4-6
	Expressions, Lists and Order of Execution	4-7
	Expression Formats	4-7
	Expression Lists	4-8
	Brackets	4-8
	Expression Entry	4-9
	Comment	4-10
	Input/Output Communicators	4-13
5	PRIMITIVE FUNCTIONS AND OPERATORS	5-1
	General	5-1
	Selection Function	5-2
	Assignment Functions	5-4
	Scalar Primitive Functions	5-6
	Integer Part, Extreme Value Functions	5-7
	Arithmetic Functions	5-8
	Power, Logarithm Functions	5-10
	Relational Functions	5-12
	Logical Functions	5-13
	Circular Functions	5-14
	Factorial, Combinatorial Functions	5-16
	Extension to Arrays of Scalar Functions	5-17
	Primitive Operators	5-19
	Outer Product Operator	5-20
	Reduction Operator	5-22
	Scan Operator	5-24
	Inner Product	5-26

Table of Contents (Cont)

Section	Title	Page
5	Identities for Scalar Dyadic Primitive Functions . .	5-28
	Mixed Primitive Functions	5-29
	Shape, Reshape Functions	5-30
	Integers, Index of Functions	5-31
	Ravel, Catenate, Laminate Functions	5-32
	Reverse, Rotate Functions	5-34
	Transpose, Permute Functions	5-36
	Compress, Expand Functions	5-38
	Take, Drop Functions	5-40
	Set Functions	5-42
	Grade Functions	5-44
	Random Functions	5-45
	Base Value Function	5-46
	Represent Function	5-48
	Matrix Inverse, Divide Functions	5-50
	Evaluate Function	5-54
	Format Functions	5-55
	Format Syntax Diagrams	5-56
	Default Format Function	5-58
	Numeric Format Function	5-60
	Character Format Function	5-62
6	SYSTEM VARIABLES, SYSTEM FUNCTIONS AND SHARED VARIABLES	6-1
	General	6-1
	System Variables	6-2
	System Functions	6-3
	Function Transformations	6-4
	Name Functions	6-5
	Debugging Aids	6-6
	Execution Controls	6-8
	Special Character Sets	6-9
	Status Inquiries	6-12
	Shared Variables	6-13
	Shared Variable Functions	6-15
	SV Offer, Coupling Functions	6-16
	SV Access Controls Functions	6-18
	SV Query, Retract Functions	6-20
	I-Bar Primitive Functions	6-22

Table of Contents (Cont)

Section Title Page

7 FILE SYSTEM FUNCTIONS 7-1

General 7-1
File Name 7-1
File Components 7-1
File Limits 7-1
File Opening, Active and Inactive Status . . . 7-2
File Integrity 7-2
File System Primitive Functions 7-2
File Create, Change Password, Rename, Destroy . . 7-3
File Component Null, Write, Read 7-4
File Component First In, Out; Last In, Out . . . 7-5
File Component Order Reverse, Rotate 7-6
File Components Take, Drop 7-7
File Components Compress, Expand 7-8
File Hold, Free, Release 7-9
File Component Existence 7-10
File System Interrogate, Status, Query 7-11

8 FUNCTION DEFINITION, EDITING AND EXECUTION 8-1

General 8-1
Function Content 8-1
Branch, Terminate, Label 8-3
Function Editing Actions 8-5
Function Define, Open, Close Actions 8-6
Function Line Replace, Insert Actions 8-8
Function Line Edit Actions 8-10
Function Multiline Group Specifier 8-12
Diagnostic Function Line Group Actions 8-14
Display Function Line Group Actions 8-16
Delete Function Line Group Action 8-17
Defined Function Execution 8-18
Scope of Names 8-18
Execution Control Sequence 8-18
Multiple Instances 8-19
Recursive Functions 8-19
Suspension of Defined Function Execution 8-20
Defined Function Editing Using APL Functions 8-21
Defined Function Documentation 8-21

Table of Contents (Cont)

Section	Title	Page
9	ERROR REPORTS AND THEIR INTERPRETATION	9-1
	General	9-1
	Error Reports	9-1
	Unimplemented Constructs	9-8
APPENDIX		
A	Glossary	A-1
B	Workspace Content Space Considerations	B-1
C	Reference Charts	C-1
INDEX	. . . .	Index-1

List of Illustrations

Figure	Title	Page
1-1	Transaction Cycle	1-3
2-1	APL Terminal, Typical 88-Character Keyboard	2-3
2-2	APL Terminal, Typical 94-Character Keyboard	2-3

List of Tables

Table	Title	Page
4-1	Examples of Data Object Forms	4-3
4-2	Tests for Properties of Data Objects	4-5
5-1	Identities for Scalar Dyadic Primitive Functions . .	5-28
6-1	Character Representation Order in Atomic Vector . .	6-11
9-1	Error Reports	9-4

INTRODUCTION

APL/700 is comprised of A Programming Language (APL) and the interactive environment in which the language is used. APL is a general purpose language for describing procedures concisely and consistently. These procedures are then used to process information. Capabilities common to APL systems include:

Terminal transaction-oriented processing
Many built-in primitive functions
Array data-objects as arguments
Direct expression entry and evaluation
User defined functions

APL/700 incorporates these capabilities, and in addition includes many exclusive features for more power and versatility:

Extended function capabilities
Improved terminal interaction
Comprehensive formatting capabilities
Enhanced function editing
Integrated file system
Explicit error reporting
Inter-process variable sharing

This APL/700 User Reference Manual, Form 5000813, contains complete information for the user.

The APL/700 Reference Card, Form 1079936, provides a syntactic summary of the material in this manual.

The B6700/B7700 APL/700 Installation Manual, Form 5000805, addresses the internal details necessary for a site to install, run, and manage APL/700 for its users. It contains no information for the APL/700 user.

Documentation for specific APL/700 applications is released with the applications.

Documentation of the APL/700 system has been prepared and is maintained using TEXTEDIT (c) 1974 Burroughs Corporation. TEXTEDIT is an APL/700 application.

OVERVIEW

The intent of this manual is to provide sufficient reference data (definitions, instructions, and examples) to help the user to understand and apply APL/700. The manual is organized into 9 sections, 3 appendices and an index. Each section covers an independent aspect of APL/700.

Section 1 summarizes from the user viewpoint APL and its environment.

Section 2 explains user interaction through a suitable terminal for APL.

Section 3 describes the system commands provided to express the user's control over the APL environment.

Section 4 introduces the general properties of the APL language: its array data objects, names and expressions.

Section 5 details the primitive functions and gives examples of their application to data objects.

Section 6 shows the system variables to specialize the processing; system functions to query or alter the environment of the account; and shared variables for interprocess communication.

Section 7 defines the file system functions for workspace extension.

Section 8 illustrates the actions provided for function definition and editing, and also their execution.

Section 9 lists the error reports displayed as they are detected for immediate repair and resumption of processing.

Appendix A is a glossary.

Appendix B gives techniques for control of memory space.

Appendix C contains a set of summary reference charts for the material detailed in Sections 1 through 8.

The Index includes terms and concepts used in this manual. It also includes terms used in other APL books and manuals.

The reader is encouraged to become a user from the start; the interactive environment allows problem formulation and solution at the user's pace.

SECTION 1

APL/700 SYSTEM DESCRIPTION

GENERAL.

APL/700 is an interactive tool for problem solvers. One purpose is to provide a means for the person formulating a problem solution to obtain desired results quickly. The user works through a terminal. Solution formulation and data entry can be intermixed. Entered information and returned results may be displayed for immediate review. APL is especially appropriate where user insight is important during solution development. APL encourages experimentation, the asking of "what if..." questions, and focusing upon immediate needs. This contrasts with traditional bulk data processing, where massive outputs are prepared in hope that somewhere therein can be extracted the answers to any potential questions.

Problem formulation can often be in terms of an immediately executed APL expression for which direct response is provided. APL/700 has many powerful built-in functions available for this use. These apply consistently to either simple data or array structured data. Uniform, parallel processing of all elements in a data structure permits significant algorithms to be concisely expressed, with irrelevant detail suppressed.

A problem solution can be developed in a logically structured manner (top down). It can be saved for later use. Progressive refinements can be easily incorporated. The data required can be kept in variables and the calculation sequence required can be retained in one or more user defined functions. Further, a file system is available to allow a problem solution to be easily extended to handle an unlimited quantity of data.

A second purpose of APL and its interactive environment is to provide a hospitable host for applications. The users of these in many cases need not know APL in detail. Many successful APL applications exist:

Financial analysis
Inventory control
Manufacturing scheduling
Forecasting
Manpower management
Resource control
PERT
Reservation control

Text processing and documentation
Report generation
Message processing and distribution
Statistical analysis
Mathematical analysis
Simulation and optimization
Computer aided instruction
Data base search and retrieval

The common property of these applications is their use of direct input and immediate display response. Traditional computation-bound applications may often be re-cast into APL to provide a more satisfactory solution for the user with the problem.

PROPERTIES AND FEATURES.

APL/700 may be characterized as:

accessible	immediate response for "trivial" requests
unobtrusive	problems quickly solved at user's pace
concise	powerful primitive functions on data structures
simple	consistent, few rules
readable	define functions in few lines
forgiving	easy error correction, good recovery
secure	protection for private or shared work

Features that make APL/700 an effective interactive system include:

built-in APL functions for processing data
expression entry and immediate execution
progressive expression development by augmenting prior entry
data entry in execution or input modes
user function creation in definition and editing mode
file system for accessing auxiliary data
shared variables for interuser or interprocess communication
formatting functions for report preparation
system functions and commands to query and alter environment
keyboard input and display controls

USE REQUIREMENTS.

To use APL one needs only:

a terminal with APL characters
an account on an APL/700 system

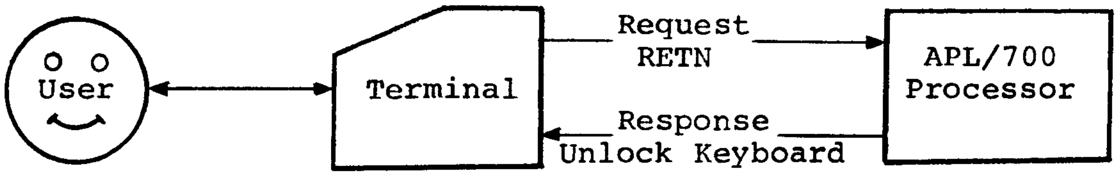
Note that typing skill is not on the above list. APL is so concise that lack of typing skill is not a significant barrier. Since the reader is encouraged to learn APL on a terminal, keyboard familiarity develops with use.

The APL/700 system cannot be damaged by user entries. The user quickly learns to experiment: when in doubt, try it.

APL/700 INTERACTIVE ENVIRONMENT.

The user seems to have exclusive use of the APL/700 processor. This illusion can be maintained for many users concurrently since the amount of computer resources required for servicing any one user is usually a small fraction of the total resources available. Peak requirements are spread in time.

A transaction is the alternating cycle starting with a user phase followed by a processor phase. The user phase starts with the terminal ready for user typing, continues through user typing an entry requiring service and completes with the return (RETN). Then the processor phase starts by receiving the RETN, provides the service required by the entry, possibly generating output, and finally makes the terminal ready for next user entry.



	User Phase		Processor Phase	
Action	Type Entry	RETN	Process	(Output)
Keyboard	Unlocked		Locked	
Typical Time Span, Seconds	1 to 30		0 to 1	0 to 6

FIGURE 1-1 TRANSACTION CYCLE

The user sets the work pace; the processor rarely slows the thought process. When the time consumed during the user phase and during output from the processor phase constitutes a large fraction of the transaction cycle, the user has the illusion of a dedicated computer system. APL/700 achieves this by scheduling "short" requests (taking no more than a fraction of a second of processing to complete) for almost immediate service. "Involved" requests (that a user might expect to take a while) are scheduled for processing that can be interrupted as necessary to service short requests from other users. Most requests are short.

The benefits from sharing the APL processor among many users concurrently include:

immediate response for short transactions
work smoothing among many irregular demands for service
powerful processor available when needed
cost spread across users as resources are used
"think time" need not be penalized
data files for data accumulation and shared access

USAGE MODES.

The user of APL may select one of three modes for use at any time. Each mode is recognizable by the prompt or appearance on the display when the keyboard is unlocked.

Execution (or Calculator) Mode.

immediate execution of entered expressions
progressive expression development by altering prior entry
assignment of values to variables
call on defined functions for execution
prompt: five space indent

Data Entry Mode.

evaluated, in response to the prompt ⎕:
character, in response to a user established prompt

Function Definition and Editing Mode.

creation and editing of defined functions
establishment of automatic debugging aids
prompt: [n] at left margin for line n of the open function

DATA ELEMENTS AND OBJECTS.

Data objects are the units for processing. A data object has the properties of type, shape, and value.

The type of a data object is either:

character	any APL characters
numeric	any value representable as a number

The shape of a data object is a vector of non-negative integers indicating the lengths along each dimension of the object. A data object may be a scalar, in which it has a single element with an empty shape (a geometric point). A data object may be an array of some number of dimensions with a shape vector. If there is only one dimension, the array is referred to as a vector. The right most element of the shape vector is the number of columns in the object. A two dimensional array is referred to as a matrix. The shape of a matrix is the number of rows, followed by the number of columns. The rank of an object is the number of dimensions.

The value of a data object may consist of a single scalar element or zero or more elements arrayed in some rectangular manner.

CONSTITUENTS OF APL LANGUAGE.

The APL language includes four kinds of entities:

constants and variables
functions
control structures
expressions

CONSTANTS AND VARIABLES.

A *constant* is a data object without a name. Constants can appear as part of defined functions or can be entered as part of execution mode expressions.

A *variable* has a name that is attached to a data object by assignment. The name is used in APL expressions as a reference for the associated value of an APL data object. Each successive assignment to a variable name attaches a new data object to it. Special system variables provide access or control over variables relating to the APL environment. Shared variables permit interprocess communication.

Constants and variables can be used as arguments to functions in APL expressions.

FUNCTIONS.

Functions perform processing according to particular, defined rules. Many *primitive functions* of general utility are built-in to APL. Other functions can be created by the user to solve problems. These are called *defined functions*. They are defined in terms of other language constituents.

A function accepts *arguments* and generally returns a value, as a result of following the processing rule for that function as applied to its argument values.

A function is defined for a *domain* of values for each of its arguments and produces a result in the allowable result *range* of values. For example, the relational function "less than", as used in:

A "less than" B

has numeric domain for arguments A and B and the values true and false as the range of values for the result.

In APL, "less than" is expressed by the character '<', and the values true and false are expressed by the Boolean numeric values 1 and 0 respectively:

```
      3<5   an entry (made following the 5 space indent prompt)
1           the result response (the relation is true)
```

Primitive Functions and Operators.

Complete families of primitive functions are provided for numeric type data objects:

arithmetic functions
relational functions
logical functions
higher functions
random number functions

A group of operators exist which act upon primitive functions to produce new functions which then apply to data.

Additional function families exist that apply to both numeric or character data types:

structure building and changing functions
mixed type functions
set functions
selection functions
assignment functions
formatting functions
input output communicators

A file system provides convenient access to extensive data using a set of file functions.

A set of system functions permits querying and altering the environment within which APL is used. There also exists a similar set of system commands that can be used only in execution mode.

Shared variable functions are provided for controlled interprocess communication between a user and one other process, either another user or a shared variable utility.

Defined Functions.

A defined function performs more complex processing than can be done by single primitive functions. It contains one or more lines. Each line combines primitive functions, operators, constants, variables, references to defined functions, labels, punctuation, and control structures.

A defined function can have arguments. Arguments provide the values to use during its execution.

A defined function may optionally return a result from execution. If so, the defined function can be used to compose expressions in a similar manner to how primitive functions are used.

CONTROL STRUCTURES.

The APL control structures determine the order of execution. A primitive function generally applies "in parallel" to all elements of the data objects that are its arguments. A function is elaborated after its argument values are determined. Elaboration order is right to left within an expression. Lines within user defined functions are normally executed in sequence. Non-sequential execution may be achieved by explicit transfer to a line number, which may be computed.

If a user-defined function is called within an expression, control is passed to the called function. Subsequently, control is returned to the calling expression after the point of call. A function may be called recursively.

There are no formal conditional or iterative control structures for user defined functions. When required, these control structures are synthesized by explicit control transfers. The need for these may be generally avoided by mutually exclusive processing logic on elements of data structures.

EXPRESSIONS.

An expression is the syntactically correct composition of one or more APL language constituents. The results of elaborating an expression include change to the state of processing, or display to the user, or both. The constituents of APL expressions may include:

data objects (constants or variables)
primitive functions and operators
calls on functions defined by the user
file functions
system functions
system variables
control structure delimiters

USER ACCOUNT.

Each user must be assigned a valid account by the installation. This account collects usage information. The attributes of an account include:

account name and optional user-supplied password
workspace quota
file number quota
file space quota
computer use quota
shared variable quota

WORKSPACES, LIBRARY AND FILES.

Each user account has an active workspace. The active workspace is the fixed size area of storage in which a user conducts transactions. At first sign-on, this workspace is unnamed and clear. At this time, only the default values for system variables exist as previously established for the account. After some transactions, the workspace may contain some variables having values, some groups, some altered values for system variables, and some defined functions having continuing use.

A user can name the active workspace and save a copy of it in the account library of inactive workspaces for subsequent reactivation. The number of workspaces in the user library is limited to the quota established by the installation for that account. All workspaces have the same size, determined by the installation.

Within a workspace are all retained variables, defined functions, and temporary storage required during processing. The conciseness of APL defined functions permits a large processing capability within a workspace.

Each account may also have a quota of files. Each file has a name and a set of numbered components. Each component is either null (having no content) or contains an APL data object. Data objects can readily be exchanged with the active workspace. Defined functions can be represented as data objects and stored in file components. They can be accessed as needed and reconverted into function form. This increases the amount of data that can be processed by functions in a workspace.

SELF PROTECTION.

The active workspace contains current work. Whenever desired in execution mode, a copy of that workspace can be saved in the library for subsequent resumption with the processing state the same as at the point of saving.

Changes to function definition or experimental computation can be done, then either kept if good, or discarded by returning to the formerly saved version of the workspace.

The active workspace is retained in the event of unexpected disconnection caused by either the terminal, the communications link, or the main system. Upon next sign-on for the account, recovery occurs automatically to within the last entered transaction if in entry phase, or to the last line processed if in processor phase.

The commands having irrecoverable effects tend to be separated and protected against accidental misuse. For example, the user can ERASE names of variables, functions or groups, but must DROP a workspace.

SECURITY AND SHARING.

Protecting an account, its workspaces and files from other users is important. Locks and passwords provide these capabilities. Selective sharing of workspaces and files among accounts is often desirable. A user can grant access privileges to those he wishes, and deny privileges to all others.

A defined function can be locked so that it can only be opened for examination in the account and workspace in which it was locked.

A user account name is unique to the installation that assigns it. It is not considered private, but only a means for identifying the account when signed on the system, and for other users to reference the inactive workspaces and files retained for it.

The account user can add a distinct password for the account, and to any of its workspaces or files. Password use can provide a degree of security, since the assigner of that password controls its dissemination. A password can be entered or changed at any time through the terminal. A blot can be requested to obscure by overprinting the area in which password entry will appear. Of course, no security is provided against someone tapping the communications line connecting the terminal with the APL system, or against failure to blot display of password.

A user cannot alter a workspace saved in another account library; only a copy of it can be obtained (assuming that the account owner has divulged the account name and workspace name, and password if any).

A user can alter any file in the APL file system, given knowledge of the owning account/file name (and password if any). To control accesses to shared files, the owner should provide a locked file access function through which all accesses to the file are made. In this function, the file password can be secured from disclosure and necessary access conditions can be checked. Thus, the file name and password need never appear in visible form to the user.

When a file is shared among several users, each user can make conflict-free component updates by requesting exclusive use during the update operation.

If a user "forgets" a password, a request to the privileged terminal, if convincing, can result in administrative granting of one action by the user without the password. This action should replace the forgotten password. The privileged user does not know either the forgotten or new password. Administrative abuse of this privilege will be detected by the user, as the next attempt to use an old password will not work.

SECTION 2

INTERACTING WITH APL/700

GENERAL.

The APL/700 system communicates with the user in an interactive manner. The user can direct the system to execute, (or edit) expressions or defined functions. The user, through the keyboard, supplies data and instructions for processing that data. The order in which the characters in an entry are typed is irrelevant; the final image of that entry is used by the system. This property is called visual fidelity.

Typing errors can be easily corrected at any time before the end of a transaction entry. Further the most recent expression entry can be retrieved for editing and reentry.

During expression or function execution, the user may halt processing and examine and possibly modify the current execution state (all of the variables and the environment). APL provides debugging tools to allow the user to follow the execution process in as much detail as desired.

The APL user environment consists of an available library of workspaces and files, accounting information, and account parameters (print line width, tab interval, print precision, and index origin). The user can establish, query and alter this environment at any time.

Interaction with APL requires a terminal which should have the special APL typeface and keyboard configuration. Such a terminal must have provisions for communicating with the system Data Communications Processor. The DCP can be programmed to communicate with any standard printing and video APL-type terminals.

This section describes procedures for sign-on, sign-off, and transaction entry editing. The procedural instructions presented assume the particular terminal configuration described in this section.

Instructions for using an acoustically coupled telephone interface with the Data Communication Processor are given; procedures for other connection means are generally simpler.

APL TERMINAL KEYBOARD CONFIGURATIONS.

Figure 2-1 shows the configuration of the most commonly available APL terminal keyboard. The terminal has 44 keys, each containing two characters (shifted and unshifted), 10 special keys/bars, and an on-off switch. Recently produced APL terminals contain 47 character keys as shown in figure 2-2.

APL CHARACTER SET.

The APL character set consists of the 26 uppercase letters, digits 0 through 9, standard punctuation and special APL characters. Some of the conventional characters are not in normal typewriter keyboard locations, but are more logically grouped. All keys contain unique characters. Since APL uses more characters than there are keys and cases, some characters are formed by overstriking.

The APL character set used throughout this manual, is the one provided for typical (standard) APL terminals. Character appearance for other terminals varies somewhat in form. For example upright block letters are used on some terminals.

Letters have uppercase, italic form:

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 full set of underscored letters may also be used; those letters are formed with a non-underscored letter, a backspace, and an underscore (shifted 'F').

A̲ B̲ C̲ D̲ E̲ F̲ G̲ H̲ I̲ J̲ K̲ L̲ M̲ N̲ O̲ P̲ Q̲ R̲ S̲ T̲ U̲ V̲ W̲ X̲ Y̲ Z̲

Numerals have upright form:

0 1 2 3 4 5 6 7 8 9

Other characters are included that are generally upright:

¨ ¯ < ≤ = ≥ > ≠ ∨ ∧ - + ÷ ×

? ⍵ ∊ ⍴ ~ ↑ ↓ ⍳ ○ ⋆ → ←

⍺ ⌈ ⌊ _ ∇ ∆ ∘ ' ⎕ ([ )]

⊂ ⊃ ∩ ∪ ⊥ ⊤ | ; , : . \ /

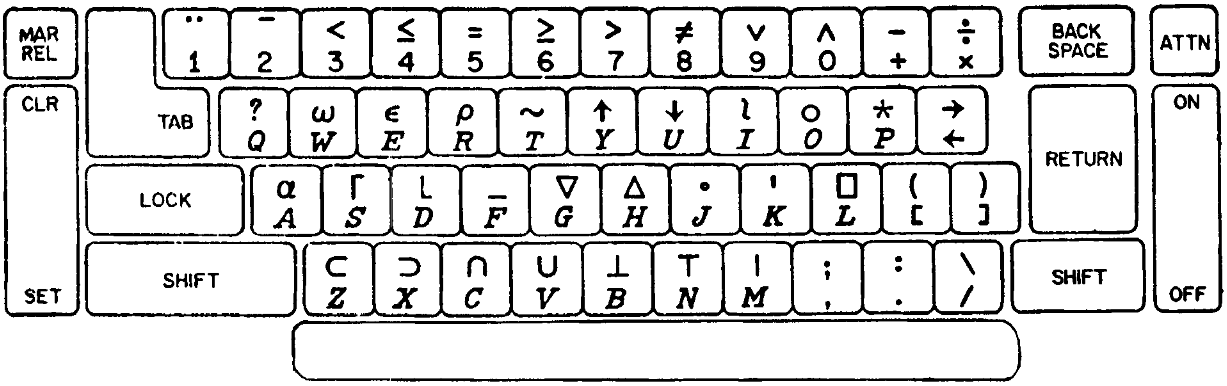


Figure 2-1. APL Terminal, Typical 88-Character Keyboard

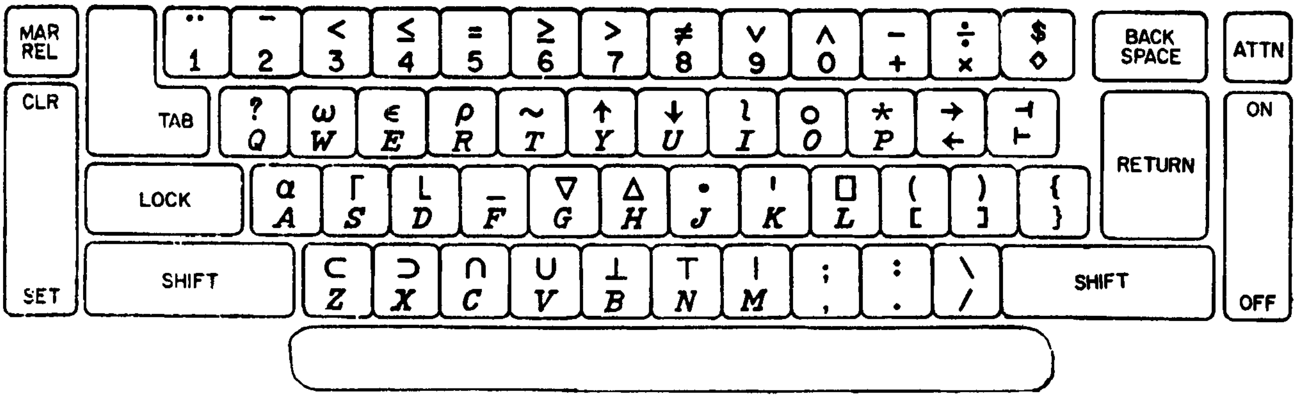


Figure 2-2. APL Terminal, Typical 94-Character Keyboard

Additional characters defined in APL are formed by overstriking:

⌶ $ ⍙ ⍲ ⍱ ⌿ ⍀ ⊖ ⌽ ⍉ ⍟ ⍎ ⍕ ! ⍋ ⍒ ⍫ ⍝

⍞ ⌹ ⍍ ⍔ ⍓ ⍌ ⎕̴ ⍃ ⌸ ⍄ ⍯ ⍇ ⍈ ⍐ ⍗ ⌻ ⌼ ⍂ ⍁

The order in which characters are overstruck is not important.

Overstruck ($) is not necessary on those 47-key configurations having the dollar ($) sign.

The essential special keys and the result of pressing them are:

Return or RETN — The return key signals the system that a user entry is complete and ready for processing. The cursor returns to the left margin and the keyboard is locked initiating the processor phase.

Shift or SHIFT — Any character key normally produces the lower character for that key. While SHIFT is depressed, the upper character for the key is produced. The shift lock can be used to keep the shift key depressed.

Space or SPACE — The space bar positions the cursor one space to the right; holding the space bar on some terminals causes repetitive spacing.

Backspace or BKSP — The backspace key positions the cursor one space to the left. On some terminals repetitive backspacing is accomplished by pressing and holding backspace key.

Attention or ATTN — The attention (interrupt or break on some terminals) key provides for initiation of special processing. Its uses include:

correction of transaction entry error
display and adjustment of the previous line
output termination
execution suspension

Local/Communicate — The terminal must be in the remote or communicate position to use the APL system. Local may be used for off-line typing, without disconnecting the APL use. (Switching between local and communicate may transmit a spurious character that can be eliminated by BKSP, ATTN).

Other convenience keys available on some terminals include:

Linefeed or LF
The linefeed (index on some terminals) key provides line advance and in-line edit correction similar to ATTN without the displayed caret.

Repeat or REPT
The repeat key provides repeated, automatic typing for any character.

Tabs or TAB
The tab key positions the cursor rightward to the next tab stop. To take advantage of the APL/700 tab conventions, the tabs should be set at constant intervals (such as every five characters).

Tab SET/CLR
The appropriate end of this key sets/clears a tab at the current cursor position. On some terminals, tabs may be cleared by positioning the cursor all the way to the right, holding the clear, and pressing RETN.

Margin
The margin key allows escape beyond mechanical cursor limits for display.

TYPING CONVENTIONS.

Except for different character key locations and certain special rules, the APL keyboard can be used in the same way as any other typewriter keyboard.

The following conventions apply:

User Entry
A user can type only when the keyboard is unlocked (the APL system locks the keyboard, preventing further entry when processing a user input or displaying response). Display of user entry is normally preceded by a prompt (5 character indentation in execution mode). The prompt helps to differentiate user entry from system responses (which normally start at the left margin).

Visual Fidelity
It is not necessary to type characters from left to right; an entry is interpreted by the system only after RETN. Backspacing allows the typing order to be arbitrary. That is, the time sequence in which the various keys are typed doesn't matter; the system interprets the entry as it appears on the terminal.

Entry Length
Each user entry should fit on a single display line to preserve visual fidelity. Some terminals with limited length buffers for character assembly may lose excess characters.

CONNECTION WITH THE APL/700 SYSTEM.

The elementary steps to use APL include sign-on, a sequence of transactions, and sign-off.

SIGN-ON.

The following procedure assumes the use of an acoustic-coupler for the telephone communications interface. Minor variations to the procedure may be required for other means of terminal connection.

1. Turn on the terminal and the acoustic coupler.

2. Lift the handset from the telephone cradle, dial a valid computer telephone number, and listen for a high-pitched tone.

3. When high-pitched tone is heard from computer, place the handset in the acoustic coupler so that the cord end of the handset is on the end of the coupler marked CORD.

4. Wait for a connection response from the computer. A typical response is:

   ```
   ON-LINE TO APL/700, YOU ARE:  1234567 (LSN:6).
   ```

   Where: 1234567 and (LSN:6) are station name and logical station number, respectively.

   If necessary, press the ATTN key several times, a second or so apart until a response is received.

5. Specify the APL Message Control System (MCS) by entering:

   ```
   \APL
   ```

   The backslash in the leftmost column signifies a message to the MCS which hosts APL.

6. Wait for the APL system prompt (cursor indents five-spaces) then enter the system command:

   ```
   )ON Acct [Password]
   ```

   Where: Acct is user account identification.

   The Password is an optional entry. It is required for a previously locked account; omit the Password and enclosing brackets if the account is unlocked.

   Example:

   ```
   )ON TERRY[HAPPY]
   ```

7. Press RETN and wait for the system sign-on response, which has the typical format:

```
FRIDAY 74/08/30 11.51 AM [V27000 W00120 T00 S006]
```

<u>Where</u>: V27000 is the version of APL/700 being used.

W00120 is the terminal width (maximum number of character positions per line) assumed for the user account before automatic display line folding occurs.

T00 is the terminal tab interval assumed for the account.

S006 is the number of the station to which the terminal is connected (used for communication purposes).

8. An optional news line may be displayed as determined by the APL system management. A typical instance might be:

```
SYSTEM OPERATION TODAY 8.00 AM TO 12.00 MIDNIGHT
```

9. Observe the system prompt (five-space indention) and keyboard unlock indicating completion of connection and readiness for transaction entry in execution mode. A light may exist that gives visual indication of keyboard unlocked condition.

The entire sign-on sequence appears on the display as:

```
ON-LINE TO APL/700, YOU ARE:  1234567 (LSN:6).
\APL
     )ON TERRY[HAPPY]
FRIDAY 74/08/30 11.51 AM [V27000 W00120 T00 S006]
SYSTEM OPERATION TODAY 8.00 AM TO 12.00 MIDNIGHT
```

TRANSACTION ENTRIES

When the system sign-on process is completed, transactions (cycles consisting of user entry and system response) may be initiated:

1. Make certain that the APL system has initiated the transaction cycle (by unlocking the keyboard).

2. Using the character keys, TAB, BKSP, and SPACE, type the desired entry. For example, to set the active workspace identification to CINDY, type:

   ```
   )WSID CINDY
   ```

3. After the transaction is completely typed, press RETN to complete the entry and initiate processing.

4. Wait for the system to provide any required display response. Such a response will generally start at the left margin. The response can be a transaction result, an error report, or a special prompt.

After any displayed response, a prompt is given and the keyboard is unlocked to complete the transaction and enable the next transaction entry. Repeat steps 2 through 4 for each subsequent transaction.

If an error message is received, make the appropriate correction and re-enter the transaction. (Refer to Section 9 for error-message descriptions and to the paragraphs describing editing and recovery procedures in this section.)

TRANSACTION EDITING.

There are a number of variations in performing transaction editing in the APL/700 system. The procedures required for editing depend on the mode of operation, on the state of the keyboard (locked or unlocked), on whether an ATTN entry is initial or non-initial, and on the type of editing required.

CORRECTING TYPING ERRORS WITHIN ENTRY.

A typing error may be corrected if it is noticed before the transaction text entry is completed:

1. Using BKSP and SPACE, position the cursor at the left-most character that is in error.

2. Press ATTN. The system will display the down caret (∨) under the character backspaced to in step 1 and then advance the display one line. (This action eliminates from the entry the characters above and to the right of the caret.) The INDEX key (if available) can be used instead of ATTN, however, the line with the down caret is not displayed.

3. When system response is completed, (keyboard unlocked), type the remainder of the transaction entry.

Example:

```
)ON MYACCT [KEYLOAD]         backspace under A, ATTN
                 v           correction mark
                 CK]         entry correction, then RETN
```

The corrected entry *)ON MYACCT [KEYLOCK]* is now entered by the RETN.

EDITING PRIOR TRANSACTION.

APL/700 has provisions for retrieving the most recently entered transaction and modifying it. This may be used to develop a computational expression, or in response to an error message or wrong result. The procedure for applying transaction editing is:

1. Without entering anything else, and with the cursor at the prompt position, press ATTN.

2. The previously entered transaction entry is displayed and the cursor returns to left margin.

3. Type edit characters below the displayed characters (spacing the cursor accordingly):

   "/" Each slash causes deletion of character above it.

   "." Each period segments display into another phrase, starting with the character above the period. (The first phrase starts at the left of the line, the last terminates at the end of the line.)

4. Terminate edit line by pressing RETN.

5. A phrase will be displayed with slashed characters deleted (up to the next period of the edit line, or the entire remaining phrase if no period is used).

6. Alter or augment the displayed phrase.

7. Enter another ATTN at the right-most position of the (possibly altered) phrase to display the next phrase (an ATTN not at the right-most position is used for intra-phrase error correction).

8. Repeat steps 6 to 7 until the entry is complete. A transaction entry is completed by entering RETN.

Any character entered but not recognized by APL/700 results in a "CHARACTER ERROR" report. An example of this and subsequent editing that also includes revision is:

```
          'THIS IS A BAD LINE.'         invalid characters
     *** CHARACTER ERROR ***            error message
          'THIS ⌷⌷ A B⌷D LINE.'         display with invalids marked
                //. ////                ATTN, enter edit characters
          'THIS IS NOW A LINE.'         ATTN after typing "NOW"
                         v              backspace to "L", ATTN
                         FIXED LINE.'   completed entry - RETN
     THIS IS NOW A FIXED LINE.          display of entry
```

All invalid characters are replaced by the "squish-quad" ⌷ display character. Entry of ATTN exdents to the left margin. Editing can now be started at step 3 of the transaction editing procedure.

If RETN is pressed during the above sequence while one or more of the phrase delimited by the '.' in the immediate edit line have not been displayed, those phrases are lost. RETN <u>completes</u> the entry.

Entering any character other than a slash or period below the characters of a line re-displayed for editing results in the following error message:

```
     *** EDIT ERROR ***
```

Pressing ATTN reinitiates the transaction editing sequence.

The use of ATTN for in-process typing corrections does not conflict with the applications described above. That is, for within-entry typing, the cursor is not at the right-most entry position when ATTN is pressed.

An ATTN can be used to interrupt the display of a line for immediate edit (step 2). This display is frequently already present, so an ATTN can save time. The result is a cursor return to the left margin, ready for step 3.

If a character error occurred as part of an entry given in response to prompted character input, the system will first display the error message, then the erroneous line, and exdent to the left margin ready for step 2. Entry of ATTN there causes a "twitch" prompt (3 spaces and 3 backspaces) to be returned, again ready for step 2. Another ATTN moves the cursor to the right end of the erroneous line.

SIGN-OFF.

When all user transactions are completed, or when it is necessary to temporarily interrupt operations at the terminal, sign-off from the system:

1. Make certain that an execute mode (5 blanks) prompt has been displayed and that the keyboard is unlocked.

2. Type one of the following sign-off system command entries, followed by RETN to terminate the work session:

)OFF	discards the active workspace
)COFF	preserves the active workspace to continue later

3. The usage record for the account will then be displayed:

```
          )OFF
     THURSDAY 74/02/01 12.47 PM
     CONNECTED 00.55.48  TO DATE 02.06.20
     CPU TIME  00.00.22  TO DATE 00.01.09
     IN APL-MCS
```

   This response indicates the type, time and date of sign-off; the four other numeric responses indicate time (hours, minutes, and seconds) spent on the current session, plus the total time to date for connection and CPU usage.

4. Another account can be signed on at this point. Start with step 5 of the sign-on actions.

   Otherwise, remove the telephone handset from the acoustic coupler and return to the telephone cradle.

5. Turn off terminal and coupler power as required.

RECOVERY OPERATIONS.

The APL/700 system provides automatic recovery from temporary work session interruptions, accidental disconnections, or system malfunctions. For any of these, or when a user signs-off from the system for a temporary work session interruption by using *)COFF*, the active workspace is retained for use when the next session is initiated on that account.

When the active workspace is preserved from a session and the account is signed-on, the system responds with the normal sign-on display. An additional statement may include the name of a preserved workspace *WS* Name and the time and date from which it was continued.

```
      )ON TERRY [HAPPY]
WEDNESDAY 74/01/30  11.18 AM [V2700 W00130 T05 S018]
WS CINDY CONTINUED FROM 74/02/30 11.02.33
```

It is possible that an accidental disconnection or system malfunction will occur during a work session. In either case, the system will automatically preserve the active workspace and provide a *CONTINUED* message when the account is again signed-on.

If execution was interrupted, then the word *EXECUTION* will appear between the active workspace name and *CONTINUED* The execution will continue until the line being executed is completed; then if in a defined function the function name and line number are printed, followed by an asterisk '*' to indicate that the function is suspended. The system then types an input prompt and waits for a transaction entry.

If a function was being defined when a work session was interrupted, the word *DEFINITION* appears between the workspace name and the word *CONTINUED* in the message. A function definition prompt is then returned to enable continuation of the function definition. An accidental interruption that occurs while an entry is being composed results in the loss of that entry.

If the continued active workspace had not been named, the *WS* Name is omitted.

SECTION 3

SYSTEM COMMANDS

GENERAL.

APL/700 has a set of special instructions called system commands. These commands deal with such practical matters as signing onto and off of the system, saving workspaces, setting default control values, copying workspaces, functions, or variables, and controlling terminal functions. These operations are only initiated in execution mode; they can not appear as part of a user defined function. A system command is executed immediately after being entered (if possible).

SYSTEM COMMAND FORMAT.

The conventions used to describe the system commands are chosen to allow ready recognition of the fixed and variable; required and optional parts.

Convention	Meaning
)	system command prefix
[ ] ( ) /	separators -- matching pairs for [ ] and ( )
COMMAND	upper-case is required literal word
Name	initial capitals is technical term
Optional	underscore is optional part
n	number

Optional parts (names, numbers, separators) change the meaning of the basic command. A command without an optional part is often an inquiry. The optional part provides a value or a name for more detailed specification.

SYSTEM COMMAND CATEGORIES.

The system commands are grouped according to categories:

session controls
terminal controls
clear workspace controls
library controls
name displays
erase names
group commands
run state

```
)ON
)COFF
)OFF
```

SESSION CONTROLS.

Session controls are used to initiate and terminate a work session.

)ON Accountname [Password]	signs on account
)COFF [Oldpassword/Newpassword]	signs off to continue
)OFF [Oldpassword/Newpassword]	signs off

)ON logs the account on the APL/700 system and initiates work. If any continuation workspace exists, it is reactivated at the point at which it was interrupted.

)COFF logs the account off, retaining the active workspace for reactivation at next)ON for that account.

)OFF logs the account off and discards the active workspace, so at next)ON for that account, the user will have a clear workspace.

Both)OFF and)COFF return date and time, then the amount of CPU (processor) time and elapsed time used. These amounts are given both for the session and cumulative for the installation accounting period. Units are hours, minutes, and seconds.

The Accountname is assigned by the installation. It is considered to be public knowledge.

The optional Password allows protection of a user's own account from unauthorized use. The Password can be initially set by the installation, or by the user at any sign-off. Once set and until removed, the proper Password must be used for any successful sign-on. Either Oldpassword or Newpassword may be empty. The forms for adjusting the password at sign-off are:

[/Newpassword]	establishes password
[Oldpassword/Newpassword]	changes password
[Oldpassword/]	removes password

An Accountname may have 1 to 6 characters; a Password 1 to 12. These characters are alphanumeric (excluding the APL underscore alphabet). A Password must begin with a letter.

```
)ON DOREEN
)COFF[/SESAME]
)ON DOREEN[SESAME]
)OFF [SESAME/NEWKEY3]
```

)BLOT   obscure an area

)BLOT provides multiple overprinting of a 17 character area, then backspaces to the prompt position to obscure subsequent display of a sensitive entry such as the Password on the account. It can be used before)ON or during a normal use session.

```
      )BLOT
             █████████████████
```

TERMINAL CONTROLS.

An account can be used from any terminal. The line width and tab setting are given default values. The suggested defaults are indicated in the initial WAS n display response in the examples. If these are unsatisfactory, alternatives may be specified and retained with the account (which is assumed to be normally used from the same terminal).

)WIDTH n   maximum characters in display line

The number of characters n is in the inclusive range 30 to 32767. If n is not specified, the result is the current width. The width setting affects the maximum characters that can be displayed on one line. Data objects requiring more characters are automatically folded onto several output lines.

```
      )WIDTH 65
WAS 120
      )WIDTH
IS 65
```

)TABS n   physical tab interval

The integer n is the number of characters between the physical tab settings. This single interval should match the tabs as actually set on the terminal. If n is not 0, then output with "white space" will automatically use tabs to minimize the time to reach a position on the display. Thus the tabs should be used if available on the terminal. The tab key can also be used for entry if tabs are set. The maximum value for n is 30.

```
      )TABS 5
WAS 0
      )TABS
IS 5
```

```
)CLEAR
)SYMS
)ORIGIN
```

CLEAR WORKSPACE CONTROLS.

Workspace controls provide the default SYMS, ORIGIN, DIGITS, SEED, and FUZZ for a clear workspace that is suited to the normal desires of the account user.

)CLEAR n   clears the workspace

The clear command without n destroys the prior active workspace and replaces it with a clear workspace having no names in it and the default attributes hereafter described. If n is specified, it refers to the number of symbols reserved for the symbol table. This number must be in the domain 16 through 1024.

```
      )CLEAR
CLEAR WS
      )CLEAR 300
WAS 256
```

The response indicates the number of symbols in the prior active workspace. It does not change the default number, which is controlled by)SYMS.

The following commands return current values or specify new default values for controls applicable only to an initially clear workspace. The examples illustrate typical installation-provided default values and samples of changes to them.

)SYMS n   default symbol table size

The default symbol table size for a clear workspace is set to n, in the domain 16 through 1024.

```
      )SYMS
IS 256
      )SYMS 400
WAS 256
```

)ORIGIN n   default ordinal index origin

Origin affects primitive functions that use ordinal numbering. The default index origin can be overridden by the index origin system variable ⎕IO.

```
      )ORIGIN
IS 1
      )ORIGIN 0
WAS 1
```

)DIGITS n   default print precision

The default maximum number of significant digits displayed in either fractional or exponential form is established in a clear workspace by the value of n. This must be an integer from 1 through 12 inclusive. This number has no effect on the internal precision of representation. The default digits can be overridden by the system variable ⎕*PP*, print precision.

```
      )DIGITS
IS 10
      )DIGITS 4
WAS 10
```

)SEED n   default random number seed

The pseudo-random number generator used in the roll and deal primitive functions is pre-set to the default value of Seed. This permits repeated execution of an algorithm to receive the same supplied random values if desired. The value of n is a non-negative integer: 0 through 549755813887 (the largest integer). The seed is the starting value for the random link. The random link changes with each use of roll or deal and can be changed by the system variable ⎕*RL*, random link.

```
      )SEED
IS 0
      )SEED 37752963
WAS 0
```

)FUZZ n   default comparison tolerance

The comparison tolerance by which two approximate representations of a number are considered equal is established in a clear workspace by)FUZZ n. The allowable range for n is 0 ≤ n < 1. The default fuzz may be overridden by the system variable ⎕*CT*, comparison tolerance. See that description for details.

```
      )FUZZ
IS 1E¯10
      )FUZZ 0.1
WAS 1E¯10
```

```
)FILES
)LIB
```

LIBRARY CONTROLS.

The library of an account includes named files and workspaces. Commands to interrogate the names and to totally or selectively access workspaces are provided. File access is done through primitive file system functions.

```
    )FILES                        display file names of account
```

The names of files owned by the account are displayed. Only the public part of the name is displayed; any password on a file is omitted.

```
         )FILES
    DATAFILE
    DOCUMENT
```

```
    )LIB                          display library names of account
```

The identifiers of workspaces in the account library (but not their passwords) are displayed.

```
         )LIB
    NEW
    TEXTEDIT
```

The form for referencing workspaces in the following)LOAD,)COPY, and)PCOPY commands is:

    Workspacename is (Account) Wsid [Password]

The Wsid is the identifier by which the workspace is known. It must start with a letter followed by 0 to 11 letters or digits.

The Account portion is the owning account name of the library in which the workspace resides. It may be elided if it is in the user's own account.

The Password is used only if the workspace is locked. The password is also a name starting with a letter and followed by 0 to 11 letters or digits.

)LOAD Workspacename load copy of workspace

The prior active workspace is eliminated. A copy of the specified workspace becomes the active workspace. The Wsid of the loaded workspace (not the Account or Password) becomes the name of the active workspace.

```
        )LOAD TEXTEDIT
        )LOAD JOANNE[KOLOHE]
        )LOAD (LIB) NEWS
```

)COPY Workspacename Namelist replace copy

Copy into the present active workspace from the library workspace identified by Workspacename. If Namelist is present, copy only the items attached to names in it that are present in that workspace. If Namelist is absent, copy all functions, variables and groups in the workspace. A copied item will replace a prior item of the same name in the active workspace.

```
        ) COPY TEXTEDIT
        ) COPY JOANNE[KOLOHE] FORECAST SCHEDULE
```

)PCOPY workspacename Namelist protect copy

Same as)COPY except that any name in Namelist already existing in the active workspace will not be copied.

```
        )PCOPY (LIB)NEWS SCHEDULE INDEX
    NOT SCHEDULE
```

)COPY and)PCOPY are more complex commands using more resources than)LOAD, so should only be used when)LOAD is inadequate.

)SAVE Wsid [Oldpassword/Newpassword] save workspace

A copy of the active workspace can be saved in the account library of the user. If Wsid is present, that name is the one used for subsequent library reference; if absent, the prior active workspace identifier is used. This will replace a former like-named workspace. The forms for establishing, changing or removing the workspace password are the same as for the account. See)OFF and)COFF for details. If the Newpassword is present, subsequent)LOAD or)COPY of that library workspace must supply the password.

```
        )SAVE WORK
        )SAVE NEW [/VERSION]
```

```
)DROP
)WSID
```

)DROP Wsid [Password]   drop account library workspace

A workspace in the account library can be destroyed by using)DROP. The password is required if the workspace is locked. A workspace in one account library cannot be dropped from any other account.)DROP does not destroy the active workspace, even if it has the same name as the command)WSID returns.

```
          )DROP NEW[VERSION]
```

The normal response from the)LOAD,)COPY,)PCOPY,)SAVE and)DROP commands is typically:

```
     SAVED 74/10/01 8.00.01
```

A suffix identifying the account and workspace name is appended for)SAVE.

)WSID Name   workspace name

The workspace name provides a reference for the workspace when saved in the account library. The clear workspace is unnamed.

```
          )WSID
     IS UNNAMED WS
          )WSID NEW
     WAS UNNAMED WS
          )WSID
     IS NEW
```

NAME DISPLAYS.

The following system commands display classes of primary names currently in the symbol table:

)FNS Name	display primary function names
)VARS Name	display primary variable names
)GRPS Name	display group names

The primary names are those existing in a workspace when the state indicator is empty. Thus no local names are displayed for these commands. If Name is absent, the entire class is displayed in alphabetical order. If Name is present, only the members of the class starting with (or after) Name are displayed. The display result can not be used as an APL data object. The system function ⎕NL, name list, should be used for that purpose.

```
      )FNS
FINDER  FORMAT
      )VARS
W  Z  CONVERT
      )GRPS
DISPLAY
```

ERASE NAMES.

)ERASE Nameset	erase set of names

Names of functions, variables and primary names of groups named in Nameset are erased from the workspace. The names in Nameset are entered, separated by spaces. Function names can not be erased while in the state indicator. Notice is given for non-existent or non-erasable members of its nameset. See discussion in Group commands following:

```
      )ERASE W X Y Z FINDER
NOT X
NOT Y
```

)ATTACH
)DETACH

GROUP COMMANDS.

A group of names can be formed and named for collective reference including)ERASE or)COPY.

```
     )ATTACH   Groupname  Nameset               group association
```

The Groupname is the identifier for the group. The Nameset provides the names that are associated with the group, and thereby, with each other. Normally, names in a Nameset match names of variables, functions or other groups. Names in the Nameset need not have any current meaning.

If Nameset is not present, the effect is to reserve Groupname, as a group, for subsequent attachment of a nameset. If the group Groupname already exists, the effect is to unite Nameset with the nameset already associated with Groupname (no name will be duplicated).

A group name included in Nameset causes the elements of that group's nameset to be implicitly included in the group.

If the Groupname is included in its own Nameset, then actions on the group apply also to the Groupname.

```
          )ATTACH GROUP1 FNAME VNAME GROUP1
          )ATTACH GROUP2 GROUP1 GROUP2 HOW
```

```
     )DETACH   Groupname  Nameset               group disassociation
```

The names in Nameset are detached from the group Groupname. If Nameset is absent, then the group Groupname ceases to exist.

Detach doesn't affect the existence of the names (other than Groupname). This is contrasted with)ERASE which eliminates the named objects.

```
          )DETACH GROUP2
```

)GRP Groupname   display group association

The names directly attached to Groupname are displayed in the order they were attached.

A group can contain in its Nameset its own name. If so, an action on the group nameset affects the group as well. A group (say G) can contain names of other groups. If so, an action on group G will replace each named group in its Nameset by that group's nameset. Any one group will only be replaced once. A second occurrence of a group name signifies the name itself rather than a replacement. Thus the primary definitions of names in a Nameset are the unique names remaining after applying the following for each name:

substituting for first occurrence of any Groupname its Nameset

retaining the Groupname on its second occurrence

ignoring any additional occurrences, giving warning: NOT Name

An example of this process illustrates these steps:

```
      )CLEAR
CLEAR WS
      )ATTACH A B
      )ATTACH B C B
      )ATTACH C D A A
NOT A
      ⍝ THIS WAS FOR THE SECOND OCCURRENCE
      )GRPS
A  B  C
      )ERASE A
NOT D
      ⍝ D HAD NEVER BEEN GIVEN MEANING
      )GRPS
C
      )GRP C
D  A
```

The illustrations at the right show the nameset tree for group A after substitution of group namesets; and the resulting primary definitions. Note that the primary definition includes groups A and B (but not C) and undefined name D. This was done while the 3 groups existed.

```
   A              A
   |             /|\
   B            D A B
   |\
   C B
   |\
   D A

 Nameset        Primary
  Tree        Definitions
```

```
)SI
)RESET
```

RUN STATE.

The run state is the record of user defined functions in process, suspended, or pending completion of other called functions.

)SI   state indicator query

The result is the stack indicating the run state of suspended and pending functions. The first line (if non-empty), is the most recently suspended function. Below are pendant functions (awaiting completion of functions above) and earlier suspended function.

Each line gives function name, bracketed line number at which execution is pendant or suspended, and an asterisk for suspended functions only.

```
      )SI
RUN[1]*
MAIN[5]
RUN[4]*
```

A function can appear more than once in the state indicator. In line 5, MAIN called RUN. MAIN is pending completion of RUN. More than one suspended function can appear. A function can reappear (independent restarts, or recursive calls are permitted).

Usually the state indicator should be emptied of unnecessary entries, as space is consumed and global names may be shielded by local names. The state indicator may also include suspensions with evaluate functions or evaluated input requests are incompleted. In each such case, the appropriate symbol ⍎ or ⎕ appears prior to the function line causing suspension.

The suspended function at the top of the state indicator may be restarted by entering →*N* where *N* is a line number. The suspended function and any pending on it may be aborted by entering →. Response is a line showing the next suspended function if any.

```
      →
RUN[4]*
      )SI
RUN[4]*
```

)RESET   state indicator reset

The entire run state can be cleared using)RESET. The resulting state indicator is reset:

```
      )RESET
      )SI
```

SECTION 4

THE APL/700 LANGUAGE

GENERAL.

The APL/700 language contains many powerful primitive functions that apply to data objects.

A data object may be:

an element of either character or numeric type,
an array structure formed of these elements,
named, forming a variable by assignment, not declaration.

Each primitive function:

is represented by a single character,
applies to one or two arguments that are data objects,
returns a data object result.

An APL expression is the syntactically correct composition of one or more APL language constituents.

data objects
primitive functions and operators
calls on defined functions
file functions
system variables
shared variables
system functions
input-output communicators
control structures

The results of executing an expression include change to the state of processing, or display to the user, or both.

This section describes data objects, names, expression composition and order of elaboration, input-output communicators, and the convention for comments. The other constituents are subsequently described.

DATA OBJECTS.

A data object is defined in terms of its type, rank, shape, and value.

The type is either numeric or character (any of the APL characters literally representing themselves).

The rank is the number of dimensions. Allowable ranks are 0 through 16. An array is a data object with positive rank. Rank can be viewed in geometric terms: a scalar (rank 0) as a point, a vector (rank 1) as a line segment, a matrix (rank 2) as a rectangle, a rank 3 object as a rectangular solid, etc.

The shape is the vector of dimension lengths, from first to last.

The value of each element of a data structure must be within the allowable domain for that type.

In general, an array is characterized as follows:

homogeneous (single type for all elements)

N-dimensional Cartesian (rank N, independent dimensions)

rectangular (all planes across a dimension have the same shape)

dense (all elements have values, as contrasted with sparse in which some means is provided to indicate the locations of elements having significant values)

A plane is a slice of an array that is orthogonal (at "right angles") to a given dimension of that object. A plane across the K-th dimension of an N dimensional object has N-1 dimensions. It retains all but the K-th dimension. Thus, a plane across a vector is a scalar. A plane across a matrix is a vector (either from a row or a column, depending on K).

A vector along dimension K is parallel to the axis for dimension K. The axis for dimension K is the vector along K formed by holding all the other dimensions at their first (origin) values.

A corner element of an array has for each dimension either the origin or anti-origin (or last value for that dimension) as index value. An N-dimensional array thus has 2*N corner elements.

A corner of an array is another array of the same rank containing at least one corner element that is also a corner element of the original array.

The size of a data object is the number of elements it contains, independent of shape.

Table 4-1

Examples of Data Object Forms

Numeric Type			Data Structure	Character Type		
Value	Rank	Shape		Value	Rank	Shape
¯100.341	0	(empty)	SCALAR	*A*	0	(empty)
¯2.5 0 3	1	3	VECTOR	*ABCDEF*	1	6
11 12 13 21 22 23	2	2 3	MATRIX	*ABCD* *EFGH*	2	2 4
111 112 121 122 131 132 211 212 221 222 231 232 311 312 321 322 331 332	3	3 3 2	ARRAY	*ABCD* *EFGH* *IJKL* *MNOP* *QRST* *UVWX*	4	2 3 1 4

Table 4-1 shows examples of data objects. For both numeric and character type, various values are shown as if displayed, and their rank and shape are indicated. The default display of numeric vectors has 2 spaces between successive elements in a row. The column spacing for numeric objects with rank 2 or more is uniform based on the largest space required between elements in a row. The display of rank 3 arrays has one blank line separating planes across the first dimension; display of rank 4 arrays has two blank lines separating (3 dimensional) planes across the first dimension, etc.

Character data can include any of the 256 allowable APL characters of the atomic vector as literal elements. Only displayable and designated special characters (see Section 6) should be entered or used for output to the display. Entry of a character string is enclosed in quotes. An embedded quote pair is entered if the quote literal is required. Thus, entry of '*DON''T*' results in the data object *DON'T*.

The display of negative numeric data uses the "¯" character (read as negative) to the upper left of the number. This character is distinct from the subtract character "-" (read as minus or negate) in primitive functions. For example:

¯23	negative	17-5	minus

Fixed point number entry for decimal fractions need not be preceded by 0; display (or constant representation in a defined function) does have the leading 0.

```
      .3125  ¯.2
0.3125  ¯0.2
```

If fixed point representation is excessively long, or if numbers have very large or small magnitude, an exponential or "scientific" representation is provided. Default output in this representation takes the form of a signed number with magnitudes between one and ten times a power of 10. Fractional parts are only displayed if necessary. Input using this notation can be any real integer or fixed point number with an exponent.

	equivalent	canonic form
¯387*E*3	¯387000	¯3.87*E*5
12*E*¯4	0.0012	1.2*E*¯3
200	2.0*E*2	2*E*2

The domains for numeric type data elements are:

Sub type	Domain
Boolean	0 *AND* 1
Integer magnitude	0 *THRU* 549755813887 ←→ ¯1+8⋆13
Real magnitude	0 *AND* 8.75811540203*E*¯47 ←→ 8⋆¯51
(normalized)	*THRU* 4.31359146674*E*68 ←→(¯1+8⋆13)×8⋆63

Integers are the subset of reals having 0 exponent. (Almost 12 digits are available for precision for either).

Booleans are a subset of integers (and reals).

Some data objects have special properties that are not always evident from their display.

A scalar is a data object with one element but having empty shape (rank 0).

A shaped data object is one with positive rank.

A single (element data object) of any rank has size one and is displayed on a single line. Any dimension must have a length of 1.

An empty data object has no elements. It does have type, resulting from the way it was generated. Its rank must be greater than 0 as the length of at least one dimension must be zero.

A string is a character type data object that is either a scalar or a vector. If the content appears as a valid numeric value, there may be no distinction in the display.

Table 4-2

Tests for Properties of Data Objects

Property	Holds if result is true (1)
Scalar	0=ρρD
Vector	1=ρρD
Matrix	2=ρρD
Shaped object	0<ρρD
Single	1=×/ρD
Empty	0=×/ρD
Numeric type	0=0\0ρD
Character type	' '=0\0ρD
String	(' _ '=0\0ρD)∧2>ρρD
Integer	((¯1+2*39)∧.≥|,D)∧(,D)∧.=⌈,D
Boolean	0 1⊃D

Table 4-2 provides tests for these properties of a data object (D) in terms of primitive functions that will subsequently be defined.

Data objects are used in expressions and are the results of elaboration. A data object may receive its value by several means:

inclusion as a constant in an expression

entry in response to an input prompt

direct result of function elaboration

reference to a variable name

reference to a file component

default (for initial values of system variables)

acceptance of a variable shared with another process

A constant is either a number (or vector of numbers) or a literal (a quoted string) entered as part of an expression. The linear entry mode restricts constants to rank 0 or 1.

A data object that is a direct result of function elaboration and that is only used as an argument to another function (other than replacement) disappears after that second function has been elaborated.

NAMES.

Names are used as identifiers of items that may change during the life of the workspace or account.

A name begins with a letter, underscored letter, ∆ or ∆̲ . The rest (if any) of the name consists of additional characters chosen from these characters, the digits and the underscore.

Most names may be of any practical length: 1 to 69 characters. Names of restricted length are used as identifiers of workspaces (12 characters), files (12 characters), accounts (6 characters), or passwords (12 characters). As well, these restricted names may include only letters and digits (an account name may begin with a digit as well). Names are used in the following ways.

Variables: A name can be associated with a data object through assignment. Thereafter, until some other meaning is given to that name, it is called a variable. Subsequent references to that name yield that data object until some other assignment of that same name, or the name becomes undefined (see user-defined functions, ERASE system command or EXPUNGE system function). There is no need to explicitly declare a name or its type or shape as these attributes are part of the data object being assigned.

User defined functions: A function name provides a reference to the processing algorithm thereby described.

Labels: Local to the user defined function in which it appears, a label is a named constant having value the number of the line on which it occurs.

File names: Each file created or referenced by a user is identified by its account name (if owned by another account), its file name, and its password (if locked).

Shared variable names: Variables shared with some other process have external names or surrogates known to both processes.

Group names: For purposes of copying and erasing, a group of names may be named. One of the group members may be the group name itself. A group member may be a function name, variable name, shared variable name, or group name. A group may also serve as documentation.

Workspace names: A workspace may be named and saved. Thereafter it can be loaded or copied by name, or names within it may be copied.

Account names: Each user account has a name supplied by the system and used for sign-on and accounting purposes.

Passwords: Each user account, workspace name or file name may have appended a password established by the user and used to control access.

EXPRESSIONS, LISTS AND ORDER OF EXECUTION.

An expression is formed from APL language constituents. Proper formation of an expression requires understanding of the order of elaboration of its constituents. *Elaboration* is the process of determining the value of an expression. Three general rules apply:

> A function is elaborated only when the values of its *arguments* (the quantities it requires for its elaboration) are known.
>
> The order of elaborating functions in an expression is from *right* to *left*.
>
> Parentheses are used in the conventional mathematical way to alter the order of execution.

Thus, a *monadic* function is elaborated when the value of its (right) argument is determined. A *dyadic* function is elaborated when both of its arguments (left and right) are determined. An argument can itself be an expression. A *niladic* function is elaborated when its result is required in the expression in which it is the rightmost constituent.

The order of argument elaboration for a dyadic function is undefined, and is generally unimportant (both arguments could be elaborated in parallel if independent). The order is usually right-to-left. An exception to this is where the right argument is a variable name. If elaboration of the left argument changes the meaning of that right argument name, the right argument is changed to conform.

EXPRESSION FORMATS.

In the following samples of expression formats, "V" represents a data object value being used as an argument, "m" represents a monadic value-returning function, and "d" represents a dyadic value-returning function. Each elaboration of a function replaces the function and its argument(s) with a value. Each elaboration of an expression within parentheses replaces it with a value. Note that there is no ambiguity in determining whether a function is monadic or dyadic; a function is dyadic if it has an argument to its left; otherwise, it is monadic.

```
V d V d V                     expression
V d(V d V)                    equivalent expression
  2   1                       order of elaboration

V d m V                       expression
V d(m V)                      equivalent expression
  2 1                         order of elaboration

(V d V)d V                    expression
   1   2                      order of elaboration

m(m(V d V)d m V)d V d m V     expression
8 6   4   5 3   7   2 1       order of elaboration
```

It is not necessary to enclose right arguments within parentheses. Redundant parentheses will be ignored. In defined functions, redundant parentheses are eliminated once the expression containing them has been elaborated.

The following examples include both the entered expression (shown indented) and the result of its elaboration (on the next line). This is the typical appearance of the examples entered and displayed on a terminal. The equivalent columns could also have been entered (they would actually also be indented for entry, no indenting for result display).

```
     Expression              Equivalent              Equivalent

          3×5+2                   3×(5+2)                 3×7
     21                      21                      21

          1-2-3                   1-(2-3)                 1-¯1
     2                       2                       2

          (1-2)-3                 ¯1-3
     ¯4                      ¯4

          5×-2                    5×(-2)                  5×¯2
     ¯10                     ¯10                     ¯10
```

EXPRESSION LISTS.

A *list* is either an expression, or has components separated by delimiters. A *delimiter* is either a semicolon, or one of matching parentheses or brackets. Each *component* is either an expression or null (two adjacent delimiters). Components are elaborated right-to-left. The value of a component that is only a variable name will be affected by any change in its meaning from subsequent component elaboration in the list. If the list is used for display purpose, the display order is *left* to *right* after all the components have been elaborated. No type requirements exist between successive components.

```
          m V; V d V; m V; V        expression list
          6 5    4  3 2  1          order of elaboration
```

BRACKETS.

Bracketing is used to bound an expression list used for subarray selection from an array, or for qualification to identify the dimension about which a function is to be applied. A bracketed expression or expression list is elaborated before the related expression that is its left argument. Matching brackets are treated as a single function.

```
          V [ m V; m V ] d V        index expression list
            4 3  2 1     5          order of elaboration

          V d [ m V ]V              dimension selector
            3 2 1                   order of elaboration
```

EXPRESSION ENTRY.

Expression constituents are entered in free form: the order of character entry is immaterial. The visual fidelity as displayed (and as in-line corrected) is what is accepted as the entry.

One blank must appear as a separator between two names or numbers. Extra blanks are ignored. The only context in which an exact number of blanks is preserved as significant is in character strings or comments. Extra matching pairs of parentheses in an entered expression may help to clarify it and do no harm. In defined function representations once elaborated, both extraneous parentheses and blanks are removed from subsequent display of the defined function.

The last entered expression is available for further editing. This is normally the last expression elaborated in execution mode. This can be used for progressive expression development. Entering a correct system command or entering function definition and editing mode has no effect on the last entered expression (unless an immediate edit is done to replace it by a line of a function as described in Section 8). It is also possible to capture the last entered expression in a function by editing it to include opening of the function and specifying the line in which the expression is to be placed.

```
      ⎕←X←3 4 5                display after assignment to X
  3  4  5
      +/X                      sum over X
  12
      +/X                      ATTN redisplays
      .                        edit mark
      (+/X)÷⍴X                 add '(', ATTN for +/X, then rest
  4
      ∇AVE X∇                  create defined function header, close
      (+/X)÷⍴X                 ATTN recovers last entered expression
      .                        edit mark
      ∇AVE[1] (+/X)÷⍴X∇        reopen function for insert in line 1
      AVE 1 2 3                execute AVE with new argument
  2
      AVE 1 2 3                ATTN recovers
                               RETN cancels
```

The entry of an expression must be syntactically valid in its composition, or an appropriate error message is given. This is true in either execution mode or function definition and editing mode. An errored entry is available for recall using ATTN. It can be then repaired by editing. See Section 9 for error reports.

A syntactically correct expression may still contain errors sensed during elaboration, such as an undefined, improperly shaped or typed variable. Again after the error message, the errored entry is available for correction.

It is permissible to use as part of an expression up to five characters typed in the indent space of the execution mode prompt.

COMMENT
⍝

Forms:

⍝ C   comment text C

E ⍝ C   comment text C after expression E

Where: C is any string of valid APL characters.
E is any APL expression, label or branch

Results:

A comment is uninterpreted text. It has no effect on execution of E to its left.

Conditions:

In a defined function each comment does take space for storage.

Locating a comment in a defined function on an unexecuted line is slightly advantageous (if no extra control transfer must be introduced to achieve this).

Examples:

```
      3×6-2 ⍝ RIGHT TO LEFT FUNCTION EXECUTION
12
      ⍝ A COMMENT BY ITSELF
```

INPUT/OUTPUT COMMUNICATORS (1) ⎕ ⍞

Forms:

⎕	Evaluated input
⎕← E	Explicit output
⍞	Character input
⍞← E	Set character input prompt
E	Implicit output
E1;E2...;En	Mixed type output

Where: E, E1, E2, En are APL expressions

Results:

The terminal keyboard is the input source; the display is the output destination.

Evaluated Input: The prompt ⎕: is displayed, followed by an indent on the next line and keyboard unlock. Input from the user of any value-producing expression is then accepted for evaluation as if in execution mode. Evaluated input occurs when ⎕ appears in an expression where a value is required. The resulting value replaces the ⎕ in that expression evaluation.

Character input: The character input prompt is displayed and the keyboard is unlocked. A character string including that prompt as prefix is accepted as input.

Explicit output: Assignment to the pseudo-variable ⎕ causes display of the value. Each such assignment causes display of the appropriate value. Several such assignments in one line result in display in the order that the values are determined.

Set character input prompt: Assignment of a character string to the variable ⍞ establishes the character input prompt which is thereafter shared with the APL processor. That prompt subsequently will be displayed prior to character input. The ⍞ can be a local variable. The default for ⍞ prompt is the empty character vector ''. Once set, a prompt is retained until changed (or cancelled by exit from the function to which it is local).

Implicit output: The value resulting from expression evaluation is displayed if it is not assigned to a variable name (the last function executed was not an assignment primitive), or the last primitive executed was not done primarily for side effect (e.g., create a function, expunge a name, offer to share variable). This is the common result of expression evaluation in execution mode. It is equivalent to placing ⎕← at the left of the expression.

Mixed type output: this is a redundant means for producing output with mixed type. This form is a list of expressions of possibly different types separated by semicolons. The expressions are evaluated right to left (En then En-1,...E1), then the results are displayed left to right and without extra space between for each scalar or vector result. Each array result of rank at least 2 starts on a new line, as does any following sequence of scalar or vector objects. Formatted conversion of numeric output with ⍕ is preferable.

Conditions:

Output to the display is also constrained by the print width established for the terminal. Automatic folding of output that is too long for the available print width occurs. For numeric vector output, folded lines are indented and a fixed number of blanks are inserted between each element. Numeric array output is put in fixed width columns.

Failure to enter a value producing expression for evaluated input results in another □: prompt. Escape from this can be achieved by terminate entry: '→'.

Escape from character input equivalent to the terminate entry above can not be by '→' as that is an acceptable character. Instead, escape is by entering the double overstrike (the only one allowed), and only in this context:

Ø (*O*, backspace, *U*, backspace, *T*)

Note that combinations are meaningful:

⍞ ← ⍞	Request character input to establish new character input prompt
□ ← ⍞	Display prompt, accept input and echo it back including prompt
□ ← □	Accept and evalute input and display value
⍞ ← □	Accept and evaluate input and use character result to set prompt

Examples:

```
      ⎕+5          ⍝ REQUEST EVALUATED INPUT
⎕:
      3 4 5        ⍝ INPUT IN RESPONSE, IMPLICIT OUTPUT
8  9  10
      ⎕←X←1+3      ⍝ EXPLICIT OUTPUT
4
      2+⎕←X×2      ⍝ EXPLICIT WITHIN EXPRESSION, THEN IMPLICIT
8
10
      ⍞←'?'        ⍝ SET CHARACTER INPUT PROMPT
      X←⍞          ⍝ DISPLAY PROMPT FOR CHARACTER INPUT
?ENTRY
      X            ⍝ INCLUDES PROMPT AND 'ENTRY'
?ENTRY
      1↓⍞          ⍝ DROPS PROMPT, KEEPS REST INCLUDING COMMENT
?NEW           ⍝ SOME TEXT ENTRY
NEW           ⍝ SOME TEXT ENTRY
      'RANK=';⍴⍴X;' SHAPE=';⍴X;' VALUE=';X←2 3⍴⍳6
RANK=2 SHAPE=2 3 VALUE=
1  2  3
4  5  6
      ⍝ NOTE ARRAY STARTS ON NEW LINE OF MIXED OUTPUT
```

SECTION 5

PRIMITIVE FUNCTIONS AND OPERATORS

GENERAL.

APL/700 provides a set of standard functions referred to as primitive functions because they are immediately available as part of the APL language to the user for application. These primitive functions are discussed under the following categories:

Selection function
Assignment functions
Scalar functions
Compound operators
Mixed functions
Format functions

The primitive functions and operators are represented by single APL characters. The same character is often used to represent both a monadic (having only right argument) and a related dyadic (having both right and left arguments) function. The descriptions of such related uses are located together.

The following notation conventions are used to describe the APL primitive functions and operators. They are not part of APL.

⊙	any monadic scalar primitive function
⊕	any dyadic scalar primitive function
⊗	any dyadic scalar primitive function
X ↔ Y	formal equivalence of expressions X and Y

Formally equivalent expressions may not yield computationally identical results. Numeric precision restrictions in computation may cause differences in the allowable extreme domains that can be accepted by the formally equivalent expressions. As in any computations using finite precision numeric representations, algorithm differences may cause small differences in the results obtained. The implementation of the APL primitive functions has been done using algorithms that in general provide stable computation with accuracy of about 12 decimal digits.

Examples of function application are given to illustrate their use, often with shaped data objects as arguments. This is done to provide a variety of significant results in a minimum of space. Numeric precision for display of fractional numbers is typically 5 digits. The results are rounded. Up to 12 digits of precision can be displayed per number if desired.

SELECTION FUNCTION (1)
[ ]

Form:

A[L]   Select elements of A indicated by L

Where: L is a index list of the form E1;...;Ei;...;Ek

A is an array name (or parenthesized value producing expression) having positive rank K.

Result:

Selection accesses a rectangular subarray of A. The index list (also called subscript list) L identifies the members of each dimension of A being selected. The typical subscript list component Ei refers to indices along dimension *I* of A. Ei may be omitted (null) meaning the ordered vector of all indices (the domain) for dimension *I* ↔ ι(ρ*V*)[*I*]. Otherwise Ei may be any integer value-producing expression of any rank with all values in that domain.

The result shape is the catenation of the shapes of the Ei. The result rank is the sum of the ranks of the Ei. If all Ei are scalars, so is the result.

Each element of the result has the same value as a single element of A selected with one dimension value from each dimension of A. Each element from any Ei is used with all members from each of the other dimensions. This is similar to the outer product applied between each of the Ei to develop the product set of possible indices.

Selection may appear to the left of the assignment arrow, in which case only the selected elements are inserted or modified. Either the data object to the right of the assignment is a single or it has the same shape as that of the selection.

Conditions:

If the same elements are selected more than once for insertion, the results are ill-defined.

Selection is origin sensitive.

Selection is a general function with attendent complexity. Simpler functions should be used for regular, contiguous subarray access. Selection should be reserved for accesses to irregular subarrays of shaped data objects.

Examples:

```
      A
11  12  13  14
21  22  23  24
31  32  33  34
      A[2;2]
22
      ⍴A[2;2]      ⍝ SCALAR

      A[,2;,2]
22
      ⍴A[,2;,2]    ⍝ ARRAY
1  1
      A[3;]        ⍝ ALL ROW 3
31  32  33  34
      A[;2]        ⍝ ALL COLUMN 2
12  22  32
```

```
      A[1 3;2 4]
12  14
32  34
      A[2 2 1;1 3 1]
21  23  21
21  23  21
11  13  11
      V
ABCDE
      V[1 3 5]
ACE
      V[3 5 4 5]
CEDE
      V[2 3⍴2 1 4 3 1 2]
BAD
CAB
```

ASSIGNMENT FUNCTIONS (1)

←

Forms:

N ← E — Replace the data object identified by N (if any) with the object resulting from E

A[L] ← E — Insert the value of E into locations from index list L of the previously existing array A

M ⊕← E — Modify M, short for *M←M⊕E*

A[L] ⊕← E — Modified Insert, short for *A[L]←A[L]⊕E*

Where:

M is name for which current meaning is not a label, function, or group (M is a variable name)

N is M, ⎕, ⍞, shared variable, system variable, or has no current meaning

E is result of evaluating an expression

A is name of a variable with shape, i.e., an array

L is index list valid for A

⊕ is any scalar dyadic primitive function

⌊ ⌈ + - × ÷ | ⋆ ⍟ < ≤ = ≥ > ≠ ∧ ∨ ⍲ ⍱ ○ !

Results:

Assignment functions give value to or alter the value of the left argument.

Results are only explicitly returned if required for further expression elaboration. If the assignment function is the last to be elaborated on a line, no explicit result is returned for display unless the leftmost argument is ⎕.

Replace: The value returned is E. This value is displayed if N is ⎕. If N is M the returned value is ignored unless required as an argument to a function.

Insert: The value returned if required is the same as the value inserted: E.

Modify: The result is the value assigned to M: *M⊕E*.

Modified Insert: The result is the value inserted: *A[L]⊕E*.

Conditions:

Replace: The value and all attributes of E are given to N, destroying any prior associated meaning for the name N. If N is M and no prior occurrence of N existed, N is added to the symbol table.

Insert: The shape of E must conform to the shape of the array selected by L, and the types of A and E must be the same.

Modify: The shape of E must conform to the shape of M. The types must be the same.

Modified Insert: Saves computer time if determination of L involves expression evaluation. The shape of E must conform to the shape of the array selected by L, and the types of A and E must be the same.

For Insert or Modified Insert, if any element from L is repeated, the result is ill-defined.

Examples:

```
      X←'APL'          ⍝ REPLACE X BY CHARACTER VECTOR 'APL'
      X
APL
      ⎕←X←¯1 0 1       ⍝ REPLACE OLD VALUE WITH NEW AND DISPLAY
¯1  0  1
      Z←Y←X            ⍝ MULTIPLE REPLACEMENTS
      Y
¯1  0  1
      Z
¯1  0  1
      A                ⍝ EXISTING ARRAY NAMED A WITH SHAPE 2 3
1  2  3
4  5  6
      ⎕←A[2;]←X        ⍝ INSERT X INTO ROW 2 AND DISPLAY
¯1  0  1
      ⎕←A[;3]←4        ⍝ COERCE AND INSERT TO COLUMN 3 AND DISPLAY
4
      A
 1  2  4
¯1  0  4
      ⎕←A×←2           ⍝ MODIFY ALL ELEMENTS OF A AND DISPLAY
 2  4  8
¯2  0  8
      ⎕←A[;1 3]÷←2     ⍝ MODIFIED INSERT COLUMNS 1 AND 3 AND DISPLAY
 1  4
¯1  4
      A
 1  4  4
¯1  0  4
```

SCALAR PRIMITIVE FUNCTIONS.

The scalar primitive functions include both monadic and related dyadic functions that apply element by element to the values of their arguments.

The scalar attribute indicates that scalar arguments return scalar results. An array argument to a monadic function returns a result of the same shape. Array arguments to dyadic functions of the same shape return results of that shape. Coercions are defined for single element arguments of any rank, and for one argument having shape that is a plane across the other argument when the function is qualified to apply to that dimension.

The scalar primitive functions include:

integer part and extreme value functions
arithmetic functions
power and logarithm functions
relational functions
logical functions
circular functions
combinatorial and factorial functions

Scalar primitive functions are used individually. The dyadic scalar primitive functions are also used as the function arguments to the primitive operators and to assignments including modify.

INTEGER PART, EXTREME VALUE FUNCTIONS ⌊ ⌈

Forms:

⌊ B	Floor of B
⌈ B	Ceiling of B
A ⌊ B	Minimum of A or B
A ⌈ B	Maximum of A or B

Where: A and B are numeric

Results:

Floor: Return the greatest integer not greater than B.

Ceiling: Return the least integer not less than B.

Minimum: Return the lesser (more negative) value of A or B.

Maximum: Return the greater (more positive) value of A or B.

Examples:

```
     ⌊¯3 ¯1.3 0 1.3 3
¯3  ¯2   0  1  3
     ⌈¯3 ¯1.3 0 1.3 3
¯3  ¯1   0  2  3
     2.1 3 ¯3⌊4.3 3 ¯6
2.1  3  ¯6
     2.1 3 ¯3⌈4.3 3 ¯6
4.3  3  ¯3
```

ARITHMETIC FUNCTIONS (1)

+ - × ÷ |

Forms:

+ B	Identity
- B	Negate
× B	Signum
÷ B	Reciprocate
| B	Magnitude
A + B	Add A to B
A - B	Subtract B from A
A × B	Multiply A by B
A ÷ B	Divide A by B
A | B	A residue of B

Where: A and B are numeric

Results:

Identity: Return the argument value. *+B* ↔ 0+*B*

Negate: Return the negative of B (unless B is 0, in which case the sign remains non-negative). -*B* ↔ 0-*B*

Signum: Return the integers ¯1, 0, 1 if B is negative, zero or positive. ×*B* ↔ (*B*>0)-*B*<0

Reciprocate: Return the reciprocal of B for non-zero B. ÷*B* ↔ 1÷*B*

Magnitude: Return the absolute value of B (a non-negative number). |*B* ↔ *B*××*B*

The expected arithmetic results occur for add, subtract, multiply and divide when B is non-zero. Divide, if both A and B are 0, returns a 1, (the limiting value of X÷X as X approaches 0). Otherwise, division by 0 is a domain error.

Residue: Return a remainder on division by non-zero A having sign of A and magnitude less than A. If A is 0, the result is B. If A<0 (>0), the result *R* is the least non-positive (non-negative) remainder for some integer *G* such that *B* ↔ *R*+*G*×*A*.

Conditions:

Note the argument order for divide and residue appear to conflict. For residue the divisor is A, whereas for divide, the divisor is B.

The identity may be used for a numeric variable to avoid the side-effect of subsequent assignment to the same name in the same expression respecifying the new value in place of the old. See Expressions, Lists, and Order of Execution in Section 3.

Examples:

```
      +7.2
7.2
      +0 5 ¯10 15
0  5  ¯10  15
      -1.2E3
¯1200
      -0 -5 ¯10 15
5  ¯10  15
      ×¯5 0 5
¯1  0  1
      ÷2 ¯5 10
0.5  ¯0.2  0.1
      |5 0 ¯5
5  0  5
```

```
      3.42E¯6+2.537E¯5
0.00002879
      1+0 5 ¯10 15
1  6  ¯9  16
      175-225
¯50
      5-0 5 ¯10 15
5  0  15  ¯10
      3 1 ¯4×¯5 2 ¯3
¯15  2  12
      5 ¯12 ¯15÷4
1.25  ¯3  ¯3.75
      3 3 ¯3 ¯3|4 ¯4 4 ¯4
1  2  ¯2  ¯1
```

POWER, LOGARITHM FUNCTIONS (1)

⋆ ●

Forms:

⋆ B	Base e to the power B
● B	Base e logarithm of B
A ⋆ B	Base A to the power B
A ● B	Base A logarithm of B

Where: A and B are numeric (see domain restrictions).

Results:

The results are numeric. The monadic forms are equivalent to the dyadic forms with A being e, the base of the natural logarithms:

2.7182818284... ↔ e

Conditions:

Power: Domain restrictions depend on the sign of A.

If A>0 then B can have any value.

If A=0 then B must be non-negative.

If A<0 then B must be either an integer or an expression whose value is N÷D where N is an integer and D is an odd integer. The comparison tolerance effects this determination whether N and D could be in the proper domains. (These cases yield a negative real root or an even power thereof).

Logarithm: The domain restrictions are:

A and B must be greater than zero.

A can only be 1 if B is 1.

Examples:

```
      *1   ⍝ BASE E
2.71828
      *¯1 0 3
0.367879  1  20.0855
      2*¯2 ¯1 0 1 10 13
0.25  0.5  1  2  1024  8192
      ¯2 ¯1 0 1 2*2
4  1  0  1  4
      1 2 3 4*0.5 ⍝ SQUARE ROOT
1  1.41421  1.73205  2
      16*÷1 2 3 4
16  4  2.51984  2
      ¯8 ¯27 ¯32*÷3 3 5
¯2  ¯3  ¯2
```

```
      ⍟*2 7 ¯3
2  7  ¯3
      *⍟2 7 1
2  7  1
      ⍟20 8192
2.99573  9.01091
      2⍟0.5 1 2 4 8
¯1  0  1  2  3
      1 2 3 4⍟1 4 27 2
1  2  3  0.5
      3⍟3*5
5
      3*3⍟6
6
```

RELATIONAL FUNCTIONS

< ≤ = ≥ > ≠

Forms:

A < B	Is A less than B
A ≤ B	Is A not greater than (less than or equal to) B
C = D	Is C equal to D
A ≥ B	Is A not less than (greater than or equal to) B
A > B	Is A greater than B
C ≠ D	Is C unequal to D

Where: A and B are numeric
C and D are either numeric or character type

Results:

Each Boolean result is 1 if the relation is true, 0 if false.

Conditions:

The equal and unequal relations having one or both character arguments are defined but they do not extend to the scan and reduction operators.

The relational functions with Boolean arguments apply also as logical functions.

The comparison tolerance applies to the results for numeric arguments. If the relation is true, to within the relative comparison tolerance based on the left argument, the result 1 is returned. See the discussion of ⎕*CT* for details.

Examples:

```
      3<2 3 4
0  0  1
      3 4 5≤5 4 3
1  1  0
      5=2 7 5
0  0  1
      1≠2 3ρ1 0 1 1 1 0
0  1  0
0  0  1
      3>5 3 1
0  0  1
      3≥5 3 1
0  1  1
```

```
      'CAB'='TAB'
0  1  1
      3='A'
0

      1 2 3≠'C'
1  1  1
      'RETN'≠'RATE'
0  1  0  1
```

Forms:

~ B	Not B
A ∧ B	A and B
A ∨ B	A or B
A ⍲ B	A nand B
A ⍱ B	A nor B

Where: A and B are Boolean numerics

Results:

Not: The result is the Boolean complement of B.

The dyadic logical functions, when extended by the six relational functions restricted to Boolean arguments, provide the ten non-trivial dyadic Boolean logical functions. The examples indicate their truth tables and their Boolean results.

Conditions:

The dyadic use of ~ as set difference is described with the set functions.

The comparison tolerance affects the determination whether a possibly non-integral numeric value is 1.

With this complete family of logical functions, it is rare that the not function is required. To illustrate:

```
A>B  ←→  A∧~B
A≤B  ←→  (~A)∨B
```

Examples:

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

CIRCULAR FUNCTIONS (1)
○

Forms:

```
  ○ B      Pi function: (pi times B)
A ○ B      Circular function A of B
```

Where: A selects the specific circular function
B is argument

Results:

```
○B ↔ B×3.14159265...

Direct Functions                  Arc (Inverse) Functions
                                                domain  range

0○N ↔ (1-N*2)*0.5                               1≥|N    1≥|X
1○R ↔ sin R            ¯1○N ↔ arcsin N          1≥|N    (○0.5)≥|X
2○R ↔ cos R            ¯2○N ↔ arccos N          1≥|N    (0≤X)∧X<○1
3○R ↔ tan R            ¯3○N ↔ arctan N                  (○0.5)>|X
4○N ↔ (1+N*2)*0.5      ¯4○N ↔ (¯1+N*2)*0.5      1≤|N    0≤X
5○N ↔ sinh N           ¯5○N ↔ arcsinh N
6○N ↔ cosh N           ¯6○N ↔ arccosh N         1≤N     0≤X
7○N ↔ tanh N           ¯7○N ↔ arctanh N         1>|N
```

Where: R is argument measured in radians
N is any numeric value in indicated domain
X is numeric result in indicated range

Conditions:

The domains indicated above (where restricted) for the arc function arguments are the ranges for the corresponding direct function results. The result ranges for the cyclic arc functions (arcsin, arccos, arctan) are the principal ranges.

Memory Aids: The positive left arguments apply to direct functions with unlimited domains for their right arguments. The negative left arguments apply to arc functions with indicated right argument domain and result range.

The even left arguments are associated with even functions (f(B))=f(-B); The odd left arguments are associated with odd functions (f(B))=-f(-B).

Both the trigonometric and hyperbolic forms are ordered sin (sinh), cos (cosh) and tan (tanh) with increasing magnitude of A.

The functions with square roots must yield real surds. Thus they all require non-negative radicands. The three forms shown are the only ones possible. The sign of A determines the sign of the constant (1 or ¯1) for the two forms that add the squared term. A=0 subtracts the squared term.

Examples:

```
          ○1 2 ¯3  ⍝ MULTIPLES OF PI
     3.14159  6.28319  ¯9.42478
          1 2 3○○0.5 0 0.25  ⍝ SIN 90°, COS 0°, TAN 45°
     1  1  1
          4 0 ¯4○0 0.8 1 ⍝ SQUARE ROOT FUNCTIONS
     1  0.6  0
          5 6 7○0  ⍝ SINH, COSH, TANH
     0  1  0
          ¯1 ¯2 ¯3○1 ⍝ ARCSIN, ARCCOS, ARCTAN IN RADIANS
     1.5708  0  0.785398
          (¯1 ¯2 ¯3○1)×180÷○1 ⍝ PRINCIPAL ANGLE IN DEGREES
     90  0  45
```

FACTORIAL,
COMBINATORIAL
FUNCTIONS
!

Forms:

```
  ! B        Factorial B
A ! B        Combinatorial A of B
```

Where: B is numeric
A is numeric

Results:

Factorial: For non-negative integer B the result is *B*×!*B*-1 ↔ !*B* where 1 ↔ !0 (alternatively ×/⍳*B* ↔ !*B* in one origin).

For non-integer B the result is the generalization of the factorial, the Gamma function of B+1:

Gamma (B+1) ↔ !*B*

Factorial is singular (undefined) for negative integer B.

Combinatorial: The result is (!*B*)÷(!*A*)×(!*B*-*A*) so long as all the indicated factorials are defined.

For non-negative integer A, B and A ≤ B, the result is the number of combinations of B things taken A at a time.

For A > B, the result is identically 0.

For non-integer A or B, the result is a generalization of combinations. It is related to the complete Beta function of A and B:

```
Beta (A,B)  ↔ ÷B×(A-1)!A+B-1
            ↔ (!A-1)×(!B-1)÷!A+B-1
```

Examples:

```
      !0 1 2 3 4 5 6 ⍝ FACTORIALS
1  1  2  6  24  120  720
      !¯0.5     ⍝ ↔ GAMMA (0.5) ↔ (○1)*0.5
1.77245
      !¯2.9 ¯1.9 ¯0.9 0.1 1.1 2.1
5.56345  ¯10.5706  9.51351  0.951351  1.04649  2.19762
      0 1 2 3 4!4 ⍝ COMBINATIONS OF 4 TAKEN 0 1 2 3 4 AT A TIME
1  4  6  4  1
      1.1!2 3 4 5
1.98713  3.13758  4.3277  5.54833
```

Each scalar primitive function applies element by element to its arguments.

Monadic ⊙B where ⊙ is any monadic scalar primitive function

The result of a monadic scalar primitive applied to an array B is an array of the same shape as B. Each element of the result is determined by applying the function ⊙ to the corresponding element of B the argument.

Dyadic A⊗B where ⊗ is any dyadic scalar primitive function

If A and B are arrays of the same shape, the result also has that shape. Each element of the result is determined by applying ⊗ to the corresponding elements of A and B.

Coercion is the process of making two data objects *conformable* for the dyadic function to which they both are arguments. Conforming arguments have the same shape. Coercion generally replicates the smaller size object to the rank and shape of the other.

If either A or B is a single, it is effectively coerced by replication to the shape of the other array and the result is as above. The single element is one argument for ⊗ applied with each element of the array as the other argument.

If both A and B are singles, the result is a single element object with rank that of the larger rank of A or B.

A⊗[K]B qualified application of ⊗ along dimension K

If the ranks of A and B differ by one and the shapes are the same when dimension K is elided from the one with larger rank, then the result has the same rank and shape as the larger rank array. Elements of the result are formed after first effectively coercing the smaller rank array to have the same shape as the larger rank array. This coercion is by replication of the entire smaller rank array as a plane for each position on dimension K of the larger. If K refers to the last dimension of the larger rank array, it may be elided. K is a single.

Without loss of generality, let A be the larger rank array, then the coercion condition may be expressed as:

```
    (K≠(ιρρA))/ρA ↔ ρB
```

For J each scalar value in ι(ρA)[K] the plane of the result R so determined is:

```
    R[...;J;...;N] ↔ A[...;J;...;]⊗B
```

EXTENSION TO
ARRAYS OF SCALAR
FUNCTIONS (2)

Examples:

```
     -3              ⍝ MONADIC SCALAR
¯3
     -¯2 0 3         ⍝ MONADIC VECTOR
2  0  ¯3
     A               ⍝ ARRAY (MATRIX)
1  2  3
4  5  6
     -A              ⍝ MONADIC ARRAY
¯1  ¯2  ¯3
¯4  ¯5  ¯6
     10+A            ⍝ SCALAR + ARRAY
11  12  13
14  15  16
     A-4             ⍝ ARRAY - SCALAR
¯3  ¯2  ¯1
 0   1   2
     A+A             ⍝ ARRAY + ARRAY WITH SAME SHAPE
 2   4   6
 8  10  12
     A+1 1 1⍴100     ⍝ ARRAY + SINGLE OF RANK 3
101  102  103
104  105  106
     A+[1]10 20 30   ⍝ ARRAY + VECTOR ALONG FIRST DIMENSION
11  22  33
14  25  36
     A+0 10          ⍝ ARRAY + VECTOR ALONG LAST DIMENSION
 1   2   3
14  15  16
     B               ⍝ RANDOM ARRAY
13  17  12  11
 1   5  29   4
 3  16   6  19
     ⌈/B             ⍝ MAXIMA OVER ROWS
17  29  19
     B=⌈/B           ⍝ LOCATION OF ROW MAXIMA OF B
0  1  0  0
0  0  1  0
0  0  0  1
```

PRIMITIVE OPERATORS.

Operators are provided that have one or two function arguments and produce a new function from them. This function is then applied to the data object arguments.

The 21 scalar dyadic functions are the only primitive functions that are used with the operators.

The following primitive operators are provided:

Operator	Possible functions
outer product	21
reduction	21
scan	21
inner product	441

The examples given include some of the more useful operators. The user should be aware of the many opportunities to use these and other operators as well.

Reduction and scan have a dimension selector appearing to the right of the function character and indicating in brackets the index number or dimension of function application. The index number is a single and is origin sensitive.

Assignments achieved by modification or modified insertion may be viewed as primitive operators, even though they are actually only a brief notation for the corresponding replace and insert functions.

OUTER PRODUCT OPERATOR (1)
∘.⊕

Form:

A ∘.⊕ B   Generalized outer product of A with B using function ⊕

Where: A is a data object
B is a data object
⊕ is any primitive dyadic scalar function:
⌊ ⌈ + - × ÷ | * ⍟ < ≤ = ≥ > ≠ ∧ ∨ ⍲ ⍱ ○ !

Results:

The result is a data object with rank (⍴⍴*A*)+⍴⍴*B* and shape (⍴*A*),⍴*B* formed by applying ⊕ between all pairs of elements; the first from A and the second from B.

If both A and B are vectors, the matrix result may be considered to be a table of values formed with A as the left argument and B as the right argument. The elements of A form the row headings; the elements for B form the column headings. If desired, the headings may be catenated onto the matrix result.

Conditions:

Outer product generates a data object with size that is the product of the sizes of its arguments. This may give a space limit error report. See Appendix B for suggestions on controlling space.

If reduction is the next operator to be applied after an outer product, they sometimes can be combined. This will avoid generating the large object, only to immediately reduce it again.

Examples:

```
     1 2 3∘.+1 2 3 4   ⍝ ADDITION TABLE
2  3  4  5
3  4  5  6
4  5  6  7
     1 2 3∘.⌈1 2 3 4   ⍝ MAXIMUM TABLE
1  2  3  4
2  2  3  4
3  3  3  4
     1 2 3∘.≥1 2 3 4   ⍝ NUMERIC COMPARE
1  0  0  0
1  1  0  0
1  1  1  0
     ' *'[1+6 5 4 3 2 1∘.≤1 3 4 2 5 6]   ⍝ HISTOGRAM
     *
    **
  * **
 ** **
 *****
******
     'ABC'∘.='BANANA'  ⍝ CHARACTER COMPARE
0  1  0  1  0  1
1  0  0  0  0  0
0  0  0  0  0  0
     1 2∘.○○÷6 3 2 1   ⍝ SIN COS 30 60 90 180 DEGREES
 5.00000E¯1    8.66025E¯1    1.00000E0     5.12669E¯12
 8.66025E¯1    5.00000E¯1    2.56334E¯12  ¯1.00000E0
```

REDUCTION OPERATOR (1) / ⌿

Forms:

⊕/[K] A	⊕ Reduction of A along dimension K from the first
⊕/ A	⊕ Reduction of A along last dimension
⊕⌿[K] A	⊕ Reduction of A along dimension K from the last
⊕⌿ A	⊕ Reduction of A along first dimension

Where:

A is a numeric data object
K is a dimension selector (origin sensitive); *K∊⍳⍴⍴A*
⊕ is any dyadic scalar primitive function:
⌊ ⌈ + - × ÷ | * ⍟ < ≤ = ≥ > ≠ ∧ ∨ ⍲ ⍱ ○ !

Results:

The reduction operator applies the indicated function to all planes across the indicated dimension. The forms with [*K*] indicate the dimension K explicitly; the other two forms implicitly specify the dimension.

The rank of the result for shaped data object A is one less than the rank of A. K is the dimension eliminated. The shape of the result is (*K≠⍳⍴⍴A*)/*⍴A*.

For scalar A, the result is A.

For vector A, the result is as if ⊕ were placed between the last two elements of the vector and then the resulting expression executed between that pair. The scalar result replaces the pair. This sequence is repeated along the entire vector until the last scalar result is returned. This sequence is equivalent to placing ⊕ between each element of the vector and executing the resulting expression.

For array A, each vector along the indicated dimension is treated as above.

Conditions:

Each partial result must match in type and be in the right argument domain for the next occurrence of ⊕.

The only exception to the simpler explanation to reduction of a vector given above is that =/ and ≠/ are undefined for character data objects even though these primitive dyadic scalar functions are defined for mixed type data.

Examples:

```
      +/1 2 3
6
      ρ+/1 2 3

      +/[1]2 3ρι6
5  7  9
      +⌿2 3ρι6
5  7  9
      +⌿[1]2 3ρι6
6  15
      +/[2]2 2 2ρι8
 4  6
12 14
      -/ι6
¯3
      ×/1 2 3
6
      ÷/ι6
0.3125
      ÷/×⌿3 2ρι6
0.3125
```

```
     |/3 5.5 17
0.5
     ⌈/1 3 2 5 ¯7
5
     ⌊/¯4 ¯7 3 8 0
¯7
     */4 3 2
262144
     ⍟/2*1 4 32
3
     !/3 4 5
10
     ≤/1 3 5 ⍝ LEFTMOST 1≤1
1
     ≤/2 4 6 ⍝ LEFTMOST 2≤1
0
     ∨/0 1 1 ⍝ LEFTMOST 0∨1
1
     ~∨/0 1 1
0
     ⍱/0 1 1 ⍝ LEFTMOST 0⍱0
1
```

SCAN
OPERATOR (1)
⊕\

Forms:

⊕\[K] A	⊕ Scan of A along dimension K from the first
⊕\ A	⊕ Scan of A along last dimension
⊕⍀[K] A	⊕ Scan of A along dimension K from the last
⊕⍀ A	⊕ Scan of A along first dimension

Where:

A is a numeric data structure
K is a dimension selector, *K*∊ ⍳⍴⍴*A*
⊕ is any primitive dyadic scalar function:
⌊ ⌈ + - × ÷ | * ⍟ < ≤ = ≥ > ≠ ∧ ∨ ⍲ ⍱ ○ !

Results:

The rank and the shape of the result are the same as A.

The dimension selector K determines the dimension vectors along which scan is applied.

For scalar A, the result is scalar A provided A is in the domain of a valid right argument of ⊕.

For vector A, element I of the result R is formed from ⊕ reduction of the first I elements of the vector *R*[*I*] ↔ ⊕/*I*↑*A* (in one origin).

For array A, each vector along the dimension K of A is developed as in the case of vector A.

Conditions:

The corresponding reduction must be defined for scan to be defined.

Examples:

```
      +\1 2 3 4         ⍝ TRIANGULAR NUMBERS
1  3  6  10
      A                 ⍝ ARRAY
1  2  3
4  5  6
      +\A               ⍝ SCAN ALONG ROWS
 1   3   6
 4   9  15
      +\[1]A            ⍝ SCAN DOWN COLUMNS
1  2  3
5  7  9
      +⍀A               ⍝ SCAN ALONG FIRST DIMENSION
1  2  3
5  7  9
      +⍀[1]A            ⍝ SCAN ALONG LAST DIMENSION
 1   3   6
 4   9  15
      ×\⍳6              ⍝ FACTORIALS
1  2  6  24  120  720
      -\⍳6              ⍝ DIFFERENCES
1  ¯1  2  ¯2  3  ¯3
      -\6⍴4 3
4  1  5  2  6  3
      ÷\⍳6              ⍝ QUOTIENTS OF ALTERNATING PRODUCTS
1  0.5  1.5  0.375  1.875  0.3125
      ⌈\3 2 4 0 6       ⍝ SEQUENCE OF ENCOUNTERED MAXIMA
3  3  4  4  6
      ∧\1 1 0 0 1       ⍝ LEADING ONES
1  1  0  0  0
      ∨\0 0 1 0 1       ⍝ LEADING ZEROS
0  0  1  1  1
      <\0 0 1 0 1       ⍝ FIRST ONE
0  0  1  0  0
      ≤\1 0 1 1 0       ⍝ FIRST ZERO
1  0  1  1  1
      X                 ⍝ AN EXPRESSION STRING OF CHARACTERS
A+((I×J)⍴K)÷B
      +\(X='(')-X=')'   ⍝ PARENTHESIS DEPTH IN STRING X
0  0  1  2  2  2  2  1  1  1  0  0  0
      Y                 ⍝ RAGGED ARRAY
  ALIGN
   ALL
 LEFT
      (+/∧\Y=' ')⌽Y     ⍝ LEFT JUSTIFY Y
ALIGN
ALL
LEFT
```

INNER PRODUCT OPERATOR (1) ⊕.⊗

Form:

A ⊕.⊗ B   Generalized inner product of A with B using functions ⊕ and ⊗.

Where: A and B are conforming data objects
⊕ ⊗ are any primitive scalar dyadic functions:
⌊ ⌈ + - × ÷ | * ⍟ < ≤ = ≥ > ≠ ∧ ∨ ⍲ ⍱ ○ !

Results:

Elements of the result are formed by taking conforming vectors along the last dimension of A and along the first dimension of B, applying ⊗ between them, and then reducing the result by ⊕.

The rank of the result is (0⌈¯1+⍴⍴A)+0⌈¯1+⍴⍴B.

The shape of the result is (¯1↓⍴A),1↓⍴B.

For vector or scalar arguments: the result is scalar ⊕/A⊗B.

For A vector (or scalar), B matrix the vector result R has element R[I]←⊕/A⊗B[;I].

For A matrix, B vector (or scalar) the vector result R has element R[I]←⊕/A[I;]⊗B.

Generally for A and B arrays, the array result R has element R[I;...;K;L;...;N]←⊕/A[I;...;K;]⊗B[;L;...;N].

Conditions:

Conformability requires that after allowed coercions, (¯1↑⍴A)=1↑⍴B   The valid coercions are:

Scalar A becomes (1↑⍴B)⍴A.

If 1=¯1↑⍴A then the plane across that last dimension is replicated (1↑⍴B) times:

```
A←(1⌽⌽⍳⍴⍴A)⍉⍉((1↑⍴B),¯1↓⍴A)⍴A
```

Scalar B becomes (¯1↑⍴A)⍴B.

If 1=1↑⍴B then the plane across that first dimension is replicated (¯1↑⍴A) times:

```
B←((¯1↑⍴A),⍴B)⍴B
```

Examples:

```
        1 2+.×3 4           ⍝ (1×3)+2×4 ←→ +/1 2×3 4
11
        2+.×2 3 4           ⍝ +/2 2 2×2 3 4
18
        5 3-.×3 2           ⍝ -/5 3×3 2
9
        A23
1  5  3
6  2  4
        B34
1  7  5  4
4  2  3  5
5  6  2  1
        A23+.×B34           ⍝ CONVENTIONAL INNER PRODUCT
36  35  26  32
34  70  44  38
        A23⌊.⌈B34           ⍝ MINIMAX
1  5  3  3
4  2  3  4
        A22
1  1
0  1
        B22
0  1
1  0
        A22∨.∧B22           ⍝ MINTERM
1  1
1  0
        A22∧.=1             ⍝ SINGLE COERCED TO 1 1
1  0
        1 0∧.=A22
1  0
        A22∧.=B22
0  0
1  0
        'ON'∧.=2 3⍴'FORANY' ⍝ CHARACTER MATCH ROW WITH COLUMN
0  1  0
```

IDENTITIES FOR SCALAR DYADIC PRIMITIVE FUNCTIONS

An identity argument for a dyadic scalar primitive function is that value which when the function is applied with any other argument returns that other argument. Let I be the identity argument, A the other argument and ⊕ a scalar dyadic primitive function:

Left identity:	*A* ↔ *I* ⊕ *A*
Right identity:	*A* ↔ *A* ⊕ *I*
Two-sided identity:	*A* ↔ *A* ⊕ *I* ↔ *I* ⊕ *A*

The result of the reduction operator (using a primitive dyadic scalar function) on an empty vector or a length zero coordinate of an array is the identity (if it exists) for that function. If the indicated dimension is the only one with length 0, the result is replication of the identity element in the entire plane across that dimension, so long as some identity element exists.

Inner product and base value are both based on reduction, so they also have this property when applied to a zero length coordinate.

Table 5-1 shows for each primitive scalar dyadic function the identity element if it exists, and whether it is left, right or two-sided (both).

Table 5-1 Identities for Scalar Dyadic Primitive Functions

For numeric arguments

⊕	identity	side
⌊	MAX *	both
⌈	-MAX *	both
+	0	both
-	0	right
×	1	both
÷	1	right
|	0	left
⋆	1	right
⍟	none	
○	none	
!	1	left

For Boolean arguments only

⊕	identity	side
<	0	left
≤	1	left
=	1	both
≥	1	right
>	0	right
≠	0	both
∧	1	both
∨	0	both
⍲	none	
⍱	none	

*MAX is the largest numeric value directly representable:
4.31359146674*E*68 ↔ *MAX* ↔ ⌊/⍳0

MIXED PRIMITIVE FUNCTIONS.

The mixed primitive functions include both monadic and related dyadic functions that apply to shaped data objects as arguments.

The functions generally use structure properties instead of the element values.

Rules for conformability, coercions, and extension from vector arguments to higher rank objects are more complex than for the scalar primitive functions.

The mixed or structure primitives may be classified as:

shape, reshape functions
integers, index of functions
ravel, catenate, laminate functions
reverse, rotate functions
transpose, permute functions
compress, expand functions
take, drop functions
set functions
grade functions
random roll, deal functions
base value function
represent functions
matrix inverse, divide functions

Many of these functions have a *dimension selector* appearing to the right of the function character and indicating in brackets the *index number* or dimension of function application. The index number *is a single* and is origin sensitive.

Some of the mixed primitive functions augment an existing data object with *fill* elements. The value of a fill element is 0 if the type of the object is numeric; or is a blank space if the type of the object is character.

SHAPE, RESHAPE FUNCTIONS
ρ

Forms:

ρ B	Shape of B
A ρ B	A reshape of B

Where: A is a non-negative integer vector or single
B is a data object, either numeric or character

Results:

Shape: The result is an integer vector indicating the length of each dimension of the data object B. In origin one, ρB indicates the largest index value for each dimension. In either origin the index domain for dimension I of B is ⍳(ρ*B*)[*I*].

Reshape: The result is an array whose shape is A, and whose elements are taken in raveled order from B and are repeated as often as necessary. Fill of the type of B is used if B is empty.

If A is an empty numeric vector, the result is scalar. Single A is coerced to a one element vector. If A contains any zero element, the result is an empty array.

Examples:

```
      ρ1 2 3
3
      7ρ1 2 3
1  2  3  1  2  3  1
      2 3ρ11 12 13 21 22 23
11  12  13
21  22  23
      3ρ1
1  1  1
      ρ12345    ⍝ SCALAR

      ρρ12345   ⍝ RANK 0
0
      ρ,12345   ⍝ VECTOR
1
      0ρ0   ⍝ NUMERIC

      ρ0ρ0   ⍝ EMPTY VECTOR
0
      (0ρ0)ρ2 3ρ⍳6   ⍝ SCALAR
1
      ρ(0ρ0)ρ2 3ρ⍳6
```

```
      ρ'APL CAN DO'
10
      10ρ'∘'
∘∘∘∘∘∘∘∘∘∘
      2 3ρ'ADDONE'
ADD
ONE
      ρ2 3 4ρ'A'
2  3  4
      ρ'12345'   ⍝ VECTOR
5
      ρ'A'       ⍝ SCALAR

      ρρ'A'      ⍝ RANK 0
0
      ρ,'A'      ⍝ VECTOR
1
      ρρ,'A'     ⍝ RANK 1
1
      ''    ⍝ CHARACTER

      ρ''   ⍝ EMPTY VECTOR
0
```

INTEGERS,
INDEX OF
FUNCTIONS
ι

Forms:

ι A	Integers to A
B ι C	Index of C in B

Where:

A is a non-negative integer single
B is a vector
C is a data object

Results:

Integers: The result is a vector containing the first A integers in ascending order, starting with the index origin. ,*A* ↔ ρι,*A*. Also called index generator.

Index of: The result is a data object with the same shape as C with integer elements. Each element of the result indicates the index position (of the first occurrence) in B of the corresponding element of C. The result range is ι1+ρ*B*. For any element of C not occurring in B, the corresponding result element is ⎕*IO*+ρ*B*.

Conditions:

Both functions are index origin dependent: ι1 ↔ ,⎕*IO*.

The comparison tolerance applies to determine if A is an integer and if an element of C is in B.

Examples:

```
      ⎕IO←1⍝ ORIGIN 1
      ι5
1  2  3  4  5
      ι0⍝ EMPTY NUMERIC VECTOR

      ι1⍝ ORIGIN SINGLE VECTOR
1
      3 1 1 2ι1 2 3 4 5
2  4  1  5  5
      'ABCDE'ι'BEAR'
2  5  1  6
      ⎕Dι'301'⍝ ⎕D↔'01...9'
4  1  2
      '+-×÷'ι'A+B×CD'
5  1  5  3  5  5
```

```
      ⎕IO←0⍝ ORIGIN 0
      ι5
0  1  2  3  4
      ι0

      ι1⍝ ORIGIN VECTOR
0
      3 1 1 2ι1 2 3 4 5
1  3  0  4  4
      'ABCDE'ι'BEAR'
1  4  0  5
      ⎕Aι3 4ρ'APL DOESWELL'
 0  15  11  26
 3  14   4  18
22   4  11  11
```

RAVEL, CATENATE, LAMINATE FUNCTIONS (1)

,

Forms:

, B	Ravel B into a vector
A , B	Catenate B to the last dimension of A
A ,[K] B	Catenate B to dimension K of A
A ,[D] B	Laminate A as the first plane and B as the last plane of a new dimension between dimensions ⌊D and ⌈D

Where:	A and B are data objects of the same type K is an index number of A or B D is a non-integer dimension injector

Results:

Ravel: Form a vector from the elements of B in row major order: first (leftmost) to last along last dimension, then first (topmost) to last along second last dimension, etc. Ravel of a scalar returns a one element vector.

Catenate: Join two conformable data structures of the same type, B after A (elements from B will then have larger indices along the joined dimension). The rank of the result is 1⌈(⍴⍴*A*)⌈⍴⍴*B*. If both A and B are scalars or vectors, the result is a vector formed by appending B after A.

Catenate to dimension K: If either A or B is an array, and the other is an object of rank one smaller and shape the same as a plane across dimension K of the larger rank object (the same shape as when dimension K of the larger rank argument is omitted), then catenation increases the length of dimension K by one and the smaller rank object A (or B) becomes the first (or last) plane across the kth dimension of the result. A scalar is coerced by replication to have the shape of all but dimension K and the above catenation is performed.

If both A and B have the same rank and the same shape except for dimension K, then the result of catenate is an array with shape the same as A and B except that the length of that dimension K becomes the sum of the lengths of that dimension in A and B with the first planes across dimension K from A and the last planes from B.

The [K] may be omitted if it refers to the last dimension of the larger rank object.

Laminate: Create a structure with a new dimension of length two. Laminate may be recognized distinct from catenate by the arbitrary fractional part of D, identifying the new dimension being injected (either before the first, between two existing, or after the last dimension). Elements from A are placed in the first plane across the new dimension and elements from B are placed in the second plane across that new dimension.

The possible values for the integer part D are from one less than the first dimension number to the last dimension number. The fractional part of D must be non-zero. Note that in 0 origin D may be negative.

Either the shapes of A and B must match, or one of A or B must be a scalar. A scalar is coerced by replication to the shape of the other argument.

The rank of the result is 1+(⍴⍴A)⌈⍴⍴B. The shape of the result is the larger shape of A or B, augmented by the new dimension of length two.

Examples:

```
      ,3            ⍝ VECTOR
3
      ⍴,3
1
      A             ⍝ ARRAY
11  12  13
21  22  23
      ,A            ⍝ ROW MAJOR ORDER
11  12  13  21  22  23
      ⍴,A           ⍝ VECTOR
6
      A,2 4⍴¯1 0 1 2
 11   12   13   ¯1    0    1    2
 21   22   23   ¯1    0    1    2
      0,[1]A        ⍝ FIRST DIMENSION
 0   0   0
11  12  13
21  22  23
```

```
      ,'C'
C
      ⍴,'C'
1
      CA
ABCDEF
      ,3 2⍴CA
ABCDEF
      CA,[0.5]'?'
ABCDEF
??????
      0,[1.5]1 3
0  1
0  3
```

REVERSE,
ROTATE
FUNCTIONS (1)
⌽ ⊖

Forms:

⌽ B	Reverse along last dimension of B
⊖ B	Reverse along first dimension of B
⌽[K]B	Reverse along Kth dimension from front of B
⊖[K]B	Reverse along Kth dimension from end of B
A ⌽ B	A rotate along last dimension of B
A ⊖ B	A rotate along first dimension of B
A ⌽[K]B	A rotate along Kth dimension from front of B
A ⊖[K]B	A rotate along Kth dimension from end of B

Where:

B is a data object with shape
K is a dimension selector single with integer value in ⍳⍴⍴*B*

A is an integer data object, scalar or with shape the same as the planes across the dimension of B about which rotation is performed

Results:

The type, shape and rank of the result are the same as B. Each element of B occurs, generally in a different position in the result.

Reverse: The general form is ⌽[*K*]*B*. The order of the planes across dimension K is reversed. Thus, plane J of the result is plane ((⍴*B*)[*K*])-*J*+⎕*IO* of B.

If K = ⌈/⍳⍴⍴*B*, referring to the last dimension, the [*K*] may be elided, resulting in ⌽*B*.

Equivalent to the general form but referenced to the end or anti-origin rather the front of the shape is ⊖[*K*]*B*. Thus, ⌽[*K*]*B* ↔ ⊖[(⍴⍴*B*)+(2×⎕*IO*)-*K*+1].

If K = ⌊/⍳⍴⍴*B* or ⎕*IO* referring to the first dimension, the [*K*] may be elided, resulting in ⊖*B*.

If B is a matrix, lines through the forms without [*K*] indicate the axes of symmetry about which reversing takes place.

Rotate: The general form here described is *A*⌽[*K*]*B*. The other forms for determining the dimension for rotation are equivalently developed as above. A has shape a plane across the Kth dimension of B, i.e.,

```
(⍴A) ←→ (K≠⍳⍴⍴B)/⍴B
```

Each element in A determines the amount that the corresponding elements of all planes across dimension K are rotated cyclically (or end around). For an element of *A*≥0, the direction is toward decreasing indices. For an element of *A*<0, the direction is toward increasing indices.

The amount rotated is ((⍴*B*)[*K*])|*A*. Thus, there is a non-negative equivalent for any negative element of A.

Conditions:

If A is a scalar it is coerced to a plane with all elements the same:

```
A ← ((K≠⍳⍴⍴B)/⍴B)⍴A
```

Examples:

```
      ⌽1 2 3 4
4  3  2  1
      ⌽'LIVED' ⍝ REVERSED
DEVIL
      A
11  12  13
21  22  23
      ⌽A
13  12  11
23  22  21
      ⌽[1]A
21  22  23
11  12  13
      ⊖A
21  22  23
11  12  13
```

```
      1⌽1 2 3 4
2  3  4  1
      5⌽1 2 3 4
2  3  4  1
      ¯3⌽1 2 3 4
2  3  4  1
      1 2⌽A
12  13  11
23  21  22
      0 1 2⌽[1]A
11  22  13
21  12  23
      B
TAKE   OUT   EXTRAS
      (X∨1⌽X←B≠' ')/B
TAKE OUT EXTRAS
```

TRANSPOSE,
PERMUTE
FUNCTIONS(1)
⍉

Forms:

⍉ B	Transpose dimensions
A ⍉ B	Permute dimensions

Where: A is an integer numeric vector of index numbers
B is a data structure

Results:

Transpose: The result is an array with rank at least 2. The elements are the same as the elements of B with the order of the dimensions reversed.

If B is a scalar, the single result has shape 1 1.

If B is a vector with shape S, the result is a column matrix having shape S,1. If B is a matrix having shape S,T, the result R is a matrix having shape T,S such that element *R*[*I*;*J*] is the same as *B*[*J*;*I*].

Analogously, if B is an array, the shape of the result is ⌽⍴*B* and element *R*[*I*;*J*;...;*N*] ←→ *B*[*N*;...;*J*;*I*].

Permute dimensions: Each element of the result R is an element from B as specified by A. A must be a vector with shape the rank of B. A must contain the index origin and possibly successive integers referring to index numbers of the result 1 ←→ ∧/(⍳⌈/*A*)∊*A*. Index numbers may reoccur. The number of different integers determines the rank of the result: ⍴⍴*R* ←→ ⍴*A*∪⍳0.

If A is a permutation of ⍳⍴⍴*B* (no repeated dimensions) then the result shape is the A permutation of B: ⍴*R* ←→ (⍴*B*)[*A*] and element *R*[*I*;*J*;...;*M*] is: *B*[*A*[1];*A*[2];...;*A*[*M*]].

If any element of A is repeated, the rank of R will be smaller. In that case, the principal diagonal selection across the dimensions of B is taken where elements of A are repeated. The length of the result dimension is the minimum of the lengths of the dimensions on which the diagonal is being taken.

(⌽⍳⍴*B*)⍉*B* ←→ ⍉*B*
(⍳⍴*B*)⍉*B* ←→ *B*

Conditions:

Elements of A are origin sensitive. Examples are given in origin 1; they would be one smaller in origin 0.

If B is a scalar, then A must be the empty numeric vector and the result is an identity: *R* ↔ *B*

Examples:

```
      ⍉3      ⍝ SINGLE MATRIX
3
      ⍴⍉3
1  1
      ⍉3 4 5 ⍝ COLUMN MATRIX
3
4
5
      ⍴⍉3 4 5
3  1
      A
0  1  2
3  4  5
      ⍉A
0  3
1  4
2  5
      ⍉3 3⍴'AHAPIPLET'
APL
HIE
APT
      B
111  112
121  122

211  212
221  222
      C
111  112  113  114
121  122  123  124
131  132  133  134

211  212  213  214
221  222  223  224
231  232  233  234
```

```
      (⍳0)⍉3      ⍝ IDENTITY
3
      ⍴(⍳0)⍉3     ⍝ SCALAR
      (,1)⍉'ABC' ⍝ IDENTITY
ABC
      (,1)⍉3 4 5 ⍝ IDENTITY
3  4  5
      2 1⍉A       ⍝ ↔ ⍉A
0  3
1  4
2  5
      1 1⍉A       ⍝ DIAGONAL
0  4
      +/1 1⍉A     ⍝ TRACE
4
      3 1 2⍉B
111  211
112  212

121  221
122  222
       ⍝ R[I;J;K]↔B[K;I;J]
       ⍝ I,J,K IN 1 2
      2 1 1⍉C     ⍝ MATRIX
111  211
122  222
133  233
       ⍝ R[I;J]↔C[J;I;I]
       ⍝ I IN 1 2 3↔⍳⌊/3 4
       ⍝ J IN 1 2↔⍳2
      1 1 1⍉C     ⍝ VECTOR
111  222
       ⍝ R[I]↔C[I;I;I]
       ⍝ I IN 1 2↔⍳⌊/2 3 4
```

COMPRESS,
EXPAND
FUNCTIONS (1)
/ ⌿ \ ⍀

Forms:

A / B	Compress with A the last dimension of B
A ⌿ B	Compress with A the first dimension of B
A /[K]B	Compress with A dimension K of B
A ⌿[J]B	Compress with A dimension J from end of B
A \ B	Expand with A the last dimension of B
A ⍀ B	Expand with A the first dimension of B
A \[K]B	Expand with A dimension K of B
A ⍀[J]B	Expand with A dimension J from end of B

Where: A is a Boolean single or vector
B is an array of any type
K is an index number single, in ⍳⍴⍴*B*
J is an index number single, *K* ↔ (⌽⍳⍴⍴*B*)[*J*]

Results:

The rank of the result is the rank of B, with only the length of the indicated dimension altered.

Compress: The general form is A/[K]B. The Boolean compression vector A must be the same length as the dimension K being compressed of B: (⍴*A*)=(⍴*B*)[,*K*]

Planes across dimension K of B are selected in ascending order wherever the corresponding elements of A are 1, and planes are ignored wherever the elements of A are 0. Thus, the length of the Kth dimension of the result is +/*A*.

Expand: There must be as many 1's in A as there are elements along coordinate K of B: +/*A* ↔ (⍴*B*)[,*K*].

The result is an object with rank the same as A but having dimension K expanded to size ⍴A. Each 1 in A indicates the position along K of the planes of B. Each 0 in A indicates a plane created from fill. Depending on the type of B, the fill element is 0 for numeric, blank for character.

Conditions:

If A is a scalar, it is coerced to the length of the indicated dimension, i.e., (⍴*B*)[,*K*]⍴*A*.

Dimension selector J counts dimensions from the end, or anti-origin whereas K counts from the beginning *J* ↔ (⌽⍳⍴⍴*B*)[*K*]. For example:

```
A/[K]B ↔ A⌿[(⌽⍳⍴⍴B)[J]]B
A⌿[J]B ↔ A/[(⌽⍳⍴⍴B)[K]]B
```

K (or J) may be elided if the desired function applies to the last (or first) dimension respectively.

Examples:

```
      1 1 0 1/1 2 3 4
1  2  4
      A
1  2  3
4  5  6
      1 1 0/A
1  2
4  5
      1/1 2
1  2
      0/1 2

      1 0 1/A
1  3
4  6
      1 0/[1]A
1  2  3
      0 1⌿A
4  5  6
       0 1⌿[1]A
4  5  6
      1 1 0 1 0/'APPLY'
APL
      CA
USABLE
APPEAL
      1 1 0 0 0 1/CA
USE
APL
```

```
      1 1 0 1\1 2 4
1  2  0  4
      1 0 1\[1]A
1  2  3
0  0  0
4  5  6
      1 0 1 1\1 0 1⍀A
1  0  2  3
0  0  0  0
4  0  5  6
      0\2 0⍴1
0
0
      ' '=0\0/'AB'⍝ CHARACTER?
1
      0=0\0/3 4     ⍝ NUMERIC?
1



      1 0 1 0 1\'APL'
A P L
      1 0 1⍀2 3⍴'FORALL'
FOR

ALL
```

TAKE, DROP FUNCTIONS (1)
↑ ↓

Forms:

A ↑ B	Take corner with shape A from B
A ↓ B	Drop A planes from B
Where:	A is integer vector or single (ρ,*A*)=ρρ*B* B is data object

Results:

Each function returns a shaped data object of the same type as B having a corner that is also a corner of B. The rank of the result is ρρ*B*.

A[*I*] refers to the number of planes across dimension I of B. Elements *A*[*I*]>0 reference the first *A*[*I*] successive planes in increasing index order starting at the origin. Elements *A*[*I*]<0 reference the last *A*[*I*] successive planes in increasing index order ending at the anti-origin, (ρ*B*)[*I*]. *A*[*I*]=0 references no planes.

Take: The result has shape A. The planes of the result across each dimension remain in the original order as they had in B. The result is strictly a subarray of B if (|*A*)≤ρ*B*.

Overtake: Occurs for all the dimensions *K*[*I*] such that 0<*K*←(|*A*)-ρ*B*. In this case, *K*[*I*] planes of fill are appended before (after) the (ρ*B*)[*I*] planes as the sign of *A*[*I*] is negative (positive). The fill is blank for character type B and zero for numeric type B.

Drop: The result has shape 0⌈(ρ*B*)-|*A*. If *A*[*I*]>0 then the first *A*[*I*] planes are dropped from the origin of dimension I of B. If *A*[*I*]<0 then the last |*A*[*I*] planes are dropped from the anti-origin of dimension I of B.

Conditions:

If A is a single, it is coerced to a vector:

```
A ←,A
```

If B is a scalar, it is coerced to a single with rank ρ,*A*.

```
B ← ((ρ,A)ρ1)ρB
```

Take and drop both return a "corner" of B. If no overtake is required, then the same corner can be specified with either take or drop.

Take or drop are origin independent. They often can be used in place of indexing, possibly in conjunction with other structure primitive functions such as compress and rotate to permit processing on a dense array.

Examples:

```
      3↑1 2 3 4 5
1  2  3
      ¯3↑1 2 3 4 5
3  4  5
      4↑1 2      ⍝ OVERTAKE
1  2  0  0
      A
1  2  3
4  5  6
      1 ¯2↑A
2  3
      ⍴1 ¯2↑A
1  2
      ¯1 ¯3↑A
4  5  6
      3 ¯4↑A     ⍝ 0 FILL
0  1  2  3
0  4  5  6
0  0  0  0
      3↑'ABCDE'
ABC
      ¯6↑'END'   ⍝ BLANK FILL
   END
      2 3↑7      ⍝ COERCED
7  0  0
0  0  0
```

```
      ¯2↓1 2 3 4 5
1  2  3
      2↓1 2 3 4 5
3  4  5
      4↓1 2 ⍝ EMPTY

      ⍴4↓1 2
0
      ¯1 1↓A
2  3
      ⍴¯1 1↓A
1  2
      1 0↓A
4  5  6
      2 0↓A ⍝ EMPTY

      ⍴2 0↓A
0  3
      ¯2↓'ABCDE'
ABC
      (,3)↓'ABCDE'
DE
      3↓'ABCDE'
DE
      ¯2 ¯5↑'?'

    ?
```

SET
FUNCTIONS (1)

∊ ⊂ ⊃ ∪ ∩ ~

Forms:

A ∊ B — Membership of A in B
A ⊂ B — Is A a subset of B
A ⊃ B — Is A a superset of B
C ∪ D — Union of C and D, unique elements in (,C),,D
A ∩ B — Intersection of A and B, unique elements in both (,A) and (,B)
A ~ B — Set exclusion, unique elements in A but not in B

Where: A,B are data objects
C,D are data objects of the same type

Results:

Membership: The shape of the Boolean result is the shape of A. Each element is 1 if the corresponding element of A occurs anywhere in B; 0 otherwise.

Subset: The Boolean scalar result is 1 if all unique elements of A also appear in B; 0 otherwise.

Superset: The Boolean scalar result is 1 if all unique elements of B also appear in A; 0 otherwise.

Union: The result is a vector of the common type of C and D containing the unique elements in (,C),,D in the order that they first occur in the catenation.

Intersection: The result is a vector of the same type as B containing the unique elements of A also occurring as elements of B. The order is the order that they first occur in ,A. For non-empty result, the types of A and B must be the same.

Set exclusion: The result is a vector of the same type as A containing unique elements of A that are not also in B in the order of the first occurrence in A. Set exclusion is also referred to as set difference.

Conditions:

Union with an empty argument provides the unique elements in the originally non-empty argument.

SET FUNCTIONS (2)

∊ ⊂ ⊃ ∪ ∩ ~

Examples:

```
      1 2 2 3 4∊2 5
0  1  1  0  0
      (2 3⍴⍳6)∊3 1 9
1  0  1
0  0  0
      1 3 5 3⊂2 3⍴⍳6
1
      1 3 5 3⊃0 1
0
      1 3 5 3∪4 3 2
1  3  5  4  2
      1 3 5 3∩4 3 2
3
      1 3 5 3~4 3 2
1  5
      (⍳0)⊃1 2 2 3~3 2 1
1
```

```
      'A+4×ABC+3'∊'ABCDE'
1  0  0  0  1  1  1  0  0
      (2 4⍴'GOODWORK')∊'BOOK'
0  1  1  0
0  1  0  1
      'APL'⊂'APPLICATIONS'
1
      'BASIC'⊃'APL'
0
      'EASE'∪'SAY'
EASY
      'APPLIED'∩'PLAN'
APL
      'APPLE'~'CORE'
APL
      ''⊃'AB'∩'CD'
1
```

GRADE FUNCTIONS ⍋ ⍒

Forms:

```
⍋ A        Grade up A
⍒ A        Grade down A
```

Where: A is a numeric vector

Results:

Each result is a permutation of the integers in *⍳⍴A*. The permutation can be used as a vector of indices to the selection function which when applied to A will produce a monotonic sequence.

Grade up: The selection sequence ascendingly sorts the argument. *A*[⍋*A*]

Grade down: The selection sequence descendingly sorts the argument. *A*[⍒*A*]

Conditions:

The permutation can be used to construct multi-column sorts, one column vector at a time starting from the last. It can also be used for key sorts, moving only the key indices rather than the entire related records.

Duplicate components in A return indices in increasing order for either function.

The results are origin sensitive, the examples are for origin 1: 1 ↔ ⍳1 ↔ ,⎕*IO*.

Examples:

```
     ⍋5 8 4 4 2                         ⍒5 8 4 4 2
5  3  4  1  2                      2  1  3  4  5
     5 8 4 4 2[⍋5 8 4 4 2]⍝ SORT        5 8 4 4 2[⍒5 8 4 4 2]
2  4  4  5  8                      8  5  4  4  2
     ⍋2.1 3.2 4.3 3.2                   ⍒2.1 3.2 4.3 3.2
1  2  4  3                         3  2  4  1
     A
1  1  0  0  1  0  1  1  0          ⍝ IF X IS A PERMUTATION VECTOR
     ⎕←B←'ABCD','12345'            ⍝ THEN X ↔ ⍋⍋X
ABCD12345                               X
     ⍋A                            2  1  5  3  4
3  4  6  9  1  2  5  7  8               ⍋⍋X
     ⍋⍋A                           2  1  5  3  4
5  6  1  2  7  3  8  9  4
     B[⍋⍋A] ⍝ MERGE                     B[⍒⍋A] ⍝ REVERSE MERGE
12AB3C45D                          D54C3BA21
```

Forms:

```
      ? N      Roll, random choice from N
    A ? B      Deal, random choice of A from B
```

Where:

N is a positive integer vector or single
A is a non-negative integer single, A≤B
B is a non-negative integer single

Results:

Roll: For single N, a pseudo-random integer is returned in the range ⍳N. Each of the possible values from the population of size N has equal likelihood of occurring as the result; thus, sampling is done with replacement. The shape of the result is the shape of N. If N is a vector, element I is chosen from ⍳*N*[*I*].

Deal: A vector of length A is returned, with elements chosen randomly without replacement from ⍳B. If A=B, the result is a random permutation of ⍳B.

Conditions:

Roll and Deal results are origin sensitive.

Roll and Deal use a common pseudo-random number generator. A side-effect of execution of either of these functions is to change the current random link used to determine the next value. The random link value can be preset using the ⎕*RL* system variable. It can also be initialized to a specified default value in a clear workspace by using the)*SEED* system command.

Examples:

```
      ?10000                                4?5
301                                   1  5  4  3
      ?6 6                                  6?6
2  1                                  4  3  5  1  6  2
      ?6 6 6 6 6 6                          6?6
6  2  6  4  2  2                      5  1  2  4  6  3
      ?2 2 2 2 100 100                      0?10
1  2  1  1  36  87
```

⊥

Form:

A ⊥ B   Base A value of B

Where: A is a numeric data object
          B is a numeric data object

Results:

The numeric result is the conversion to decimal of B expressed in positional number base with radices the rows of A. This base can be a constant (such as 10 meaning powers of 10) or a vector of mixed values.

The result is the inner product of W (a weighting of A having the same shape as A) with B.

```
W+.×B
```

The shape is (¯1↓ρA),1↓ρB. Each vector along the last dimension of W is the positional weighting to be applied to corresponding vectors along the first dimension of B, where the most significant elements have the smallest index numbers. Each weighting vector is formed from the reversal of the product scan of the reversal of the vector along the last dimension of A having the first element eliminated and 1 catenated at the end. (I and K are scalars):

```
W[I;...;K;] ←→ ⌽×\⌽1↓A[I;...;K;],1
```

If neither A nor B are singles, then A and B must be conformable. The length of the last dimension of A must equal the length of the first dimension of B.

```
(¯1↑ρA) = 1↑ρB
```

The coercion of a single is by replication along the appropriate dimension to the length of the other. Scalars are treated as vectors.

If either A or B is the empty vector and the other is a single or empty vector, then the result is 0, the identity element for +/ι0.

Base value can be used to pack vectors of many small precision numbers into a single number. This is a space saving technique.

The numeric range for integers (¯1+8*13) is a limit for the results of base value that can be reconverted subsequently using the represent function '⊤'.

Examples:

```
                        ⍝ WEIGHTING COMPUTATION

      10 10 10⊥3 7 1    ⍝ 100 10 1+.×3 7 1
371
      10⊥3 7 1          ⍝ 100 10 1+.×3 7 1
371
      100⊥3 7 1         ⍝ 10000 100 1+.×3 7 1
30701
      2 2 2 2⊥1 1 0 1   ⍝ 8 4 2 1+.×1 1 0 1
13
      4 ¯3 2⊥1 3 2      ⍝ ¯6 2 1+.×1 3 2
2
      0 3 ¯4⊥2          ⍝ ¯12 ¯4 1+.×2 2 2
¯30
      3.5 2.5 1.5⊥4 3 2 ⍝ 3.75 1.5 1+.×4 3 2
21.5
      0 0 4⊥¯3 ¯2 ¯5    ⍝ 0 4 1+.×¯3 ¯2 ¯5
¯13
      A                 ⍝ ARRAY
1  2  3
4  5  6
      A⊥3 2 1           ⍝ (2 3⍴6 3 1 30 6 1)+.×3 2 1
25  103
      0 3⊥A             ⍝ 3 1+.×A
7  11  15
      A⊥3 2⍴⍳6          ⍝ (2 3⍴6 3 1 30 6 1)+.×3 2⍴⍳6
20  30
53  90
      HEX               ⍝ HEXADECIMAL VECTOR
0123456789ABCDEF
      ¯1+HEX⍳'D9F'
13  9  15
      16 16 16⊥13 9 15
3487
      16⊥¯1+HEX⍳'D9F'   ⍝ HEXADECIMAL TO DECIMAL
3487
                        ⍝ FUTURE VALUE OF CASH FLOWS AT 10∘/∘
      (1+.10)⊥100 200 50 ⍝ 1.21 1.1 1+.×100 200 50
391
                        ⍝ POLYNOMIAL (X*2)+(2×X*1)+¯8 AT X=¯4
      ¯4⊥1 2 ¯8         ⍝ (¯4*2 1 0)+.×1 2 ¯8
0
```

REPRESENT FUNCTION (1) ⊤

Form:

A ⊤ B   Base A representation of B

Where: A is a numeric data object
          B is a numeric data object

Result:

The result is the representation of B in the number system having as base(s) the vectors along the last dimension of A. The rank of the result is (ρρ*A*)+ρρ*B*. The shape of the result is (ρ*A*),ρ*B*.

If B is a scalar and A is a scalar, the result is *A*|*B*.

If B is a scalar and A is a vector, the result is the representation of B in the number system having (possibly mixed) base A.

For example:

```
R← 5 3 4 ⊤ 117
R[3] ↔          1 ↔ 4|117      QUOTIENT IS 29
R[2] ↔          2 ↔ 3|29       QUOTIENT IS 9
R[1] ↔          4 ↔ 5|9        QUOTIENT IS 1
R IS 4 2 1
```

This result is the same as if 117 had been 57 + 60 x J for any integer J.

If A is an array, each vector along the last dimension of A is a separate base for determining the corresponding element of the result. Thus, if A is a matrix, each column is a separate base vector.

If B is an array, each element is represented in the base vector(s) of A. This process is analogous to outer product.

Conditions:

The highest index 0 in a base vector returns the entire remaining quotient in that position of the result. All index values with smaller indices are 0.

Note that A values can be general numerics. Thus, fractional or negative base systems can be used.

Represent and base value are related by the following relation for vectors A and B:

If (|*B*)<|×/*A* then *B* ↔ *A*⊥*A*⊤*B*

Examples:

```
      10 10 10⊤234
2  3  4
      10 10 10⊤234 ¯234
2  7
3  6
4  6
      2 2 2 2⊤2 13 ¯1
0  1  1
0  1  1
1  0  1
0  1  1
      10 10⊤234 ¯234
3  6
4  6
      ⍝ HOUR MINUTE SECOND
      24 60 60⊤3723
1  2  3
      ⍝ INTEGER, FRACTION
      0 1⊤3.75
3  0.75
      0 1⊤3.75 ¯3.75
 3      ¯4
 0.75    0.25
      ⍝ 0 GETS REST OF QUOTIENT
      5 5 0 2⊤5
0  0  2  1
      10 10 10 ⊤10⊥2 3 4
2  3  4
      HEX
01234567891BCDEF
      HEX[1+16 16 16⊤3487] ⍝ DECIMAL TO HEXADECIMAL
D9F
```

```
      ¯10⊤234
¯6
      ¯10 ¯10 ¯10⊤234 ¯234
¯8  ¯3
¯4  ¯7
¯6  ¯4
      2 2 2 2⊤5
0  1  0  1
      2.5 0.4 0.5⊤0.62
2  0.2  0.12
      2 2 1⊤3.2 ¯3.2
1    0
1    0
0.2  0.8
      2 ¯2 2⊤2 ¯3
 1  1
¯1  0
 0  1
      (2 3⍴⍳6)⊤7 15
0  0
1  1
1  2

3  3
2  0
1  3
```

MATRIX INVERSE,
DIVIDE
FUNCTIONS (1)
⌹

Forms:

⌹ B	Matrix inverse of B
A ⌹ B	Matrix divide A by B

Where: A is either a vector or a matrix with at least as many rows as columns

B is a matrix with at least as many rows as columns

Results:

If B is singular, i.e., having fewer linearly independent rows than columns, a domain error results. Otherwise, B is non-singular and the following apply.

Inverse: The shape of the result is ⌽⍴B and the rank is 2. The result is the generalized inverse of B. If B is square, then

```
Identity matrix ↔ (⍳1↓⍴B)∘.=⍳1↓⍴B
                ↔ (⌹B)+.×B
                ↔ B+.×(⌹B)
```

If B is non-square, then the result is the generalized inverse.

```
Identity matrix (⍳1↓⍴B)∘.=⍳1↓⍴B ↔ (⌹B)+.×B
```

Matrix Divide: A and B must be conformable, i. e.,

```
(1↑⍴A)=1↑⍴B
```

The result is formally the same as (⌹B)+.×A. The rank of the result is the rank of A. The shape of the result is (1↓⍴B),1↓⍴A.

Conditions:

The finite precision of computation results in only the approximate inverse: the magnitudes of off-diagonal terms should be 0 but normally are small compared to the main diagonal terms of (⌹B)+.×B or B+.×⌹B. The matrix is ill-conditioned to the degree that the largest magnitude of the off-diagonal term approaches 1.

The comparison tolerance is used to determine singularity: with large comparison tolerance most coefficient matrices are "singular"; with the comparison tolerance ≤1E¯12, few matrices are considered singular.

The method used is Householder's orthogonal decomposition. It is chosen over the more efficient Gaussian elimination for the following reasons:

> complete stability unless the coefficient matrix is essentially singular
>
> readily detectable singularity
>
> single precision computations suffice
>
> generalizable to overdetermined systems of equations.

Although *A*⌹*B* is formally equivalent to (⌹*B*)+.×*A*, the former matrix divide is preferable as it only requires about half the computation and is more accurate.

A detailed discussion of these functions and some of the following examples are adapted from the article:

> Jenkins, M. A., "DOMINO-an APL Primitive Function for Matrix Inversion--Its Implementation and Applications", APL Quote Quad, Vol III, No. 4, February 10, 1972

Examples:

```
      B22
2  1
5  3
      B33
 3   2   5
 1  ¯5  ¯1
 2   1  ¯3
      A3
18  3  ¯4
      A32
 18   ¯5
  3    5
 ¯4    1
      ⌹B22 ⍝ INVERSE
 3  ¯1
¯5   2
      B22+.×⌹B22 ⍝CHECKS
 1.000E0      1.455E¯11
¯2.328E¯10    1.000E0
      (⌹B22)+.×B22
 1.000E0     ¯5.821E¯11
 1.164E¯10    1.000E0
```

```
      ⌹B33
0.1524     0.1048     0.219
0.009524  ¯0.181      0.07619
0.1048     0.009524  ¯0.1619
      ⌹⌹B33
3   2   5
1  ¯5  ¯1
2   1  ¯3
      (⌹B33)+.×A3
2.181  ¯0.6762  2.562
      A3⌹B33 ⍝ MATRIX DIVIDE
2.181  ¯0.6762  2.562
      B33+.×A3⌹B33 ⍝ CHECK
18  3  ¯4
      A32⌹B33 ⍝ TWO SETS
 2.181    ¯0.01905
¯0.6762   ¯0.8762
 2.562    ¯0.6381
      B33+.×A32⌹B33 ⍝ CHECK
18   ¯5
 3    5
¯4    1
```

MATRIX INVERSE, DIVIDE FUNCTIONS (3) ⌹

Linear Equations:

Find X, the solutions to the equation (B +.× X) = A, given arrays A and B.

```
      B                  ⍝ COEFFICIENT MATRIX
 3  2  5
 1 ¯3 ¯1
 2  1 ¯3
      A                  ⍝ VECTOR OF RIGHT HAND SIDES
18 ¯4 5
      A⌹B                ⍝ SOLUTION
3 2 1
      B+.×A⌹B            ⍝ CHECK
18 ¯4 5
      AA                 ⍝ SEVERAL SETS OF RIGHT HAND SIDES
 18  31
 ¯4  ¯5
  5   1
      AA⌹B               ⍝ SOLUTIONS
3 4
2 2
1 3
```

Interpolation:

Find coefficients of approximating polynomial Y = F (X) given that X is a vector of independent values and Y is a vector of corresponding values.

Approximate F by polynomial of order n with coefficients

```
A = A[0],A[1], . . . ,A[n]

Y[I] =  A+.×X[I]*⌽¯1+⍳⍴X
```

Solution for coefficients through the n+1 points in X, F (X)

```
A ← Y ⌹ X ∘.* ⌽ ¯1+⍳⍴ X
```

Interpolation at XX not necessarily in X

```
XX ⊥ A
```

For example, if F (X) is SIN X, find SIN 0.223 given SIN 0.1×⍳10.

```
      X←0.1×⍳10                 ⍝ INDEPENDENT VARIABLE
      Y←1○X                     ⍝ DEPENDENT VARIABLE, SIN (X)
      0.223⊥Y⌹X∘.*⌽¯1+⍳⍴X       ⍝ INTERPOLATED VALUE
0.221156329002
      1○0.223                   ⍝ ACTUAL COMPUTED SIN 0.223
0.221156329006
```

Linear Least Squares:

Estimate parameters A[I] occurring in a model to be fitted to data of the form:

```
Y = (A[1]×F1 X) + (A[2]×F2 X) + . . . + (A[n]×Fn X)
```

where F1, F2, . . . , Fn are functions of a single variable or of several independent variables.

The maximum likelihood estimator for the A[i] are given by the least squares solution to the overdetermined equations

```
(F +.× A) = Y
```

where F[;i] has the values Fi X; and Y are the observed data (more than n points).

The solution for A is

```
A ← Y ⌹ F
```

Linear curve fit

```
Y = (A[1] × X) + (A[2] × 1)
```

```
F←X∘.*1 0          ⍝ COEFFICIENT MATRIX
A←Y⌹F              ⍝ PARAMETERS OF BEST LINEAR FIT
YP←F+.×A           ⍝ PREDICTED VALUES
R←YP-Y             ⍝ RESIDUALS
```

Nth degree polynomial curve fit

```
F←X∘.*⌽0,⍳N
```

Multiple linear regression: If F is a matrix of the form:

```
F = 1,X
```

where X is the matrix of observations:

```
X[i;j]
```

is the value of variable j at observation i. Then the parameters of the linear regression model

```
Y = A[1]+(A[2]×X[1])+ . . . +(A[m+1]×X[m])
```

are

```
A ← Y ⌹ F
```

EVALUATE
FUNCTION
⍎

Form:

⍎ S   Evaluate string S

Where: S is character string representing an APL expression

Results:

The result is the same as if S were an input entry for evaluation. S is generally the result of expression elaboration. Computed strings can be developed and then evaluated.

Conditions:

S may not be a system command or any function definition and editing action.

Examples:

```
          ⍎'3+4'
      7
          ⍎'3','+-'[1],'4'
      7
          INDEX←1                        ⍝ SAMPLE VALUE
          ⍎'3','+-'[INDEX],'4'           ⍝ FUNCTION SELECTION
      7
          WORD←'ADD'                     ⍝ A SAMPLE STRING
          ⍎(⍕INDEX),'⌽',WORD             ⍝ USING DEFAULT FORMAT ⍕
      DAD
```

FORMAT FUNCTIONS.

Formatted character data structures can be produced using the format primitive functions. The monadic form provides an implicit format. The dyadic forms permit explicit specification of the desired format. The discussion common to all forms or comparing forms is contained here; detailed differences are described on subsequent pages.

Forms:

⍕ E	Implicit format
V ⍕ N	Numeric explicit format
C ⍕ E	Character explicit format
C ⍕(L)	Character explicit formatted list

Where:

E is a data object of numeric or character type
N is a numeric data object
V is a numeric vector defining the edit format
C is a character string defining the edit format
L is a list of components, separated by semicolons; each component is either a null, or a data object of any type.

Results:

The result is a character data object that represents the data objects(s) of the right argument, formatted as specified.

The implicit and numeric explicit forms preserve the lengths of all dimensions except the last dimension which is altered if E is numeric. E may be a vector, matrix or general array.

Each character explicit form accepts as right argument (or list components) only scalar, vector, or matrix data objects. The result is a character matrix having at least one row, and generally the maximum number of rows of any matrix in the list.

Conditions:

The numeric explicit form is more efficient where appropriate than the character explicit format. The character explicit format has many more capabilities.

FORMAT SYNTAX DIAGRAMS.

Syntax diagrams are directed graphs used to show the syntax clearly and concisely. The allowable constructs, defaults, alternatives, and iterations are recognized as encountered along any path. The diagrams are rigorous without being cumbersome.

The rules for interpreting these diagrams are simple:

syntactic units are either literal APL characters or descriptive names or underscored mnemonics

syntactic units are set off by spaces and separated by lines and nodes

any path traced along a forward direction of the arrows will produce a syntactically valid format

lines terminate at nodes: ∘ arrows indicate entrance directions

iteration is achieved by a leftward path ∘←

limited number of occurrences is shown by a "bridge" /2̅\ covering a number indicating the maximum number of crossings (here 2)

Format Function:

```
-->∘---------------------------- ⍕ --- Data Object ----->∘-->
   ↓                                                     ↑
   ∘- Numeric Format Vector ---- ⍕ -- Numeric Array ---->∘
   ↓                                                     ↑
   ∘- Character Format Vector -- ⍕ --∘-------- E -------->∘
                                     ↓                   ↑
                                     ↓     ∘← ; -∘       ↑
   E      is character or numeric    ↓     ↓     ↑       ↑
          data object                ∘- ( →∘---->∘- ) -->∘
                                           ↓     ↑
                                           ∘- E →∘
```

Numeric Format Vector:

```
      ∘←-------------------------------∘
      ↓                                ↑
   -->∘- Width →∘-- Decimal Digits -->∘-->   FIXED POINT
                ↓                      ↑
                ∘--------- 0 -------->∘      INTEGER
                ↓                      ↑
                ∘- ¯ Decimal Digits →∘       FLOATING POINT
```

Character Format Vector:

```
   o←----------- ; ----------o
   ↓                         ↑
   ↓ o←--------- , --------o ↑
   ↓ ↓                     ↑ ↑
--→o→o→o----- Clause ---→o→o→o---→
       ↓                 ↑
       o- R ( Clause ) -→o
```

Clause:

```
--→o-----------------→o--→
   ↓                  ↑
   o←------- , -------o
   ↓                  ↑
   o-----→o- Phrase -→o
   ↓      ↑
   o- R -→o
```

Phrase:

```
--→o------------------------------ <S> -------------------------→o--→
   ↓                                                             ↑
   o------------------------------- T --→o- C ------------------→o
   ↓                                     ↓                       ↑
   ↓                                     o----------------------→o
   ↓                                     ↑                       ↑
   o------------------------------- X --→o- W ------------------→o
   ↓                                                             ↑
   o------------------→o-----→o→o-- A --→o- W ------------------→o
   ↓                   ↓      ↑ ↓        ↓                       ↑
   ↓                   o- L -→o ↓        o----→o----------------→o
   ↓                            ↓        ↑     ↑                 ↑
   ↓                            o-- E --→o- W →o- .D -----------→o
   ↓                                                             ↑
   o-------→o→o-----→o→o-----→o→o-- F --→o- W →o- .D -→o         ↑
   ↓        ↑ ↓      ↑ ↓      ↑ ↓        ↓     ↓       ↓         ↑
   o- *<S> →o o- SN →o o- QU →o ↓        o----→o------→o→o------→o
                                ↓        ↑     ↑         ↓       ↑
                                o-- I --→o- W →o         o- SN -→o
```

SN is sign selector:

```
   o←-/2̄\----o←--------o
   ↓         ↑         ↑
--→o- - /1̄\-→o-- <S> -→o--→
   ↓         ↑
   o- o /1̄\-→o
   ↓         ↑
   o- + /1̄\-→o
```

R is replicator

S is string

C is column

W is width of field

D is decimal digits

QU is qualifier:

```
   o←--------o
   ↓         ↑
--→o- L /1̄\-→o--→
   ↓         ↑
   o- B /1̄\-→o
   ↓         ↑
   o- C /1̄\-→o
   ↓         ↑
   o- Z /1̄\-→o
```

IMPLICIT FORMAT FUNCTION (1)

⍕

Form:

⍕ E   Format E using implicit format

Where: E is a data object

Results:

The result is a character data object.

If E is of character type, the result is identically E.

If E is of numeric type, the result is formed by application of the rules:

> Every element of E is rounded according to the current print precision to get the specified number of significant digits (integers are not truncated and trailing fractional zeros are ignored) and then converted to characters.
>
> If E is scalar, one blank is prefixed.
>
> If E is vector, the result is also a vector. This result is the ravel of an array formed containing the character representation of each element. Sufficient columns are provided that at least one blank precedes each non-blank, and all decimal points are alined.
>
> If E is an array, the result is also an array except that the last dimension is expanded in the same manner as if the array were raveled.

Conditions:

The length of the last dimension of the result is an integer multiple of the length of the last dimension of E, since the same width applies to each element.

For some element(s) there will be only one preceding blank. Other may have more than one blank.

Print precision also controls the printed numbers.

Exponential notation is used for all output if any element has either an integer part too big to be exactly expressed, or only a fractional part and the exponential notation would be shorter by 3 or more characters than the numeric notation.

Examples:

```
     ⎕PP←6                    ⍝ 6 DIGITS PRINT PRECISION
     ⎕←Y←12345                ⍝ NORMAL DISPLAY
12345
     ⍕Y                       ⍝ SCALAR IMPLICIT FORMAT
 12345
     ⍴⍕12345                  ⍝ CHARACTER VECTOR
6
     X                        ⍝ NORMAL DISPLAY
12.34  0  11  222  ¯333  0.44
     ⍕X                       ⍝ ←→ ,⍕⍉X VECTOR IMPLICIT FORMAT
   12.34    0      11      222     ¯333        0.44
     ⍕⍉X                      ⍝ COLUMN MATRIX IMPLICIT FORMAT
   12.34
    0
   11
  222
 ¯333
    0.44
     1000*¯2 ¯1 0 1 2         ⍝ E NOTATION ONLY WHERE REQUIRED
1E¯6  0.001  1  1000  1000000
     ⍕1000*¯2 ¯1 0 1 2        ⍝ E NOTATION IF ANY DOES NOT FIT
1E¯6  1E¯3  1E0  1E3  1E6
     A                        ⍝ ARRAY NORMAL DISPLAY
  12.34     0       22
¯333        0.44    ¯0.5
     ⍕A                       ⍝ ARRAY IMPLICIT FORMAT
   12.34    0      22
 ¯333       0.44   ¯0.5
     ⎕PP←2                    ⍝ 2 DIGITS PRINT PRECISION
     ⍕A                       ⍝ AFFECTS IMPLICIT FORMAT
  1.2E1   0       2.2E1
 ¯3.3E2   4.4E¯1 ¯5.0E¯1
     ⍕'APL'                   ⍝ CHARACTER IMPLICIT FORMAT
APL
     ⍴⍕'APL'                  ⍝ NO CHANGE
3
     'TEMP=',(⍕99.2),'° F'    ⍝ ONE SPACE TO LEFT, NONE TO RIGHT
TEMP= 99.2° F
```

NUMERIC
FORMAT
FUNCTION (1)
⍕

Form:

V ⍕ N   Numeric explicit format of N according to V

Where: V is numeric format vector
N is the numeric data object to be formatted

Results:

The numeric data object N is represented as a character data object. The shape of the result is the same as N, except that the last dimension is determined by the format V.

The format V must be an integer vector of length 2xM where M is a positive integer. Successive pairs of elements from V specify how successive planes across the last dimension of N are to be formatted.

If W is the first and D is the second member of a pair, all elements of the corresponding plane across the last dimension of N are formatted in a field W wide with D decimal places. The character format equivalent is also shown.

D > 0 ↔	*F* W . D	fixed point
D = 0 ↔	*I* W	integer
D < 0 ↔	*E* W . D	exponential

If M is less than the length of the last dimension of N, then the format V is cyclically reused.

Conditions:

A field width inadequate to allow representation of the number is filled with '*'.

In fixed point representation this fill occurs if the integer part requires more than W-D+2 digits.

The exponential result is left justified with leftmost column for negative mantissa sign (otherwise blank). The next column is the mantissa integer part N, 1≤N<10, then the decimal point, then D decimal part digits. Next is *E* , then exponent negative sign only if needed, then finally exponent (one digit if sufficient). Thus, W must be at least D+4 and may need to be as much as D+6 to allow representation.

Examples:

```
     10 3⍕0 123 0.0125 ¯1234.5678  ⍝ VECTOR, FORMAT CYCLIC
     0.000   123.000     0.012 ¯1234.568
     10 3⍕⍉0 123 0.0125 ¯1234.5678  ⍝ ARRAY
     0.000
   123.000
     0.012
 ¯1234.568
     5 0 5 0 8 4 12 ¯3⍕0 123 0.0125 ¯12345.678  ⍝ VECTOR
    0  123  0.0125¯1.235E4
     5 0 8 3⍕100 200∘.+10 20∘.+1 2 3  ⍝ ARRAY
  111 112.000  113
  121 122.000  123

  211 212.000  213
  221 222.000  223
     5 0 5 2⍕ 3 5.12 8 27.3456 ¯5
    3 5.12    827.35    ¯5
     7 ¯1⍕⍉¯53.8 ¯0.0000345 0 12345678 2.35E10 4.0E¯15 0.25
¯5.4E1
¯3.4E¯5
 0.0E0
 1.2E7
 2.4E10
*******
 2.5E¯1
     7 1 ⍕ ⍉¯53.8 ¯0.0000345 0 12345678 2.35E10 4.0E¯15 0.25
  ¯53.8
    0.0
    0.0
*******
*******
    0.0
    0.3
```

CHARACTER
FORMAT
FUNCTION (1)
⍕

Forms:

C ⍕ E	Character explicit format
C ⍕ (L)	Character explicit formatted list

Where: C is a character string specifying the format
E is a data object of rank at most 2
L is a list of components separated by semicolons.

Results:

The result is character data matrix representing the right argument or list components according to the format specification. The number of rows in the result is the maximum of the number of rows in the matrices that comprise the right argument. If only scalars or vectors appear in the right argument, then a matrix with one row results.

A scalar component is replicated in all rows. Each element along the last dimension of a shaped component is formatted according to the corresponding format phrase.

Conditions:

Each list component is either E or null. There is no type or shape conformability requirement between list components.

A character matrix is created of appropriate shape filled with blanks. Then, non-blank characters are inserted according to the format string applied to corresponding portions of the right argument. Separate format interpretation occurs for each row in increasing order. Only the top fields in the result have values for any matrices with less rows than the maximum.

A null list component may be used to allow replication in all rows of the format specifier.

Character Format Syntax Chart:

The format character string C has many options. It should conform to the following syntax. The leftmost entry is the syntactic unit being defined in terms of one of the alternatives, if any, to the right of 'is'. Upper or lower case letters in this type font represent syntactic units further defined. Letters or characters in the APL font represent themselves. 'text' represents any APL string excluding '>'. Blanks are ignored except within 'text'. Character representations of integers are used for r, M, W and D.

f	is	s or s;s; . . . ;s	format
s	is	g or g,g, . . . ,g or empty	segment
g	is	c or r(c)	group
r	is	optional clause replicator, default is infinite	replicator
c	is	p or p,p, . . . ,p	clause
p	is	one of:	phrase

M J *A* W	character object formatting
M J *E* W.D	floating point numeric formatting
M L Q *F* W.D R	fixed point numeric formatting
M L Q *I* W R	integer numeric formatting
M *X* W	skip W characters forward, M times
M *T* N	tab to N characters from start of format; (may be used to back up for replacement)
<text>	literal text for each row;

M	is	optional phrase replicator default is 1	phrase replicator
W	is	total columns for field	field width
D	is	optional number of places to right of decimal point, default is 0	decimal places
L	is	B or C or B C or empty	left decorator
B	is	*<text>	background for fielc
R	is	C or empty	right decorator
C	is	S<text> or S<text> S<text> or S<text> S<text> S<text>	conditional text
S	is	one or more of:	sign selector

-	insert 'text' in field if negative
o	insert 'text' in field if zero
+	insert 'text' in field if positive

J	is	*L* or empty, default is right justify in field	justifier left
Q	is	zero or more of:	qualifier

L	left justify in field
B	skip if zero
C	insert commas
Z	leading zero insert

N	is	columns to right of start of format	next column

The prior syntax chart provides named syntactic elements for semantic description only. The terminal forms as used in Q (shown in APL font) are the same as in the syntax diagram.

In general, a right argument data object is treated as a matrix. A vector or scalar is treated as a matrix with only one row.

The form using a parenthesized list containing component data objects separated by semicolons imposes no conformability or type restriction on adjacent components. The formatted result will have as many rows as there are in the data object having the most rows. The corresponding fields for objects with less rows will be blank. Each semicolon represents a synchronizing point with a semicolon in the corresponding format.

Each format segment applies in order to the corresponding data list component. The format segments are cyclically reused if necessary, until the entire data list has been formatted. If the format segment is empty, default formatting is used to format that data object.

Each format group applies in order to the corresponding columns of any one data list member. The format group is cyclically reused if necessary, until all columns of the data list member are formatted.

Within the format group an integer clause replicator can be used to limit replication. Without the replicator the clause is assumed to replicate cyclically as often as necessary.

A format clause is a series of phrases separated by commas.

Each phrase specifies the field width, and the content for that field resulting from either conversion of a data object or a literal text.

A The character object formatting phrase permits expansion between the columns of the object if *W* is greater than 1. It can be explicitly justified left, or right by default.

E The floating point numeric formatting phrase provides results in scientific notation: mantissa E exponent, e.g., ¯3.2*E*¯2 or 9.73*E*21. Default columns for non-negative signs are elided. This format can be explicitly justified left, or right by default.

F The fixed point numeric formatting phrase provides fixed, aligned format with a specified number of decimal places. This phrase permits qualifiers and left or right decorators.

I The integer numeric formatting phrase provides integer results with qualifiers and left or right decorators.

Any numeric formatting phrase for which the field width is too small gives '*' replicated for the entire field in the row in which the data element was out of range.

X The skip formatting phrase provides rightward skip over the indicated number of columns. The replicator is not needed. Instead, using the default replicator of 1, the width can be the product of replicator times width. The columns are skipped, not blanked, to allow any prior content to remain.

T The tab formatting phrase allows absolute repositioning to any result column starting from the leftmost as column 0. Any subsequent formatting phrase will overwrite any prior contents.

A <text> phrase unconditionally includes the text string in every row of the result. The text cannot contain the '>' character.

R The integer phrase replicator specifies the number of uses of the phrase before moving to the next phrase in the clause.

W The total field width for character or numeric phrase formatting should include sufficient columns for the entire anticipated result range of values including signs and decorations.

D The decimal places for fixed point and floating point numeric formatting permit specified precision result. Rounding occurs as part of formatting.

Left and/or right decorators apply to fixed point or integer formatting.

- o + The sign selectors alter the result depending on the sign of each individual data element. These prefixes to explicit text can be applied separately, or in combinations. At most one of each sign selector should occur on each side of a formatting phrase. The same sign selector may appear in the left and right decorators. A '-' selector removes the negative sign from any negative element.

*<text> A field background can be specified. The text, replicated if necessary, is initially placed in the field, then partially replaced.

L The default justification of phrases that do not require the specified width is to the right. Unless background is specified, excess columns to the left are blanked. Left justification can be explicitly specified instead, blanking excess columns to the right.

L B C Z qualifiers alter the field content for integer and fixed point formatting. They include left justification; blanking (the numeric result) if the element value is zero; insertion of commas to set off positive powers of 1000 for large numeric results; and insertion of leading zeros to fill the field.

Character Vector Formatting Examples:

Numeric data objects

```
      ⎕←NV←¯1230 4.55 0 ¯0.765 60.525
¯1230  4.55  0  ¯0.765  60.525
      ⎕←NM←¯0.05 25∘.×410 1 0.025
¯2.050E1   ¯5.000E¯2   ¯1.250E¯3
 1.025E4    2.500E1     6∘250E¯1
```

Floating Point

```
      'E10.2'⍕NV
¯1.23E3     4.55E0      0.00E0  ¯7.65E¯1    6.05E1
      'E10.4,E6.0,E10.2'⍕NM
¯2.0500E1 ¯5.E¯2 ¯1.25E¯3
 1.0250E4   3.E1   6.253¯1
      'E6.1'⍕¯0.12 0.12
************
      'E7.1'⍕¯0.12 0.12
¯1.2E¯1 1.2E¯1
```

Fixed Point

```
      'F10.2'⍕⍉NV
  ¯1230.00
      4.55
      0.00
     ¯0.47
     60.53
      'F10.2'⍕NV
  ¯1230.00      4.55      0.00     ¯0.76     60.53
      'F7.2,F6.1,F8.4'⍕NM
 ¯20.50   0.0 ¯0.0013
*******  25.0  0.6250
```

Integer

```
      'I6'⍕NV
¯1230     5     0    ¯1    61
      'I5,I2'⍕NM
  ¯21 0    0
1025025    1
```

Phrase Replicator

```
      '2I3,2I5,3I2'⍕ 1 2 3 4 5 6 7 8 9
  1  2    3    4 5 6 7  8  9
```

Justify Left

```
      'LI5'⍕⍉¯1 2 34 567
¯1
2
34
567
```

Background

```
      '*<∘>I5'⍕¯1 0 2
∘∘∘¯1∘∘∘∘0∘∘∘∘2
      '*</|\>I5'⍕1 23 456
/|\/1/|\23/|456
```

Sign Selectors

```
      '+<P>○<Z>-<N>I5'⍕¯1 0 2
   M1   Z0   P2
      '+○< >-<(>I5+○< >-<)>'⍕⍉¯1 0 2   ⍝ (NEGATIVE)
  (1)
   0
   2
      '-<       ->○<    >LI12'⍕⍉¯1 32 0 541 ¯35
        -1
32
    0
541
        -35
```

Blank Zero Field

```
      'BI5'⍕¯1 0 5
   ¯1         5
      'BLI5'⍕⍉¯1 0 5
¯1

5
```

CHARACTER
FORMAT
FUNCTION (7)
⍕

Comma Insert

```
      'CI10'⍕1234567
 1,234,567
      'CF12.4'⍕1234.5678
  1,234.5678
```

Zero Insert Left

```
      'ZI3'⍕⍉1 23 456
001
023
456
```

Combined

```
      '*<∘>ZBI5'⍕⍉1 23 ¯456 0 987
00001
00023
¯0456
∘∘∘∘∘
00987
      'ZBCI7'⍕1 0 2345 ¯1
000,001       002,345¯00,001
      'BCI5'⍕⍉¯1 0 234 5678
   ¯1

  234
5,678
```

Character

```
      'A2'⍕2 4⍴'GOODWORK'
 G O O D
 W O R K

      'A1,A2,A3,A4'⍕'OPEN'
O P  E   N

      'LA2'⍕'LEFT'
L E F T
```

Tab and Skip

```
      'I15,T0,I5,X20,I5'⍕25 50 75
   50        25               75

      'I15,T0,I5,I25'⍕25 50 75
   50                       75
```

Text

```
      '<|>,I5,<|∘>'⍕⍉1 10 ¯25

|    1|∘
|   10|∘
|  ¯25|∘
```

Combined

```
      'I5;X4,2A1'⍕(5 6;'AB')
    5    6    AB

      'I5;X4,A2'⍕(100;⍉'AB')
  100     A
  100     B

      'I5;X4,A1'⍕(,100;⍉'AB')
  100    A
         B

      'I5;A5;F5.1'⍕(⍉1 10 100;⍉'FINE';2 3⍴1.1×1 2 3 4 5 6)
    1    F  1.1  2.2  3.3
   10    I  4.4  5.5  6.6
  100    N
         E

      'LI5,2(LI3,F7.2,X4),I3'⍕3 5 15.72 17 23.15 ¯3
3    5    15.72    17   23.15     ¯3
```

SECTION 6

SYSTEM VARIABLES, SYSTEM FUNCTIONS AND SHARED VARIABLES

GENERAL.

The system variables provided within each workspace of the APL processor specially tailor the processing to the application of that workspace.

The system functions are provided to permit the user to perform many functions that query or alter the run environment of the account or to query the total environment of the APL system.

The shared variables and the system functions that handle them permit the user to communicate with other processes concurrently running with APL/700 or with other APL users.

The classes of system functions include:

Function transformations
Name functions
Debugging aids
Execution controls
Special characters
Status inquiries
Shared variable handlers
I-bar primitive functions

SYSTEM
VARIABLES (1)
⎕CT ⎕IO ⎕PP ⎕RL

System variables always have values. They are provided in a workspace by default. They are used by the APL processor to specialize its behavior for the current needs of the user of that workspace. Only values (N) in limited domains may be assigned to these variables.

System Variable	Name/ System Command	Purpose	Suggested Default Value for new account	Domain for N
⎕CT	Comparison Tolerance *)FUZZ*	relative tolerance used in comparison with Boolean and integer domains and the primitive functions: < ≤ = ≥ > ≠ ∊ ⊂ ⊃ ∩ ∪ ⍳	1*E*¯10	0 ≤*N*< 1
⎕IO	Index Origin *)ORIGIN*	origin for ordinal counting, applies to the primitive functions: ⍳ ⍋ ⍒ ? [ ] ⍉	1	0 or 1
⎕PP	Print Precision *)DIGITS*	number of significant digits used to round and display or default format fractional or scientific notation numbers	10	integer 1 thru 12
⎕RL	Random Link *)SEED*	starting value for random number generator	131131704506	integer 0 thru ¯1+2*39

Any of these system variables may be included in the local names list of a defined function. In contrast to other identifiers in the local names list, the global value of a system variable is retained within the function until first an assignment is made to that local instance of it. This permits the function to remain sensitive to the calling environment. For example, assume a result must depend on the callers origin. The global origin value can be retained in another local variable. Then the function is executed in the desired local origin to develop the desired local result. Finally the result adjusted for the global environment origin value before return to the calling function.

In a clear workspace the suggested default values for the system variables will result. These can be overridden by the user of the account with the system commands corresponding to the system variables. The system variables do not alter the defaults, and changes to the defaults only affect clear workspaces, they do not alter the values of the system variables in a non-clear workspace.

The comparison tolerance is a relative tolerance used in comparisons. It helps resolve the problem of the finite precision with which numbers are represented within the computer. In a dyadic function the comparison tolerance is relative to the left argument. For example:

```
A=B ↔ ⎕CT≥|(A-B)÷A
A<B ↔ ⎕CT≥(B-A)÷|A
```

The comparison tolerance is also used for domain checking where the domain of the function is non-continuous, e.g., integer or Boolean domain. In this case the test is:

```
(⌈(|X)×1-⎕CT)=⌊(|X)×1+⎕CT
```

The index origin affects the denumeration of elements and the dimensions in an array.

Origin	Denumeration begins with
0	0
1	1

The index origin affects the first number for ordinal numbering:

⍉	permute dimensions (dyadic left argument)
⍳	integers, index of
⍋⍒	grade up, grade down
?	roll (monadic), deal (dyadic)
[]	subscripts on arrays [bracketed]
	dimension selector [bracketed]
	laminator [bracketed]
	file component selector [bracketed]

The print precision affects the result of all numeric outputs in fractional or exponential form. No more than ⎕PP significant digits are displayed. Rounding is invoked first. Integers are displayed with full precision if their magnitude is less than 2*39. Also, print precision affects the character object result of default formatting using ⍕.

The random link affects the result of the roll and deal functions. The random link is used as the seed to the random number generator. Each time the random number generator is called, the seed provides the starting value to determine the next value(s) delivered. Each use delivers a result and changes the seed. Given the same seed and the same range, the random number generator will generate the same random numbers (and return the same new seed).

SYSTEM FUNCTIONS.

System functions allow the user to affect the run environment.

FUNCTION TRANSFORMATIONS
⎕CR ⎕VR ⎕FX

System Function	Name	Results
⎕CR N	Canonic Representation	Character matrix. N is the character string name of an unlocked defined function. If not, result has shape 0 0. Otherwise each row is a line of function N. The first row is the function header. Line numbers and opening and closing dels are omitted.
⎕VR N	Vector Representation	Character vector. N is the character string name of an unlocked defined function. If not, result is an empty vector. Otherwise each line of function N is terminated by the return character ⎕R after the last non-blank. Line numbers and opening and closing dels are omitted.
⎕FX C	Fix	Defined function. C is either a character vector or matrix in the form from the vector or canonic representation. The function name will be from the first line of C. If that name is local to the function in which the fix is executed, the fixed function is also local. If an explicit result is required, it is the name of the fixed function.

Canonic Representation of a function is useful for user-written function editing routines where line rearrangement, function merging or separation is desired. Note that the shape of the result is the number of lines (including header) by the length of the longest line. Thus, this form generally takes more space than vector representation, particularly if the line lengths differ.

The Vector Representation is usually the more compact representation, and is the preferred form for storing functions as file components.

A Fix of a character representation returns the function in unexecuted form. This form takes slightly more space than after first execution.

The defined function name resulting from a Fix must not have prior meaning. If the function name is local to some function in the calling sequence resulting from executing the Fix, then the fixed function is local to that function. The definition of the fixed function disappears upon exit from the function to which the fixed function is local.

Name system functions work with a string or matrix of names.

System Function	Name	Result
⎕NL N	Name List	Matrix of names of objects of specified kinds in the current environment. Names are alphabetized, left justified, one per row. N is a numeric scalar or vector selecting object kinds: 0 no associated meaning 1 labels 2 variables 3 functions 4 other (groups)
A ⎕NL N	Selective Name List	Like Name List, but only includes names starting with a character in the string A. A is chosen from letters, underscored letters, ∆ and ⍙.
⎕NC C	Name Classification	Vector of integers indicating name use in the current environment for corresponding name in character or matrix C. Result values: 0 no associated object 1 label 2 variable 3 function 4 other (group)
⎕EX C	Expunge	Objects corresponding to names in character vector or matrix C are expunged. The objects must not be labels, groups, or active functions. If required, the result is a Boolean vector with ones everyplace the corresponding name from C was expunged.

A character string argument to Name Classification or Expunge must contain only one name. A character matrix argument must contain one name per row.

The most local occurrence of a name in the current environment determines its kind. A more global occurrence may be shielded by an occurrence as a local name in an active function. A more global meaning (if any) is restored upon exit from the function to which the name is local.

Expunge may be used to eliminate current meanings for objects from the current environment so long as they are not names of active functions or labels. Unlike)ERASE, other local names can be expunged.

DEBUGGING AIDS (1)
⎕*ST* ⎕*SS* ⎕*SM*
⎕*RT* ⎕*RS* ⎕*RM*
⎕*QT* ⎕*QS* ⎕*QM* ⎕*MV*

The following system functions are oriented to lines of unlocked user-defined functions.

Monadic (all lines)	Name	Dyadic (specified lines)	Result
⎕*ST* F	Set Trace	N ⎕*ST* F	L
⎕*SS* F	Set Stop	N ⎕*SS* F	L
⎕*SM* F	Set Monitor	N ⎕*SM* F	L
⎕*RT* F	Reset Trace	N ⎕*RT* F	L
⎕*RS* F	Reset Stop	N ⎕*RS* F	L
⎕*RM* F	Reset Monitor	N ⎕*RM* F	L
⎕*QT* F	Query Trace		B
⎕*QS* F	Query Stop		B
⎕*QM* F	Query Monitor		B
⎕*MV* F	Monitor Values	N ⎕*MV* F	V

Where:

F is character vector name of unlocked defined function
N is numeric vector of line numbers
L is numeric vector of lines with property (set, reset) returned only if required
B is Boolean vector, 1 if property set, 0 if reset; one element per line including header
V is vector of numeric monitored values accumulated during executions since set.

The monadic forms apply to all lines including the header line 0. The dyadic forms apply only to altering the current setting for line numbers in the left argument.

During function execution, the effects are as follows on encountering a line on which one or more aids are set:

Aid	Header Line	Body Line
Trace	result returned by function	result
Stop	suspend prior to return	suspend before execution
Monitor	increment number of calls	increment CPU time in line execution

The Trace result forms are:

```
Function-Name [Line-Number]

Function-Name [Line-Number]   Type (Shape) Value
```

The first form occurs if the line has no result; otherwise, the second form occurs (including a leftmost control transfer value or assignment).

The Type is B for Boolean, C for character or N for numeric. The Shape is a numeric vector; the Value is the normal displayed value.

The Stop result form is:

```
Function-Name [Line-Number]*
```

After a Suspend on the header after function completion, the local names are still defined.

The Monitor values are internally accumulated more precisely than they are displayed. The ceiling of the accumulated number of milliseconds is displayed. A time of 0 is shown only for unmonitored lines or monitored lines that have not been executed. Thus, monitoring all lines over a period of execution is an effective way to determine if some program path has reached each line, and also the time spent in each line.

If a line contains a call on another function, any time spent in that function would be accumulated there, instead of in the calling line.

EXECUTION CONTROLS
⎕DL ⎕ED ⎕ER

Normal execution can be altered using the following system functions.

System Function	Name	Result
⎕DL D	Delay	optional actual delay D in seconds
⎕ED S	Edit	edited line after editing with normal entry of within-line editing marks '/', '.' or ' '
B *⎕ED* S	Phrase Edit	edited line after editing string S according to Boolean vector B with ones meaning phrase terminators '.'
⎕ER S	Error	simulates an error occurring at the point of execution. S is displayed as the error message.

The specified Delay amount D is an integer indicating minimum desired execution pause before resumption. The actual delay, returned if required, also includes time awaiting an APL processor once the specified delay has occurred.

Each Edit function accepts a character string as the right argument. This string may not include any of the following characters: linefeed, return, backspace, tab or null. The monadic form displays the string and returns to the left margin for entry of a line of edit characters applied to the characters above: '/' for delete, '.' for phrase end before, and spaces for no change. The next line displays the first phrase for editing. The ATTN causes entry of the next phrase, etc.

The Phrase Edit dyadic form uses the Boolean left argument (of the same length as the string) with each one indicating a phrase end. This avoids the line of entered edit characters.

The Error message is displayed, an error indication prompt is given, and execution is suspended. This is principally useful in a locked function, where the error message results in the suspension point indicator being in the line of the calling function containing the call, rather than in the line containing the *⎕ER*. The last line executed in the function is the one containing the *⎕ER*; no other explicit control transfer out of the function is required.

SPECIAL CHARACTER SETS (1)
⎕*B* ⎕*L* ⎕*R* ⎕*T*
⎕*N* ⎕*A* ⎕*D* ⎕*AV*

The single characters or character vectors below are the values returned by niladic system functions.

System Function	Name	Result
⎕*B*	Backspace	scalar backspace character
⎕*L*	Linefeed	scalar linefeed character
⎕*R*	Return	scalar carrier return character
⎕*T*	Tab	scalar tab character
⎕*N*	Null	scalar null character
⎕*A*	Alphabet	character vector '*ABC...Z*'
⎕*D*	Digits	character vector, '0123456789'
⎕*AV*	Atomic Vector	all APL characters

These characters are processed internally to APL just as any other elements of a character data object. The only special properties of the first five are associated with output processing for terminal display. Some terminals may not adequately accept these characters.

The Backspace character can be used to display overstruck output characters not in the allowed character set. It can not be used to move to the left of the start of the display line.

The Return character causes completion of an output line, just as the RETN key does for input. It includes both line feed and cursor return to the left margin.

The Linefeed character can be used for advancing the display line while the cursor is positioned into a line without return.

In cases where the cursor is at the left margin, Linefeed and Return have the same external effect.

The Tab character can be used to prepare output with irregular terminal physical tab settings. In this use, the normal APL editing to insert tabs in output for display should be disabled. The tab interval should be set to 0 by)TABS 0. The print width may be exceeded.

SPECIAL CHARACTER
SETS (2)
⎕*B* ⎕*L* ⎕*R* ⎕*T*
⎕*N* ⎕*A* ⎕*D* ⎕*AV*

The Null character takes one unit of transmission time when sent to the display, but has no visual effect on the normal static display. Its principal use is with non-standard display devices such as plotters that may require time to complete a prior command.

The alphabet and digits are often useful in text processing.

The atomic vector includes all characters defined for APL. The displayable characters are shown in table 6-1. The index position numeric location of each character is shown in the last line below each character. The hexadecimal equivalent is shown in the middle line.

The shape of the atomic vector is 256. Only the printing and special characters are shown in the table. The entries shown as ??? and the others above 175 are non-printing. Any attempt to display one of these results in the squish-quad ⌷. Since these are not displayable, their use should be carefully considered. The principal application of atomic vector is for communication with external processes through shared variables.

The left tack (77), right tack (78), diamond (133), left brace (134), right brace (135), and currency symbol (143) are not available on all terminals. Printing conventions for these are uncertain on 88 character terminals. Note that these 6 extra characters are not part of the necessary APL character set.

Table 6-1

Character Representation Order in Atomic Vector

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*
00	01	02	03	04	05	06	07	08	09	0*A*	0*B*	0*C*	0*D*	0*E*	0*F*
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*	0	1	2	3	4
10	11	12	13	14	15	16	17	18	19	1*A*	1*B*	1*C*	1*D*	1*E*	1*F*
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
5	6	7	8	9	.	+	-	×	÷	⌈	⌊	*	●	|	!
20	21	22	23	24	25	26	27	28	29	2*A*	2*B*	2*C*	2*D*	2*E*	2*F*
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
?	○	~	∧	∨	⍲	⍱	<	≤	=	≥	>	≠	⍴	,	⍳
30	31	32	33	34	35	36	37	38	39	3*A*	3*B*	3*C*	3*D*	3*E*	3*F*
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
↑	↓	⍋	⍒	/	\	⌽	⍉	∊	⊥	⊤	∪	∩	⊢	⊣	⊂
40	41	42	43	44	45	46	47	48	49	4*A*	4*B*	4*C*	4*D*	4*E*	4*F*
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
⎕*N*	⊃	⍙	∆	←	:	⎕	(	)	[	]	'	→	;	⌶	∘
50	51	52	53	54	55	56	57	58	59	5*A*	5*B*	5*C*	5*D*	5*E*	5*F*
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
∇	⍫	¯	_	*A̲*	*B̲*	*C̲*	*D̲*	*E̲*	*F̲*	*G̲*	*H̲*	*I̲*	*J̲*	*K̲*	*L̲*
60	61	62	63	64	65	66	67	68	69	6*A*	6*B*	6*C*	6*D*	6*E*	6*F*
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
M̲	*N̲*	*O̲*	*P̲*	*Q̲*	*R̲*	*S̲*	*T̲*	*U̲*	*V̲*	*W̲*	*X̲*	*Y̲*	*Z̲*	⍞	⍝
70	71	72	73	74	75	76	77	78	79	7*A*	7*B*	7*C*	7*D*	7*E*	7*F*
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
α	ω	¨	*$*	⌹	◊	{	}	⎕*B*	⎕*L*	⎕*R*	*???*	*???*	*???*	⎕*T*	$
80	81	82	83	84	85	86	87	88	89	8*A*	8*B*	8*C*	8*D*	8*E*	8*F*
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
⍎	⍕	⌿	⍀	⊖	⌼	⎕̴	⍓	⍌	⍃	⌸	⍄	⍯	⍐	⍗	⍁
90	91	92	93	94	95	96	97	98	99	9*A*	9*B*	9*C*	9*D*	9*E*	9*F*
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
⍂	⍍	⍇	⍈	⌻	⍔	*???*	*???*	*???*	*???*	*???*	*???*	*???*	*???*	*???*	*???*
*A*0	*A*1	*A*2	*A*3	*A*4	*A*5	*A*6	*A*7	*A*8	*A*9	*AA*	*AB*	*AC*	*AD*	*AE*	*AF*
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175

STATUS INQUIRIES
⎕PT ⎕PW ⎕WI ⎕AN ⎕AI
⎕NEWS ⎕LC ⎕TS ⎕UL
⎕WA ⎕NA ⎕LA ⎕FA ⎕SA

Status inquiries are niladic, value returning system functions:

System Function	Name	Result	Remarks
⎕PT	Print Tabs	uniform physical tab interval assumed for terminal	set by)TABS n
⎕PW	Print Width	maximum characters/display line	set by)WIDTH n
⎕WI	Workspace ID	character vector: identifier	)WSID
⎕AN	Account Name	character vector: identifier	⌶29 ↔ ⎕AN
⎕AI	Accounting Information	computer time, connect time this session	in milliseconds
⎕NEWS	News	system news sign-on message	
⎕LC	Line Count	numeric vector: includes line on which line count occurs, then other line numbers of functions in state indicator	⌶27 ↔ ⎕LC ⌶26 ↔ (⍳0)⍴⎕LC
⎕TS	Time Stamp	numeric vector: year, month, day, hour, minute, second, millisecond	Example 1974 12 31 23 59 59 999
⎕UL	User Load	number of user accounts on APL	⌶23 ↔ ⎕UL
⎕WA	Working Availability	bytes remaining, bytes in use in workspace	⌶22 ↔ 1↑⎕WA
⎕NA	Name Availability	slots remaining, slots assigned in symbol table	
⎕LA	Library Availability	workspace slots remaining, workspaces in)LIB	
⎕FA	File Availability	file slots remaining, files in)FILES	
⎕SA	Shares Availability	shared variable slots remaining, in use	

Use of the above status inquiries is preferred to the redundant I-bar primitives. The sum reductions of the last two area inquiries provide the quotas established by the installation for the account. The number of symbols in the name table is +/⎕NA, set by)SYMS n for the clear workspace default, or)CLEAR n for a particular workspace. Space in a workspace is measured in bytes. See Appendix B.

SHARED VARIABLES.

A shared variable permits coordinated data exchange between the user process and one other partner process external to it. A process is either an active workspace of an APL user or an APL shared variable utility. APL user processes are referred to by their account names. APL utility processes have account names that are character representations of integers from 1 to 999.

Sharing means that either process can use or set the shared variable value. Sharing is bilateral; no more than two processes can share a variable at one time. Neither process is dominant.

A shared variable has a name used internal to the workspace. It also has an external name, or surrogate, used in common by sharing processes. The surrogate may be the same as the name, in which case, only the name is needed. Several shared variables may be in use at one time. The same surrogate may be used with more than one internal name, each shared with possibly different processes. An internal name of a shared variable may have only one surrogate associated with it.

Use of a shared variable is initiated by this typical sequence:

Process A	Process B
tenders an offer to share	accepts the offer

Thereafter either process can access the variable being shared. The degree of coupling is the number of processes that currently agree to share a particular variable, as viewed by ones own process:

0 if the name is currently not in use as a shared variable
1 if an offer has been made but not been accepted; or after sharing, an offer is retracted by the other process
2 if an offer has been made and accepted

When the degree of coupling is 2, either process may access the common value. Access includes both setting (assigning a value to) and using (once assigned, then referencing the present value of) the variable.

The coordination of data exchange between the two processes is based on a Boolean access control matrix (ACM), whose elements control the allowable sequence of accesses. Each shared variable has an ACM.

The access control matrix (ACM) has shape 2 2 and has Boolean elements:

1 access is constrained
0 access is not so constrained

In summary form, ACM elements have meaning:

Set A	Set B
Use A	Use B

Where: A represents one's own process
B represents the sharing partner process.

In more detail:

ACM Element			Constraint if value is 1
		two successive	requires intervening
Set A	1 1↑ *ACM*	sets by A	access by B
Set B	1 ¯1↑ *ACM*	sets by B	access by A
Use A	¯1 1↑ *ACM*	uses by A	set by B
Use B	¯1 ¯1↑ *ACM*	uses by B	set by A

Note the symmetry of the above. For elements with value 1 in:

Top row - Two successive sets by one process requires an intervening access by the other. This may be used to assure that the second process has an opportunity to accept the value set by the first.

Bottom row - Two successive uses by one process requires an intervening set by the other. This may be used to assure that (at least one) new value has been set prior to use.

First column - Individual controls on one's own process setting and use.

Last column - Individual controls on partner's process setting and use.

If a constraint is 1 and the required intervening event by the second process has not occurred, the first process is delayed.

Each process sees the access control matrix with one's own process as the first column and the partner process as the second column.

The four Boolean element access control vector (ACV) used to restrict the ACM is established from one's own process as 2 2ρOWNACV and the effect of the setting by the partner process as viewed by one's own process is ⌽2 2ρ PARTNERACV.

The resulting ACM↔(2 2ρ OWNACV)∨⌽ 2 2ρ PARTNERACV describes the total restriction imposed by both processes. The defaults are 0 for OWNACV, OTHERACV and hence ACM. Thus unrestricted access is the default. Restrictions must be explicitly established. One partner can only increase restrictions set by the other. Upon retraction by one partner, the explicit access controls set by the other remain.

A set of surrogate lists is maintained between two particular processes. Each such list has the record of offers to share using one particular surrogate. A surrogate list is ordered in time of offering to share a variabale using that surrogate. Acceptance of an offer initiates sharing with the oldest outstanding offer. Termination of sharing occurs when one partner retracts the offer. Then the other partner still has a valid offer and will commence sharing with the oldest remaining offer having the same surrogate (if any).

An offer to share a variable can be made explicitly to another process, or can be made general, to any process that may desire sharing. The first capability permits inter-process communication, typically between APL users. Queries are provided to determine if any processes have explicit sharing requests outstanding to the querier, and also what the surrogate names are. No queries are provided for general offers. They are typically used by utilities ready to accept an offer when made.

The shared variable does not provide additional space to the user beyond that in the active workspace. There must be sufficient space to use whatever size object the partner sets. The workspace contains the data object that was most recently used or set by the user. Using a value set by the partner changes the value in the workspace.

A workspace may be saved while a shared variable is offered or accepted. If there had been no value assigned to that variable, the name only will be saved as a name without meaning. If a value had been assigned when saved, the last value either set or used by the user will be saved as a non-shared variable. Loading or copying does not reinitiate the shared variable.

SHARED VARIABLE FUNCTIONS.

There is provided a family of functions for handling shared variables. These include:

shared variable offer and degree of coupling
shared variable access controls query and augment
shared variable offers query and retract

SV OFFER,
COUPLING (1)
⎕*SVO*

Forms:

⎕*SVO* N	Determine degree of coupling of N
P ⎕*SVO* N	Offer N to P

Where: N is a character vector or matrix. Each row contains a name possibly followed by a surrogate separated by at least one space.

P is a vector (if N is a vector) or a matrix with as many rows as N. Each row contains either the specific name of an external process (an APL account or external process name) with which sharing is desired, or an empty vector or row of blanks indicating a general offer to share with any process.

Actions/Results:

Coupling: The current degree of coupling of the name or names in N is returned as viewed by the own process. Each element of the vector result in corresponding order as N may be:

0 if not currently offered as a shared variable
1 if offered by own process but not accepted
2 if both offered and accepted

Offer: Each offer by a different process of a shared variable increases the degree of coupling of that variable by one up to a maximum of 2. If an offer is made to a specific process, only that process can accept it. If a general offer is made, any process can accept it by an explicit offer for that name.

An offer made by another process for a shared variable already having degree of coupling 1, binds that variable to the two processes involved (and makes the general offer, if any, specific) so long as the degree of coupling remains 2. Once a general offer is accepted, it becomes and remains specific even if the acceptor retracts the share.

The returned result, if required, is the attained degree of coupling.

Conditions:

An attempt to make a second offer of the same name is ignored and returns the present degree of coupling if required.

Examples:

Time sequence is downward for both columns in parallel. Entries on the same line could occur in either order.

```
     ⍝ PROCESS CLF                  ⍝ PROCESS TSG

     ⎕←'TSG'⎕SVO'Y Z'
1
     ⎕SVO'Y'
1
     ⎕SVO'Z'
0                                   ⎕←'CLF'⎕SVO'A Z'
                               2
                                    A←7700
     Y                              A
7700                           7700
     Y←'HI TSG'
     Y
HI TSG                              A
                               HI TSG
                                    ⎕←''⎕SVO'B'
                               1
     BB←'CABBAGE'                   B←1 2 3
     B←'PATCH'
     ⎕SVO'BB B'
2                                   B
     BB                        1  2  3
1  2  3                             ⎕SVO'B'
     B                         2
PATCH                               C←32
                                    ⎕←'CLF'⎕SVO'C'
     ⎕SVO'C'                   1
0
     ⎕←''⎕SVO'C'
2
     C
*** VALUE ERROR ***
     ∨                              C
     C                         32
     C←'HELLO'
     ⎕←''⎕SVO'C'                    C
2                              HELLO
```

SV ACCESS CONTROLS (1) ⎕SVC

Forms:

⎕*SVC* N	Query access controls for N
C ⎕*SVC* N	Augment access controls for N by C

Where: N is a character vector or matrix. Each row contains one name and is possibly followed by a surrogate separated by at least one space.

C is a Boolean access control vector or matrix with a row of 4 elements for each row of N.

Actions/Results:

Query Controls: For each row of N, the current access control vector is returned.

Augment Controls: For each row of N, the corresponding row of C is used to augment the access control matrix for that variable:

```
ACM← (2 2ρCO)∨⌽2 2ρCP
```

Where: CO is the control vector specified by own process
CP is the control vector specified by partner process

The effect by any one process on the access control matrix of a shared variable is to only alter elements not restricted by the partner (since the 'or' function on C by one process can not remove any restriction already placed by the other process).

Note the symmetry in specifying or querying ACM. For each process, the first column refers to the controls applied to it; the second refers to the controls applied to the sharing partner process. The total access control vectors can be determined:

For own process	*CO*←, *ACM*
For partner process	*CP*←, ⌽*ACM*

If an explicit result is required, it is the resulting access control vector; or the matrix of the access control vectors as rows.

Conditions:

If N is a scalar, it is coerced to a one element vector.

C is coerced to the necessary shape if it is a Boolean single, or 4 element Boolean vector:

```
C←4⍴C              if N is a vector
C←((1↑⍴N),4)⍴C     if N is a matrix
```

When an offer to share a variable is initially made, the access control matrix is all zeros.

When a prior offer to share is withdrawn, the access control matrix returns to only those restrictions established by the remaining process still offering to share.

Examples:

Time sequence is downward for both columns in parallel. Entries on the same line could occur in either order.

```
      ⍝ PROCESS CLF                       ⍝ PROCESS TSG

      ⎕←'TSG'⎕SVO'X'
1
      ⎕SVC'X'
0  0  0  0
      ⎕SVO'X'
1                                         ⎕←'CLF'⎕SVO'X'
                                    2
                                          ⎕SVC'X'
                                    0  0  0  0
      ⎕←1 0 1 0⎕SVC'X'
1  0  1  0                                ⎕SVC'X'
                                    0  1  0  1
                                          ⎕←1 1 0 0⎕SVC'X'
      ⎕SVC'X'                       1  1  0  1
1  1  1  0
      ⎕←0⎕SVC'X'                          ⎕←'CLF'⎕SVO'A B'
1  1  0  0                          1
                                          ⎕←1 0 0 1⎕SVC'A'
      ⎕←''⎕SVO'C B'                 1  0  0  1
2
      ⎕SVC⍉'CX'
0  1  1  0
1  1  0  0
```

SV QUERY,
RETRACT (1)
⎕*SVQ* ⎕*SVR*

Forms:

⎕*SVQ* P	Shared variable query about offers P
⎕*SVR* N	Shared variable retract offer for N

Where: P is a character vector, either empty, or containing an external process name

N is a character vector or matrix. Each row contains a name possibly followed by a surrogate separated by at least one space

Actions/Results:

Query: If P is empty, it returns a matrix of processor names having unaccepted specific offers to the inquiring process. The names are left justified in a six character row with trailing blanks.

If P is the name of a process, it returns a character matrix of the surrogates for names of variables being offered for sharing by that process specific to the querying process, but not yet accepted. There is no means to query general offers.

Retract: The result if required is the degree of coupling existing prior to the retraction.

A previously made offer to share names in N is retracted and the degree of coupling reduced to 0 by the retractor and reduced by 1 for the partner (but not below 0).

A retract with prior degree of coupling =2 terminates sharing. Any access control restrictions from the retracting process are relaxed on that shared variable.

There is no effect on the sharing partner's contribution to restricting the access control matrix.

If the sharing had resulted from acceptance of a general offer, retraction by the acceptor does not restore the general offer, but leaves it as a specific offer to that acceptor.

Erasing or expunging a shared variable retracts the share.

Examples:

Time sequence is downward for both columns in parallel. Entries on the same line could occur in either order.

```
      ⍝ PROCESS CLF                       ⍝ PROCESS TSG

      ⎕←'TSG'⎕SVO'X Y'
1                                         ⎕SVQ''
      X←5                           CLF
      ⎕←1 0 0 0⎕SVC'X'                    ⎕SVQ'CLF'
1  0  0  0                          Y
                                          ⎕←'CLF'⎕SVO'Z Y'
                                    2
                                          Z
                                    5
                                          ⎕SVC'Z'
                                    0  1  0  0
      ⎕←⎕SVR'X'
2                                         ⎕SVC'Z'
      X                             0  0  0  0
5                                         ⎕SVQ''
      ⎕SVQ''
                                          Z
TSG                                 5
      ⎕SVQ'TSG'                           ⎕SVO'Z'
Y                                   1
      ⎕←''⎕SVO'A B'
1                                         ⎕SVQ''
      ⎕←⎕SVQ''
TSG
```

I-BAR FUNCTION
⌶

The primitive monadic function defined in early APL systems for querying the environment has the form:

⌶ N I-bar primitive selected by N

Where: N is an integer between 20 and 29, excluding 28.

This primitive is included but is redundant, having been replaced by the system functions. Since it may exist in old APL programs, it is described here. Deimplementation is expected in some future release.

Time units below are sixtieths of a second for I-bar results. Note that replacements naturally have different units (hours, minutes, seconds, milliseconds; or milliseconds). Conversion to the earlier (sixtieth second) time base causes the bulk of the computation below. Some results are vector instead of scalar.

Primitive	Result	Approximate Replacement
⌶20	scalar time of day	⌊0.06×0 60 60 1000⊥3↓⎕*TS*
⌶21	scalar CPU time used this session	0.06×1↑⎕*AI*
⌶22	scalar bytes remaining unused in the workspace	1↑⎕*WA*
⌶23	scalar number of users currently signed on	⎕*UL*
⌶24	scalar time of day at start of the work session	⌊0.06×(0 60 60 1000⊥3↓⎕*TS*)-1↓⎕*AI*
⌶25	scalar date in form MMDDYY where M,D,Y are digits representing month, day, and year respectively	100⊥100|1⌽3↑⎕*TS*
⌶26	scalar first element of ⌶27	1↑⎕*LC*
⌶27	vector of line numbers in state indicator: first element is line being executed, or the one last suspended; the next element is the line which called the first, or the prior suspension, etc.	⎕*LC*
⌶29	character vector containing 6 character left justified user account identification	⎕*AN*

There is no I-bar 28 (meaning terminal type on some other APL implementations). The terminal type is implicit in the line to which the terminal is connected.

SECTION 7

FILE SYSTEM FUNCTIONS

GENERAL.

The APL/700 System includes a filing system and a set of file functions that together provide a user with effective and convenient means to retain and access APL data objects outside the workspace. Defined functions can be represented as data objects and subsequently can be fixed back into the functions. Thus, a user can work with more data or functions than will fit in a workspace at one time.

FILE NAME.

Each file has a name unique among the file names of the account.

File Name is (Acct) Name [Password]

where File Name and optional Password are strings of 1 to 12 alphanumeric characters starting with a letter.

The optional Acct is the account name required if the file is owned by another account. The Acct is a string of 1 to 6 alphanumeric characters.

FILE COMPONENTS.

At any time a file has a number of components. These are numbered starting with the index origin. Any component may be null, or may contain a value. A component can contain any APL data object created in a workspace and subsequently assigned to the file component. Each component is independent, and can have any type, rank or size. In particular, some components can be user created directories to the file. A null component is one that has no value (this is different from containing an empty array as a value).

FILE LIMITS.

Any file has a maximum of 1000 component slots. The installation allocates to an account a maximum number of files, which can be determined as +/⎕*FA*. Also there is a maximum number of bytes per file which can be determined as 2⌼ File Name. There are system-imposed maximum numbers of files that can be concurrently opened by any one user (12), or by all accounts ⌼4, and number of accounts concurrently using files ⌼3.

FILE OPENING, ACTIVE AND INACTIVE STATUS.

A file may be open in one or more accounts. A file has active status if any account has the file open; otherwise, the file is inactive.

A file is opened for an account when first any file operation is executed other than create, rename, destroy, or file status test. A file remains open until either explicit release, or account sign-off.

FILE INTEGRITY.

File integrity is automatically maintained by retaining a master file and an up-date file so long as a file is active. All transactions that alter the file components are made to the up-date file. All file component reads are from the most recent value. When there are no active users of the file, it becomes inactive and any up-date file is merged with the old master file into a new master file. Any user attempt to access the file is deferred during this period when the APL file system is closing the file. Only after the closing is complete are the old up-date and master files destroyed. Thus the file will not be partially updated.

If the user expects several accounts to concurrently access the file, provision is made for any account to .hold it for exclusive use during an update. Any transaction entries while the user has the file held are provisional. They become part of the up-date file only when a file free is executed by that account, or any return to execution mode, or terminal disconnect. Any return to execution mode before the free occurs removes the provisional transaction. This capability protects the file from being partially updated.

File updating integrity over interruption or system failure is achieved by assuring that an undisturbed backup is available until any updating is complete.

All file functions that do not explicitly return a value implicitly return the file name if required. This permits a sequence of file operations to be executed in the same line of a defined function. Thus, even user interruption using a single ATTN (for which the line is completed) can have an update transaction completed in a single line. Of course, a user-entered double ATTN can violate this integrity.

FILE SYSTEM PRIMITIVE FUNCTIONS.

A group of file functions is provided for file management. Each is denoted by overstriking the quad (box) symbol with another symbol. The resulting file function has generally similar meaning to the APL primitive function using the same second symbol.

Many of the file functions have both monadic and dyadic forms. The right argument of each is the File Name, symbolically represented as 'F'.

FILE CREATE, CHANGE PASSWORD, RENAME, DESTROY
⍍ ⍔

Forms:

```
     ⍍ F        Create file F
     ⍍ F[O/N]   Change password on file F
   N ⍍ F        Rename file F to become N
     ⍔ F        Destroy file F
```

Where:

F is own account File Name, may include password
N is new File Name for file of own account
O is old password for file F, empty if none previously
P is new password for file F, empty if none desired

Actions/Results:

The File Name F is returned if required.

Create: A file F is created with no components.

Change Password: New password P replaces old password O of existing file F. Variants include:

> add password if O is empty,
> change password if both O and P are not empty,
> delete password if P is empty.

Rename File: The file F is renamed to become N.

Destroy: The file F owned by this account is destroyed.

Conditions:

Create: The file Name must not already exist.

Change Password: This can only be done by the file owner when the file is inactive.

Rename File: A file can only be renamed if inactive.

Destroy: The File Name (including lock if any) of a file owned by this account and not currently held by any other user must be provided. No file of another account can be destroyed.

Examples:

```
⍍'NEWFILENAME'
⍍'LOCKEDFILE[KEY]'
⍍'NEWFILENAME[/KEY1]'
'CHANGENAME[NEWLOCK]'⍍'NEWFILENAME[KEY1]'
⍔'LOCKEDFILE[KEY]'
```

FILE COMPONENT NULL, WRITE, READ

⍈ ⍇

Forms:

⍈[K] F	Null component K of file F
A ⍈[K] F	Write A to component K of file F
⍇[K] F	Read component K of file F

Where:
F is File Name
K is component number
A is any APL data object

Actions/Results:

Null: Destroy any prior content of component K. If required, return the file name.

Write: Replace prior value of component K by value A, or append to end of F is 1 + largest component number. If required, return the file name.

Read: Return the non-null value of component K.

Conditions:

Null: K must be an existing component number.

Write: K must be either an existing component number or 1 + the largest component number.

Read: The component must be non-null.

Examples:

```
      ⍈[3]'FILENAME'
      2 5⍈[2]'FILENAME'
      ⎕←'SMITH'⍈[3]'FILENAME'
FILENAME
      ⍇[2]'FILENAME'
2 5
      ⍇[3]'FILENAME'
SMITH
```

FILE COMPONENT
FIRST IN, OUT;
LAST IN, OUT
⍃ ⍄

<u>Forms</u>:

⍄ F	Read and pop first component out of file F
⍃ F	Read and pop last component out of file F
A ⍃ F	Append component before components already in file F
A ⍄ F	Append component after components already in file F

<u>Where</u>: F is File Name
A is any APL data object

<u>Actions/Results</u>:

The file components may be treated as a stack or a queue. The component at either end may be read and removed (out). A component may be appended to either end (in).

Out: The result returned is the indicated first (last) component. That component must be non-null. That component is taken out of (popped from) the file. If first, the component numbers of the old components are decreased by 1.

In: The data object is put in the file (pushed into). It is appended before (after) the existing file components. If before, the component numbers of the old components are increased by 1. If required, the File Name is returned.

<u>Examples</u>:

```
      'JONES'⍄ 'PERSONS'
      'SMITH'⍄'PERSONS'
      (2 2 ρ 1 1 4 7) ⍃'FILENAME'
      ⍃ 'PERSONS'
JONES
      ⍃'FILENAME'
1 1
4 7
```

FILE COMPONENT
ORDER REVERSE,
ROTATE
⊖

Forms:

⊖ F	Reverse component order in file F
I ⊖ F	Rotate circularly the components in file F

Where: F is File Name
I is integer

Actions/Results:

If required, the file name is returned.

Reverse: The component order of file F is reversed; i.e., the first changes with the last, the second changes with the second last, etc. Reverse is analogous to the primitive reverse function on a vector. If required, the File Name is returned.

Rotate: The components of file F are rotated circularly by an amount I. File rotate is analogous to the primitive rotate function on a vector. If I is negative, this is effectively a right rotate. If required, the file name is returned.

Conditions:

Rotate: I is effectively the (number of components) residue of I. I=1 causes the first component to become the last, the second component to become the first, etc.

Examples:

```
⊖'FILENAME'
2⊖'FILE[LOCK]'
¯3⊖'FILENAME'
```

FILE
COMPONENTS
TAKE, DROP
⍐ ⍗

Forms:

I ⍐ F	Take I components from file F
I ⍗ F	Drop I components from file F

Where: F is File Name
I is integer magnitude in ⍳1000
I>0 applies to components from start of file
I<0 applies to components from end of file

Actions/Results:

These are similar to the primitive take and drop functions in the components chosen. However, they are destructive of components dropped or not taken.

Take: The resulting file F has I components. If required, the file name is returned.

Drop: The resulting file F has I components dropped. If required, the file name is returned.

Conditions/Options:

Take: If the magnitude of I exceeds the number of components previously in the file, sufficient null components are appended to the file at the appropriate end:

before if I<0
after if I>0

Drop: A minimum of 0 components remain.

Examples:

```
5⍐'FILENAME'
¯23⍐'FILENAME'
2⍗'FILENAME'
```

FILE COMPONENTS
COMPRESS, EXPAND
⍁ ⍂

Forms:

```
B ⍁ F        Compress components from file F where B is 0
B ⍂ F        Expand components of file F where B is 0
```

```
  Where:     F is File Name
             B is Boolean vector
```

Actions/Results:

The ordered set of file components can be compressed or expanded. These file functions are similar to the primitive expand and compress functions.

Compress: The result is a new component set selected in order from the components previously in F, wherever a 1 exists in the Boolean B. The components of the original file are destroyed wherever a 0 exists in B. If required, the file name is returned.

Expand: The result is an expanded, ordered component set preserving the order of the original components within which null components are inserted wherever zeros exist in Boolean B. If required, the file name is returned.

Conditions:

Compress: The length of B must be the same as the number of components in the original file F: (⍴*B*)=3⌻*F*.

Expand: The number of ones in B must be the same as the number of components in the original file F: (+/*B*)=3⌻*F*.

Examples:

```
1 1 0 1 ⍁'FILENAME'
1 0 1 0 1 ⍂'FILENAME'
```

Forms:

⍓ F	Hold file F for exclusive use
⍌ F	Free own hold on file F
⍗ F	Release own use of file F

Where: F is File Name

Actions/Results:

If required, the file name if returned.

In file use shared among several accounts, exclusive use can be achieved for critical up-dates.

Hold: If the file is not currently being held (even if it is active), a hold is placed on the file which prevents any other account from accessing it. If already held by another account, hold causes a wait until freed by that account. If required, the file name is returned.

Free: A held file is freed from exclusive use. If required, the file name is returned.

Release: The account's active use of file F ceases.

Conditions:

Hold: A hold only persists while execution continues in a defined function (including input requests) or single entry from execution mode. Any return to execution mode (or file destroy while held) breaks the hold.

Free: The actual file up-dates to a held file take place provisionally into the up-date file. They are accepted as up-dates to that file all at once when the free occurs. Any interruption before the free voids the provisional entries.

Release: When no users have a file active, and a up-date file exists, it is merged with the master file. During this period when the file is being closed by the system, it is unavailable to any user. A file is also released by any sign-off or involuntary termination.

Examples:

```
⍓'(OTHER)FILE'
⍌'(OTHER)FILE'
⍗'FILENAME[KEY]'
```

FILE COMPONENT EXISTENCE
⌸ ⍯

Form:

⌸ F Map of non-null components of file F
⍯ F Map of null components of file F

Where: F is File Name

Actions/Results:

The results are Boolean vectors with length the number of components.

Non null: In component order, each resulting element is 0 if the corresponding component is null; 1 if the corresponding component is non-null.

Null: The result is the not (logical negation) of the non-null map: 1 if the corresponding component is null; 0 if the component is non- null.

Example:

```
        ⍍'FILE'
        1 2 3 4 ⍄'FILE'
        3 ⍐'FILE'
        ⌸'FILE'
  1  0  0
        ⍯'FILE'
  0  1  1
```

FILE SYSTEM INTERROGATE, STATUS, QUERY ⌼

Forms:

⌼ I	Interrogate file system
⌼ F	Test status of file F
I ⌼ F	Query attribute of file F

Where: F is File Name
I is integer single

Actions/Results:

Interrogate: Usage properties across the file system can be determined for each valid value of I:

1 current number of accounts using files
2 current total number of files that are active
3 maximum number of accounts using files
4 maximum number of active files

Status: The availability status of file F is returned:

0 file F does not exist in this account
1 file exists and is not active
2 file is active
4 file is unavailable
5 file is held by some account

Query: The result for each valid value of I is:

1 current size of file in bytes
2 maximum size of file in bytes as established by the installation
3 number of components in file (not more than 1000), including nulls
4 Boolean, 1 if any modification since file was last organized
5 number of accounts with file open
6 cycle number of last reorganization
7 last update time stamp: year, month, day, hour, minute, second, millisecond as 7 element vector

Examples:

```
      ⌼1
7
      ⌼'FILENAME'
1
      3⌼'FILENAME'
14
      7⌼'FILENAME'
1974  12  31  23  59  59  999
```

SECTION 8

FUNCTION DEFINITION, EDITING AND EXECUTION

GENERAL.

A defined function provides an algorithm for specialized processing. The algorithm, or solution method, is expressed in APL terms by the user in function definition and editing mode. This mode allows actions to be performed that define or edit the algorithm. The definition of the function is thus captured for subsequent execution or editing. Many different defined functions can coexist, recognized by their unique function names.

Execution of a defined function is similar to execution of a primitive function: it can be elaborated when the values for its actual arguments are determined. A defined function that returns an explicit result can be used similarly to APL primitive functions in composition of APL expressions.

FUNCTION CONTENT.

A defined function has a header line and a body. The header line begins with a template and optionally may include a list of local names, each preceded by a semicolon.

A function template determines the syntax required for its execution. A defined function may have any of the six templates:

	niladic	monadic	dyadic
Returns explicit result:	R←F	R←F B	R←A F B
Returns no explicit result:	F	F B	A F B

Where:
R is the local name for the function result
F is the function name
A is left local argument name
B is right local argument name

The names R, F, A and B must all be distinct. F must not have any current global meaning.

When the function is called to be executed, the argument local names A and B are established initially to have the argument data objects as values. Thereafter within the function the names A and B can be used like any local name. When the function execution is completed the meaning of the result local name R (a data object or undefined) is the function explicit result.

A local name is a name that can be attached to a data object (or fixed function) without affecting any use of that name outside (at a more global level than) the function. This determination is made for each instance of execution of a function. A name in the local name list has no meaning until given one during execution of the function.

System variables and the character input prompt communicator may also appear in a local name list. Until assignment to a system variable is made within the function, the global value is retained. This permits the calling environment to affect the returned result.

A function body has zero or more lines. Each line must have at least one of the following, in left to right sequence in the order given if more than one:

labels, each terminated by colon	`L1:`
branch transfer of control	`→L2`
APL expression	`,X←3+4`
comment	`⍝ NOTE`

For example, a line containing all parts is:

```
L1:→L2,X←3+4⍝ NOTE
```

Each line of the body has a line number: The first line is line 1, the next line 2, and so on. When displayed in function definition and editing mode, each line is preceded by a bracketed prompt including the line number.

Sample Function

```
    ∇ AVE←AVERAGE VALUES;SUM;SIZE
[1]   SUM←+/VALUES⍝ SUM OF VALUES
[2]   SIZE←ρ,VALUES⍝ SAMPLE SIZE
[3]   AVE←SUM÷SIZE
[4]  LABEL:⍝ AVE←(+/VALUES)÷ρ,VALUES
    ∇
```

The header line here defines a monadic, value returning function named AVERAGE with five local names: argument VALUES and explicit result AVE; local names SUM and SIZE, and label LABEL. Line 4 illustrates a labeled line containing as a comment an alternative and generally preferable algorithm that could be used to determine the average, if the comment character were removed. The initial and final ∇ 'Del' characters bracket the function.

Forms:

→ E	Branch to line E
→	Terminate
L:	Label

Where:

E	is a line specifier expression yielding a non-negative integer scalar or vector value
L	is a named local constant

Results:

Branching and terminating are the means to alter line control flow from the normal next line in sequence in defined functions. Labels provide names for lines.

Branch: After the line containing the branch is elaborated, the path of control transfers to the next line to be executed as determined by the non-negative integer value of the first element of E:

Value of first element of E	Next execution
a line number	that line
empty numeric vector	next line in sequence
0 or greater than last line	exit to caller

Terminate: Stop execution of this function and all functions pending its completion.

Label: A label is a local constant name used as a destination for branching. A label has as its value the number of the line in which it appears followed by a colon. One or more labels, each followed by a colon, may occur on any line. All labels must precede any branch, expression or comment on a line. No assignment of value to a label is permitted. Each label name must differ from the function name or any other local name in the function. Because function editing may cause line numbers to change, labels may be used to identify targets for branching. Labels are attached to line contents and so automatically change their values as function lines are renumbered through editing.

Conditions:

Branching and terminating apply to the function on top of the state indicator. That function is either being executed or suspended. If suspended, entry of a branch

BRANCH,
TERMINATE,
LABEL (2)
→ :

applies to relieve the suspension and continue execution. Terminate abandons execution of the function and any other functions pending its completion.

The comparison tolerance applies to determine if the first element of the value of non-empty E is an integer.

Branching in execution mode is ignored if there is no state indicator, otherwise it applies to the most recently suspended function.

In a user defined function, the branch or terminate function may only appear as the leftmost function on a line. Only labels may appear to their left.

No branch to any line in any pending function, other than the return to the point of call, is automatically provided. To achieve this the returned value may be used to select the desired line as target of a control transfer in the pending function when it is reactivated.

The constant value of a label may be referenced as a global value in a function called from the function in which the label is defined.

Examples:

Typical branching expressions include:

```
→L               ⍝ GO TO L
→0               ⍝ EXIT THE FUNCTION
→B/L             ⍝ IF B=1 THEN GO TO L ELSE CONTINUE (B=0)
→(L1,L2,L3)[N]   ⍝ BRANCH TO L1 IF N=1, L2 IF N=2, L3 IF N=3
→N↓L0,L1,L2      ⍝ BRANCH TO L0 IF N∊¯2 ¯1 0, TO L1 IF N=1, TO
                 ⍝ L2 IF N=2, ELSE CONTINUE
→(×E)⌽L2,L3,L1   ⍝ BRANCH ON SIGN OF E: TO L1 IF E<0, TO
                 ⍝ L2 IF E=0, TO L3 IF E>0
→N+⎕LC           ⍝ BRANCH TO CURRENT LINE + N
→B×L             ⍝ EXIT IF B=0 ELSE TO L IF B=1

→                ⍝ TERMINATE
```

Where: L, L0, L1, L2, L3 are line number specifiers
B is Boolean
E is expression yielding numeric single
N is integer

FUNCTION EDITING ACTIONS.

A defined function is created and edited in function definition mode. This mode is entered using the character ∇, followed by the function header if this is a new definition. If the function already exists, the ∇ is followed by only the function name (and an optional action specification to be described).

The function definition mode may be recognized by the display of a bracketed prompt starting at the left margin. This prompt is the default action specifier indicating a line number where the next entry will appear unless overridden by an alternative action. This prompt is generally to a non-existent line (the next line in sequence), so no current line will accidentally be replaced.

To begin defining a defined function, the initial line entered is the header. The prompt returned is [1], the default action specifier for the next entry. An entry following the default action specifier not commencing with a ∇, ⍫ or a [causes the line referred to by the prompt to receive the entered string of text, and then a new prompt to be returned (if the text string is syntactically valid).

If the last character entered after the prompt is a ∇ or ⍫, function definition terminates and the five character indent prompt is received indicating the return to execution mode.

Six classes of function editing actions will be described:

Function definition, open and close
Line replace, append or insert
Line content edit
Line group diagnostic aids
Line group display
Line group delete

Each action is recognized by its unique form. The action specifier encloses this form in brackets.

If an action is entered at the start of an entry, it overrides the displayed prompt for that line.

The numbers associated with (but not part of) lines of the function body are always the continous set of integers starting with 1 for the number of body lines. The header is referenced as line 0. If lines are inserted or deleted, line numbers larger than the smallest line affected by the action will be renumbered.

Most action specifiers identify the line(s) to which they apply by inclusion of one or two line number specifiers. A line number specifier has an integer value of an existing line or sometimes 1 + the last line number. This value may have any of the forms:

integer	absolute line number
label	existing in function
label + integer	relative to and following label
label - integer	relative to and preceding label

FUNCTION
DEFINE, OPEN,
CLOSE ACTIONS (1)

∇ ⍫

Forms:		next prompt
∇ H	Define function header H	[1]
∇ F	Open defined function with name F	[Z]
⍫ F	Open own locked defined function F	[Z]
∇	Close open function	indent 5
⍫	Close and lock open function	indent 5

Where:

H is function header for new function
F is existing function name
Z is 1 + the last line number

Actions:

Function define or open changes system mode from execution to function definition and editing. Function close returns to execution mode.

Define: Create a new function with header H. H has the form of one of the templates, possibly followed by a list of local names each preceded by a semicolon. The function name in the template must not already have current global meaning.

Open: Reopen an existing defined function. The open entry can include an action specifier and text if desired.

In either case the prompt displayed is the bracketed line number of the next unused line, unless the open with action overrides.

Close: The close symbol (only entered as the last non-blank character of a line) closes the function and returns to execution mode. It can follow a prompt, or any command except full edit.

In place of the ∇ character if ⍫ is used with close, the function is locked. Subsequent opening using the ⍫ can only be done by the workspace owner loading (not copying) the workspace in which the function was created. A locked function cannot be opened if it is copied into another workspace or loaded into the workspace of a another account.

During execution of a locked function, user initiated ATTN or any error encountered causes function exit, and passes any error message to the caller environment. Line trace and suspend within a locked function are ignored, even though their settings are retained should the function be subsequently unlocked.

Examples:

```
      ∇R←F X;Y∇

      ∇F
[1]   LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
[2]    ⍫

      ⍫F
[2]   LABEL2:R←G X+Y⍝ G IS A FUNCTION∇

      ∇F
[3]    ∇
```

The first example creates a function header and then immediately closes, effectively reserving a name for subsequent function editing which will provide the function body.

The next open returns the prompt [1]. The content of line 1 is then entered. After the prompt [2], the function is locked.

The next open of the locked function must use ⍫. The prompt is now [2], the first unused line. That line is given content and the function unlocked by the close with ∇, here done at the last character of the entered line.

The final open demonstrates that the function is now unlocked. After the prompt [3] the function is again closed.

FUNCTION LINE
REPLACE, INSERT
ACTIONS (1)

↓ ↑

Forms:

		next prompt if T is	
		empty	non-empty
[A]T	Text of line A is replaced by T	[A]	[Z]
[↑]T	Insert text T before prior line 1	[↑1]	[↑2]
[↑B]T	Insert text T before prior line B	[↑B]	[↑B+1]
[↓]T	Insert text T at end	[Z]	[Z]
[↓C]T	Insert text T after line C	[↓C]	[↓C+1]

Where:

A is existing line specifier or Z
B is existing line specifier except 0
C is existing line specifier
T is text string or empty
Z is 1 + last line specifier

Actions:

If T is empty, the entered action specifier becomes the next prompt, otherwise the text of T becomes a line.

Replace: Replace the prior content of line A (if A exists) by T. Replace causes no change to line numbering.

Insert before: Create a new line B with content T and increase by one the line specifiers of the former lines starting with B (or Z). The next prompt allows continued insertion before the same old line, whose number increases by 1 for each insertion.

Insert at end: Create a new last line with content T without affecting any prior line. Same as replace entry to line Z.

Insert after: Creates a new line with content T, and increases all former line specifiers larger than C by one. The next prompt allows continued insertion before the original line C + 1, whose number increases by 1 for each insertion.

Conditions:

If C has value Z-1 then the action is the same as insert at end, and the next prompt is Z.

FUNCTION LINE
REPLACE, INSERT
ACTIONS (2)
↑ ↓

If the text T is empty (the entry contains only one of these action prompts) this prompt becomes the next prompt instead of the one indicated above (a line without content is not allowed). By this means, using the replace action it is possible to have the default prompt refer to an existing line. Subsequent entry of text only (without another action) destroys the prior content of the line.

Examples:

```
        ∇F[2]R←Y+3
[3]     [↑]⍝ NEW LINE 1
[↑2]    [↓]⍝ LAST LINE
[5]     [↑3]⍝ NEW LINE 3
[↑4]    ⍝ AFTER 3
[↑5]    [⎕]∇
        ∇ R←F X;Y
[1]     ⍝ NEW LINE 1
[2]    LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
[3]     ⍝ NEW LINE 3
[4]     ⍝ AFTER 3
[5]     R←Y+3
[6]     ⍝ LAST LINE
        ∇
```

FUNCTION LINE
EDIT ACTIONS (1)

ϵ α ω ι

Forms:		next prompt
[ϵA]	Full edit line A	[Z]
[αA]	Prefix edit line A after line number	[Z]
[αA]T	Prefix text T before text of line A	[Z]
[ωA]	Suffix edit line A	[Z]
[ωA]T	Suffix edit text T after text of line A	[Z]
[ιA]	Inject text of line A to last executed APL expression	[Z]

Where: A is a specifier of an existing line
T is text string
Z is 1 + the last line number

Actions:

Full Edit: Display line A, and return carrier to left margin awaiting edit position controls entry under any characters of line A. These controls may only include spaces, periods and slashes: space indicates no change, '.' indicates phrase terminator before character above, '/' indicates delete character above. Upon next RETN, the first phrase is displayed ready for normal entry typing. Each subsequent ATTN with cursor to the right of the current display brings the next phrase. Any RETN causes entry of the line as it appears. If there are no more phrases left, an ATTN acts like a RETN. During any phrase, ATTN not at the rightmost attained display position acts to delete display characters above and to right, but not undisplayed phrases. The entire line (including prompt, labels, APL expressions, and comments) may be edited. There must be at least an action specifier remaining when the entry is made.

Prefix Edit: This edit bypasses the edit position controls entry and assumes a single '.' was entered after the bracketed line number. This command displays the prompt, then awaits entry. This is useful either to change the line number within the prompt (and thus make a second copy of the original line) or to place a label or further expression at the start of the existing line.

Prefix Edit with string T causes the string to become the leftmost part of the line following the prompt, without displaying the line.

Suffix Edit: This edit displays line A and awaits text entry at the end. A change near the end of a line may often be made more quickly using this action (by backspaces, ATTN, then correction) than using full edit.

Suffix Edit with string T appends T to the end of line A, without displaying it first.

Inject: Place a copy of the content of line A into "the last executed APL expression", available for examination, alteration and execution in execution mode. Only the last inject done in function definition and editing mode applies at function close. If no inject is done, then the most recently executed APL expression is unchanged by function mode actions.

Conditions:

An edit that removes all non-blanks from the line is the same as a new action. No edit can remove the action. Changing the line number relocates a (possibly edited) copy of the line. The original line remains: if it is labeled, the line copy will only be permitted if the label is changed.

Text insertion as part of prefix or suffix edit actions does not provide visual fidelity since only the change to the line is shown.

Examples:

```
     ∇F[∊1]
[1]   ⍝ NEW LINE 1
        ///.
[1]   ⍝ EDITED PHRASE IN LINE 1
[7]   [ω2]
[2]  LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
                                v
                                MONADIC FUNCTION
[7]   [α1]
[1]   X⍝ EDITED PHRASE IN LINE 1
[7]   [⍳2]∇

     LABEL1:Y←LINE X⍝ LINE IS A MONADIC FUNCTION
     ///////
     Y←LINE X⍝ LINE IS A MONADIC FUNCTION
```

FUNCTION
MULTILINE GROUP
SPECIFIER (1)

Actions having potential effect on more than one line use the following forms for indicating the lines in the group. The character o is used to indicate any one of the allowable actions.

Unqualified: applies to all lines in the range.

Form	Line Range
[o]	0 thru Y
[oA]	A only
[Ao]	A thru Y
[AoB]	A thru B

Name Qualified: applies to only those lines within the inclusive range that contain the name X.

Form	Line Range
[(oX)]	0 thru Y
[(oX)A]	A
[A(oX)]	A thru Y
[A(oX)B]	A thru B

Where: o is any multiline function editing action, one of

⊤ ⊥ ⌈ ⌊ ∩ ∪ ⎕ ? ~

A is line number specifier: $A \epsilon\ 0,\ \iota Y$
B is line number specifier not less than A:

$B\ \epsilon A,\ A + \iota Y - A$

X is name of label, function or variable
Y is number of the last line defined for function

Examples:

To illustrate line specifier use, the action character ⎕ (display lines) is used.

```
            ∇F[⎕]
          ∇ R←F X;Y
     [1]    ⍝ NEW LINE 1
     [2]   LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
     [3]    ⍝ NEW LINE 3
     [4]    ⍝ AFTER 3
     [5]    R←Y+3
     [6]    ⍝ LAST LINE
          ∇
     [7]    [LABEL1+1⎕4]
     [3]    ⍝ NEW LINE 3
     [4]    ⍝ AFTER 3
     [7]    [5⎕]
     [5]    R←Y+3
     [6]    ⍝ LAST LINE
     [7]    [⎕1]
     [1]    ⍝ NEW LINE 1
     [7]    [(⎕LINE)]
     [2]   LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
     [7]    [3(⎕Y)]
     [5]    R←Y+3
     [7]    ∇
```

Note the initial display action ∇F[⎕], does not include a close, ∇, at its end. Therefore, after the display of the entire function, ∇ is shown to indicate that line 6 was the last defined line; then [7] prompt is given. This indication only occurs if the entire function is displayed. The 7 in [7] is 1 + the last line, and appears after each of these examples and serves as a default for entry of a next line unless a new action is specified. In each of the above cases, a new display action is specified following the [7]. All other lines above are the result of these display actions.

The qualified use of the name LINE does not recognize occurrence of LINE in comments, in quotes, or as part of another name.

DIAGNOSTIC
FUNCTION LINE
GROUP ACTIONS (1)
⊤ ⊥ ⌈ ⌊ ∩ ∪

Forms:

		system function	next prompt
[⊤]	Set trace	⎕*ST*	[Z]
[⊥]	Reset trace	⎕*RT*	[Z]
[⌈]	Set stop	⎕*SS*	[Z]
[⌊]	Reset stop	⎕*RS*	[Z]
[∩]	Set monitor	⎕*SM*	[Z]
[∪]	Reset monitor	⎕*RM*	[Z]

Where: Z is 1 + last line number.

Actions:

These actions are analogous to the system functions by the same names, except that they are entered in function definition mode, and may only refer to a group of contiguous lines, possibly name qualified (the principal advantage). Both these actions and the system functions have the same execution effects.

Trace: Upon completion of execution of a line on which trace is set, the function name and bracketed line number is printed followed by the type (N numeric, B Boolean, C character), shape in parentheses, and value. Trace of line 0 refers to the returned value (if any) on function exit.

Stop: Upon transfer to a line on which stop is set, the function suspends there, the function name and bracketed line number are displayed followed by an asterisk. Control returns to execution mode for user examination or alteration of the current state. Stop on line 1 causes suspension after actual arguments are assigned but before any processing in the body. Stop on line 0 causes suspension before actual return to the caller, so all local names still have values.

Monitor: Upon completion of execution of each monitored body line, the computer time there consumed is accumulated in a counter for that line. The precision of this time is 2.4 microseconds. This time excludes time spent within any user defined functions called in that line. (Such time may be separately monitored in their own body lines). Monitor of line 0 provides a count of the number of calls on the function. The display unit for these times is milliseconds and the result is rounded, (⎕MV gives the ceiling of the time instead).

DIAGNOSTIC FUNCTION LINE GROUP ACTIONS (2)
⊤ ⊥ ⌈ ⌊ ∩ ∪

Conditions:

The forms for the range of lines specified resulting from inclusion of left and/or right line specifiers and parenthesized name qualifier apply.

Examples:

```
          ∇F[⎕]
        ∇ R←F X;Y
[1]       ⍝ NEW LINE 1
[2]     LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
[3]       ⍝ NEW LINE 3
[4]       ⍝ AFTER 3
[5]      R←Y+3
[6]       ⍝ LAST LINE
        ∇
          ∇F[⊤]
[7]     [(⌈R)]
[7]     [(∩Y)]
[7]     ∇
        ⎕QT'F'
1  1  1  1  1  1  1
        ⎕QS'F'
1  0  0  0  0  1  0
        ⎕QM'F'
1  0  1  0  0  1  0
        ∇F[LABEL1(∪Y)]∇
        ⎕QM'F'
1  0  0  0  0  0  0
        ∇F[(∩LINE)]∇
        ⎕QM'F' ⍝ ADDITIVE, NOT IN COMMENTS
1  0  1  0  0  0  0
```

DISPLAY
FUNCTION LINE
GROUP ACTIONS
⎕ ?

Forms:

		next prompt
[⎕]	Display lines	[Z]
[?]	Display vector of line numbers	[Z]

Where: Z is 1 + last line number

Actions:

Lines: display lines causes display with bracketed line numbers followed by text of all indicated lines.

Numbers: display numbers causes display of the numbers of all lines in the indicated range. This is useful where name qualified.

Conditions:

The forms for the range of lines specified resulting from inclusion of left and/or right line specifiers and parenthesized name qualifier apply.

Name qualification displays the line number but not the header line content, should the name occur in the header, line 0.

Examples:

```
      ∇F[⎕]
     ∇ R←F X;Y
[1]    ⍝ NEW LINE 1
[2]   LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
[3]    ⍝ NEW LINE 3
[4]    ⍝ AFTER 3
[5]    R←Y+3
[6]    ⍝ LAST LINE
     ∇
      ∇F[1(⎕Y)]
[2]   LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
[5]    R←Y+3
[7]    [(?Y)]
2  5
[7]    [(⎕LINE)]
[2]   LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
[7]    ∇
```

DELETE
FUNCTION LINE
GROUP ACTION
~

Form: next prompt

[~] Delete lines in indicated range [Z]

Where: Z is 1 + last line number remaining

Action:

The lines in the indicated range are deleted. If qualified, only those lines containing the qualifying name are deleted.

Deleting lines causes renumbering of lines after the first deleted.

The effect of [~0] is only to eliminate the local names list from the header; the template cannot be deleted, and thus the line remains.

Conditions:

The forms for the range of lines resulting from inclusion of left and/or right line specifiers and parenthesized name qualifier apply.

If a sequence of deletions (or line insertions) is to be done, they should be done from the bottom up so that renumbering will not effect the previously known line numbers.

Example:

```
      ∇F
[7]     [~6]
[6]     [3~4]
[4]     [0~1]
[3]     [⎕]∇
     ∇ R←F X
[1]   LABEL1:Y←LINE X⍝ LINE IS A FUNCTION
[2]     R←Y+3
     ∇
```

DEFINED FUNCTION EXECUTION.

The execution of an instance of a defined function begins when the function is called (appears in an expression being executed) either from execution mode or by another function. From the instant execution of an instance of a function begins until the execution of the instance of the function is completed, the function is active. An active function is either in process of being executed, or may be suspended or pendant. A pendant function is one which is awaiting completion of a function it called. A suspended function is one whose execution was stopped for some reason other than a call to another function.

SCOPE OF NAMES.

A name can be global, having existence in the workspace independent of an execution of a defined function. It can also be specified as local in a defined function. The existence (scope) of a local name is then no longer (in time) than the instance of the function is active to which it is local. A local name to one defined function becomes global to any function called from that function. A global name becomes inaccessible while an instance of a local use of the same name exists.

A local variable or function can be dynamically expunged from within the function to which it is local. The name is still local, so a more global instance does not become accessible.

The importance of scope is its aid to structured programming. Names that are of no consequence outside the function to which they are local need only be contained (and thus known) therein. Understanding at the global level is not confused by these extraneous names.

EXECUTION CONTROL SEQUENCE.

At function call, the values of arguments are bound to their equivalent local arguments. All local names are established. If any of these names already had more global meaning, that meaning is shielded while that instance of the function is active.

Execution begins with control at line 1 of the function. Within each line order is right-to-left elaboration of primitive or other defined functions. When a line is completed, control moves to the next line in sequence unless explicitly altered by a control transfer.

Function completion occurs when control transfers to line 0 or some other non-existent line (including implicit last line plus 1). If an explicit result variable is included in the function header and is required by the call, a value must have been assigned to it prior to completion. The last such executed assignment is the value returned by the function.

MULTIPLE INSTANCES.

More than one instance of execution of a function may be active at the same time. This can result from *unrelated* calls on the same function name (directly or indirectly via call from some other function) while the earlier instance is pending or suspended. This is generally to be avoided as extra space is consumed. Recursive function calls are permitted, which also causes multiple instances.

RECURSIVE FUNCTIONS.

A function is *recursive* if completion of one instance of its call can require another call on another instance of the same function. Recursive functions are the natural means to formulate some algorithms. A *directly* recursive function includes a call on itself. An *indirectly* recursive function includes a call on some other function that either itself calls the first, or includes in its call sequence one that does. The number of instances is limited by the amount of space required for each instance and the amount of space available in the workspace.

Determination whether or not a function is potentially recursive is generally not possible. Recursion is a dynamic property of an instance of a function, determined by data values. The appearance of more than one instance of a function in the state indicator without intervening suspensions does indicate recursion. An intervening suspension does not necessarily indicate whether a reappearing function is recursive.

Static function content examination may detect potential recursion. Since dynamic control flow is generally not known, actual recursion is even less readily recognized. If the evaluate primitive and function fixing are excluded, it is possible to detect potentially recursive functions by recognizing the reappearance of the function name within itself, or in a function in any potential static call sequence of other functions from it. This process is complicated since a name may be in some contexts either a variable or a function, only known dynamically. If the source data object for function fixing is known and examinable, it can be handled as above. The source may not be known; it may be any expression. Since evaluation or fixing of a general expression is permitted, in general no static examination will suffice to detect all potentially recursive functions.

SUSPENSION OF DEFINED FUNCTION EXECUTION.

The normal line-to-line path of control resulting from defined function execution may be interrupted by execution suspension. Suspension occurs in three ways:

The path of control reaches a line with a stop set on it.

The user enters one or two ATTN during function execution or output. The first kills output and suspends after completing any line in progress. The second may interrupt mid-line.

An error occurs in the line recognizable only during execution.

The result of suspension is a return to execution mode after displaying the suspension prompt, typically for line 3 of the function named RUN as

```
RUN[3] *
```

While execution of a function is suspended, it is still active. The user may do most of the things normally available in execution mode, but in the environment defined by that instance of the function:

examine or alter values of local or unshielded global variables
create new variables or define new functions
enter expressions or system commands for evaluation
alter the most recent suspended function by edit actions

No pendant or suspended function other than the most immediately suspended one can be altered. (They can be displayed and diagnostic aids changed). The header line cannot be changed in the suspended function. No pending or suspended function may be expunged.

Execution of the suspended function may be resumed. To resume on the line specified by expression N (which need not be the same as the line where suspension occurred), enter:

```
→N
```

Termination of the execution of the suspended function (and any pending its completion) may be achieved by entering

```
→
```

The response to termination is a reminder of the suspension prompt for the immediately prior suspended function if any; followed by the execution mode prompt.

It is good practice to eliminate all suspensions soon after they occur, as suspended and pendant functions take up space in the workspace. The user should usually avoid a second execution of a function from the beginning after execution is suspended.

The *)RESET* system command may be used to remove all suspended functions at once, rather than entering a sequence of terminates.

DEFINED FUNCTION EDITING USING APL FUNCTIONS.

An alternative to line-at-a-time function editing exists: edit a data object that represents a function, then fix it back into a function again.

The canonic representation ⎕CR is a convenient means to create a data array from a function with one row per line. In this form, user defined functions can be used to select or rearrange lines. Simple defined functions permit merging separate function bodies or selecting line groups to become the body of a new function.

The alternative vector representation ⎕VR of a function is convenient for name replacement or other contextual editing.

After completion of editing on these APL variables, they may be refixed into functions by ⎕FX. If the function name in the header is unchanged, the old version must be purged using ⎕EX or)ERASE first.

DEFINED FUNCTION DOCUMENTATION.

One approach to documentation is to have function pairs: one executable, the other containing the documentation (each line a quoted string). A common way to relate the pair is to suffix the executable function name by 'HOW'. This method sacrifices the proximity of the functions to their descriptions. The space saving results from erasing or excising all the 'HOW' functions before execution. An alternative is to save the 'HOW' as a variable. The vector representation is useful in that it can readily be fixed for changes.

A second approach is to maintain two equivalent workspaces: one for documentation, the other for execution. The documented functions can have copious comments and descriptive names. Then this documented workspace is saved and a copy of it edited to shorten these names and eliminate comments. This condensed workspace becomes the working version.

A third approach is to maintain vector representations of functions as file components. Vector representation is preferable to canonic representation for this purpose as it is generally more compact. Selective fixing of needed functions and expunging of extraneous functions can be used to save much space. The documentation can normally be left in the file components. Either of the previous approaches can be used in conjunction with this to minimize the size of the vector representation that is used as the basis for function fixing. If the name of a function to be fixed is in the local names list of a small "cover" function which fixes it then automatic expunging occurs upon exit from the cover function.

SECTION 9

ERROR REPORTS AND THEIR INTERPRETATION

GENERAL.

The APL/700 system includes a comprehensive error-reporting capability that helps to determine the cause of error, the specific location, and the corrective action. This section provides descriptions of the various error reports and sufficient information to aid the user to interpret and correct errors. A complete listing of error reports is contained in Table 9-1.

ERROR REPORTS.

An error message line displayed on the terminal starts at the left margin. It indicates the error message text and is surrounded by asterisks.

```
*** SYNTAX ERROR ***
```

Additional lines may be displayed, depending on the particular error.

If the error is detected in an execution mode entry the second line indicates the point(s) at which the error is detected. The third line is the entry in error. An ATTN entered here recalls this entry for inline editing. (See Section 2.)

```
     8 6 7-5 3
*** LENGTH ERROR ***
          v
     8 6 7-5 3
```

An error detected during attempted execution of a line of a user defined function results in the error report, then a line containing the function name and bracketed line number, asterisk indicating suspension on that line, then the line content. The next line indicates the error position(s).

```
     TEST
*** LENGTH ERROR ***
TEST[1]* 3 4+4 5 6
            ^
```

The return to execution mode allows examination of the process state and adjustment if desired. The suspended function can be opened or altered as desired. Execution may be resumed.

Note the down-caret v indicates that the error is in the last entered expression and is available for error correction. The up-caret ^ is displayed otherwise.

Two additional lines may appear if the error is detected during an attempted evaluation. These lines indicate the errored position in the string being evaluated. They occur after the error message.

```
          ⍎'1 2 3+4 5'
     *** LENGTH ERROR ***
          v
     1 2 3+4 5
          v
          ⍎'1 2 3+4 5'
```

A similar indication occurs for an error in evaluation during execution of a function line. Note the difference in caret use.

```
          TRY
     *** LENGTH ERROR ***
          v
     1 2 3+4 5
     TRY[1]* ⍎'1 2 3+4 5'
                ^
```

If any characters other than ' ', '/' or '.' appear in the edit specifier of a line edit, the one line error message appears.

```
          1 2 3+4 5
              2
     *** EDIT ERROR ***
```

The REPORT column of Table 9-1 lists in alphabetical order the error report texts. The DEFINITION column provides the corresponding system interpretation of the cause for each error report. Where applicable, corrective action is indicated.

The basis for error reports is system inability to complete an indicated transaction. The report identifies what is found to be wrong; it does not try to prejudge a correction.

If the user types a parenthesis in the wrong location, or omits a required entry, the system can only report what problem it encountered as it tried to execute the instruction, it can't tell the user what should have been typed. This has to be determined by the user alone.

Normally, when the error occurs, the expression has to be edited or reentered. The value of an intermediate expression within the instruction is not saved, unless the instruction specifically directs that it should be assigned to a name. This arises only when a specification arrow was executed earlier than the caret that indicates where the trouble is. If the result of an intermediate step has been assigned only the unexecuted part of the entry has to be reentered.

The following paragraphs give samples of how some of the more common errors may occur.

When the user attempts to enter an expression whose syntax is invalid, the "SYNTAX ERROR" message is reported. Examples causing this error include: two variable names appearing without an intervening function, a missing function argument, or unmatched or mispaired parentheses or brackets (several caret marks may result).

Incorrect usages of the definition mode include: embedding the del (∇) not within quotes in a line entry, attempting to alter the definition of any active function not on top of the state indicator, or to alter the header line of the suspended function on top of the state indicator, attempting to start a new definition for an existing function whose header contains a result, an argument, or a local names list, and entry of an incorrect action request.

When an argument to a function contains an element outside the domain for which the function is defined, a "DOMAIN ERROR" message is reported, for example, an attempt to divide a non-zero value by zero.

A "TYPE ERROR" message is reported if the type is incorrect for the function. Examples are attempts to perform arithmetic on character objects, catenation of character with numeric objects, or character object insertion into a numeric array.

A "VALUE ERROR" message indicates that the expression being elaborated references a name for which no value has been assigned. Causes are failure to assign a value to that name to make it a variable, misspelling the name, or failure to define a function of that name. A value error will also arise if the result of a defined function is required but the function definition or execution fails to provide one.

A "RANK ERROR" message indicates that the arguments to a dyadic function are non-conformable or an argument has improper rank for the particular function. Some functions (such as the left arguments of ⍳ or ⍉) can take arguments only of rank 1 or rank 0. Grades require a rank 1 argument.

Any error report on any system command indicates failure to process. There are no side effects of partial processing.

Any system response not enclosed in asterisks is information only, it does not indicate an error. For example

```
      )ERASE X
NOT X
```

Table 9-1

Error Reports (1)

*** Report ***	Cause
ACCOUNT ACTIVE	An attempt was made to sign on an account that is already signed on to APL.
ACCT-NAME ERROR	A reference was made to a nonexistent account, or the name was improperly formed.
BUFFER LIMIT	An attempt was made to execute a string longer than the buffer, or an attempt was made to set the prompt to be a string longer than the buffer. The buffer length is 1620 characters.
CHARACTER ERROR	An invalid overstrike was entered. The locations of the invalid overstrikes are indicated by the squish quad (⌷) symbol.
CONTEXT ERROR	A name was used out of context with its current definition.
CONTROL ERROR	A parameter to a command was incorrect.
DEFINITION ERROR	An attempt was made to define a new function with a name that already exists, or the function header was improperly formed. (Refer to Section 8.)
DIMENSION ERROR	The dimension specified does not exist. (This occurs with a function that can be applied on one of several dimensions.)
DOMAIN ERROR	The argument of a function (or any element of it) was outside the acceptable values for that argument to the function.
DUP-NAME ERROR	An attempt was made to give a local name multiple definitions, or to repeat a label.
EDIT ERROR	Something other than a ' ', '/', or '.' editing control symbol was typed beneath a line when using the full edit action.
FILE ACTIVE LIMIT	The user has the maximum number of files permitted; no more requests to make more files active can be accepted.
FILE ALREADY EXISTS	An attempt was made to create a file that already exists.

Table 9-1 Error Reports (2)

*** Report ***	Cause
FILE ERROR	Either execution of APL was halted or a line-drop occurred while a file operation was in process. The file operation may or may not have been completed.
FILE INDEX ERROR	An attempt was made to read or write a component of a file with index value more than one larger than exists in the file.
FILE LOCKED	Either no password when required or an incorrect password was used in a file reference.
FILE NAME ERROR	An attempt was made to use an improperly formed name as a file name.
FILE NONCE ERROR	The file operation referenced is not presently implemented.
FILE NONEXISTENT	The referenced file does not exist.
FILE QUOTA LIMIT	An attempt was made to create more files than the account is permitted.
FILE SPACE LIMIT	The space reserved for the file has been exhausted.
FILE SYSTEM ERROR	An unexpected execution error occurred in the file system. (This should be reported to the system manager; all relevant output should be saved.)
FILE SYSTEM LIMIT	The maximum number of files allowed to be active are currently active; no more requests that activate a new file can be accepted at present.
FILE UNAVAILABLE	The referenced file is unavailable at this time.
FILE USERS LIMIT	The maximum allowable number of file users are currently using the file system; no more file users can be accepted at this time.
FILE VALUE ERROR	An attempt was made to access a null component of a file.
FORMAT ERROR	The left argument to the format function is not a valid format.
GRP-NAME ERROR	A reference was made to a nonexistent group.
INDEX ERROR	An index into an array was out of the array bounds.

Table 9-1 Error Reports (3)

*** Report ***	Cause
INTEGER LIMIT	A number larger than the largest integer that may be represented by the machine was used where an integer was needed. The magnitude of the largest integer is 549755813887 ←→ ¯1+8*13.
INTERRUPT ERROR	An error was forced at a non-suspendable point by striking the attention key twice.
LENGTH ERROR	The length of a vector is incorrect for a function using one or more vector arguments.
NAME ERROR	An argument to a system function requiring a name was given an improperly formed name, or a name with incorrect meaning was given.
NONCE ERROR	An attempt was made to use a feature that is not presently implemented.
NUMBER LIMIT	The result of a computation is a number with magnitude greater than the largest number that the machine can represent. The magnitude of this number is 4.31359146674E68 ←→ (¯1+8*13)×8*63.
PASSWORD ERROR	An incorrect password was used.
RANK ERROR	The rank of an object is incorrect for the function to which it is an argument.
RANK LIMIT	An attempt was made to create a structure whose rank was greater than 16, the maximum allowable.
SHAPE ERROR	The shapes of objects are incompatible for the function to which they are arguments.
SIGN-ON ERROR	An incorrect sign-on entry was made.
SIZE ERROR	A one-element object was needed as an argument to a function, but it was not found.
SPACE LIMIT	An attempt was made to use more space than is available in the active workspace.
STATE ERROR	A edit request was made on a function which could cause the state indicator to be incorrect if the edit were performed.

Table 9-1 Error Reports (4)

*** Report ***	Cause
SV - QUOTA LIMIT	An attempt was made to share more variables than the processor is permitted to share.
SV - SPACE LIMIT	An attempt was made to use more shared variable space than the processor is permitted.
SV - UTILITY ERROR	An attempt was made to offer a variable to an undefined utility.
SYMBOLS LIMIT	An attempt was made to create more symbols than there is space for in the symbol table. (Unless otherwise specified by the user, there is space for 256 symbols.)
SYNTAX ERROR	The syntax of the APL expression entry is incorrect.
SYSTEM LIMIT	APL encountered an unexpected error during execution. (This problem should be reported to the system manager; all relevant output should be saved.)
TIME-QUOTA LIMIT	This error occurs once an account has exceeded its computer usage quota. The user session is then terminated, and the quota must be increased before the account may use APL again.
TYPE ERROR	The type of an argument is incorrect for the function being done.
VALUE ERROR	An attempt was made to use a name as an argument for which no value has been specified.
WS-NAME ERROR	A reference was made to a nonexistent workspace, or the name was improperly formed.
WS-QUOTA LIMIT	A)SAVE could not be executed because the account has used all available workspace slots. Some workspace must be dropped, or the workspace quota for the account must be increased.

UNIMPLEMENTED CONSTRUCTS.

Some constructs previously described are not implemented in the 2.7 release of APL/700.

1. A "SYNTAX ERROR" results from an attempted dimension selection from the anti-origin for the structure mixed primitive functions:

⊖[*K*]*B*	reverse
A⊖[*K*]*B*	rotate
A⌿[*K*]*B*	compress
A⍀[*K*]*B*	expand.

2. An empty segment in the character format string gives a "FORMAT ERROR".

3. The dyadic form of the edit system function B ⎕*ED* F gives "SYNTAX ERROR".

4. The result of monitoring line 0 of a defined function does not provide a count of the number of calls on the function, but gives 0 invariably.

5. The Name List system function ⎕*NL* does not permit specifying the value 0, meaning objects with no associated meaning. A "DOMAIN ERROR" is given instead.

6. The Shares Availability system function ⎕*SA* gives a "SYNTAX ERROR".

7. Prefix and suffix edit actions with following text give "EDIT ERROR":

[α*L*]*T*	prefix
[ω*L*]*T*	suffix.

APPENDIX A

GLOSSARY

Term	Meaning
Account Name	The identification which the APL system records resources consumed by a user.
Across	An orientation of a "plane" orthogonal (at right angles) to a specified dimension of an array.
Active Workspace	The working area within which all transactions are performed.
Along	An orientation of a vector, relative to a specified dimension of an array. Vectors can be considered to be "along" a dimension when they are parallel to the axis of that dimension.
APL	A Programming Language. A language for describing procedures in an interactive environment. Originally developed by K. Iverson.
APL/700	APL enhanced for the Burroughs 700 series of computers.
Argument	A data object (or list) supplied to a function or operator.
Array	A data object having shape. An array may be a vector, a matrix, or an n-dimensional object and may have zero or more elements.
Assignment	Replace, insert into, or modify the value attached to a variable name.
Boolean	Subtype of numeric data type, having values 0 (false) and 1 (true).
Calculator Mode	See Execution Mode.
Character Type	Data object containing literal character elements.
Coercion	Replication of a data object to a conforming shape for the function being applied to it.

Term Meaning

Comment (APL) Any text prefixed by the lamp symbol (⍝) and terminated by RETURN or a new line.

Component Any member of a list. A component may be any data object or may be null. (Also see File Component.)

Constant A data object without name.

Control Structures The rules for determining order of execution.

Corner Any n-dimensional sub-array having for each dimension at least one face that is a sub-face of an n-dimensional array.

Data Object A unit of data for processing, with properties: type, rank, and possibly shape and value.

Defined Function A procedure or program defined by a user, containing lines of APL expressions and used to perform a discrete function, such as averaging.

Definition Mode Mode of APL system in which defined functions are created or altered.

Dimension One of the independent axes of a shaped data object. Dimensions are numbered from 1 to n for an n-dimensional object (origin 1).

Dimension Qualifier A single indicating the dimension for coercion or application of a function or operator.

Domain Allowable set of values for function argument.

Dyadic Function A function having two (explicit) arguments (left and right).

Elaboration The process of applying functions to arguments in an expression to determine its value.

Element A scalar object; for an array, located by a set of scalar indices for each dimension.

Empty A size-zero datum of any rank with shape and type.

Execution Mode Normal mode of APL/700 terminal in which entries are directly executed.

Expression A constant, variable, a niladic, monadic, or dyadic function, or syntactically valid combination of these.

Term	Meaning
File	A named workspace extension with file components containing data objects.
File Component	An APL data object referenced by file name and either file component number or end of component queue.
File Library	The files owned by an account.
Fill	Objects used to expand the size of a datum. Blanks (spaces) are used for character objects; zeroes (0's) are used for numeric objects.
Format	Specifier for mapping of a list of data objects of various types into a character type data object.
Function	A transformation on zero, one, or two arguments that generally produces a value.
Function Definition and Editing Mode	Mode in which functions are defined or changed.
Global	Definition of a name outside (in the calling environment of) a defined function. See local.
Group	A name to which other related names are associated for reference.
Identifier	A string starting with a letter of the alphabet, an underscored letter, or a delta (∆) or underscored delta (∆̲) and followed by zero or more of the above characters, the digits, or underscore.
Inactive Workspace	A workspace in a user library.
Index Number	An integer specifying the position of a plane across a dimension of an array, starting with the origin.
Index Origin	The first ordinal number, either 0 or 1.
Instance	A single occurrence of the environment resulting from execution of a defined function, commencing with its call and completing either by return to the calling environment or termination. The environment of local names shields any more global uses of the same names.
Integer	Subtype of numeric having no fractional part; in inclusive range 1-2*39 to (2*39)-1.

Term	Meaning
Iteration	A single execution of repetitive function lines, returning to common point in a loop.
Label	Local name for line of defined function, always followed by ':', having constant value the line number on which it occurs.
Lamp Character ⍝	A prefix to denote comment text following in entry or on a line of a defined function.
Last Executed Expression	The retained string last entered, available for recall by ATTN for further editing.
Library	Inactive workspaces of an account stored for later use. Also workspaces from other accounts to which access has been granted.
List	Expression, or sequence of component expressions separated by semicolons.
Local	Definition of a name within (local to) a defined function, possibly shielding a more global instance of meaning of that name.
Lock	A user access control to protect an account, workspace, file, or function.
Loop	Failure to find a parallel solution, resulting in a path in a function that can lead to iteration.
Matrix	A rank-2 datum (two dimensions).
MCS	Message Control System (data communications control system, one of which is APL).
Mode	System interpretation of transaction entry: execution, function definition and editing, evaluated, or character input. Recognized by prompt.
Monadic Function	A function having only one (explicit) argument (always right argument).
N-Dimensional	A rank-N array--see vector, matrix.
Name	An identifier used to denote a variable, defined function, group, local name, or a label.
Niladic Function	A function having no (explicit) argument.
Null	File component or list element without value (contrast with empty).

Term	Meaning
Numeric	Type of datum consisting of only numbers, has subtypes integer and Boolean; in inclusive range for mantissa magnitude 0 to (2*39)-1 and exponent (8*-63) to 8*63.
Operator	On defined functions to produce a new function that applies to arguments.
Orthogonal	Mutually perpendicular, or independent; referring to different dimensions of an array.
Password	User selected name for access control of account, workspace, or file.
Pendant Function	A function that is awaiting completion of another function that it called.
Plane	Any "slice" of a shaped object that is orthogonal to a given dimension of that object. A plane "across" the K-th dimension of an N-dimensional object is a (N¯1)-dimensional object with all but the K-th dimension of the original retained. Thus a "plane" of a vector is a scalar element, and a "plane" of a matrix is a vector from a row or column.
Primitive Function	Any of the functions supplied as part of the APL language.
Prompt (system)	A displayed response (from APL) that identifies the mode. The terminal is unlocked to accept user entry following a prompt.
Qualification	Specification of dimension for application of function, or name for function editing.
Range	Allowable set of values for result of applying a function.
Rank	The number of dimension of a data object. Scalars are rank 0, vectors are rank 1, matrices are rank 2, and n-dimensional arrays are rank n.
Recovery	Restoration of the work in progress after an interrupted work session.
Scalar	A data object without shape; that is, a rank-0 data object; may be either a number or character.
Scalar Primitive	Function applied element by element to its argument(s).
Selection	Specify a subarray by providing a list of indices.
Set	Unique values in data object independent of shape or order.

Term	Meaning
Shape	A vector specifying the number of planes across each dimension of a data object with positive rank. Arrays have shape, scalars do not.
Shared Variable	A system variable that is shared between a user and another user or process external to APL.
Single	A data object of any rank with only one element.
Size	The scalar number of elements in an array.
State Indicator	Record of user defined functions in process, suspended, or pending completion of other called functions.
String	A character type data object that may be either a scalar or vector.
Subscript List	List of expressions or nulls, one for each dimension of an array data object.
Surrogate	A substitute, or external name, for shared-variable reference.
Suspended Function	A function whose execution was stopped for some reason other than a call to another function.
Symbol Table Entry	Any of the set of distinct names and numeric constants occurring in a workspace.
System Commands	Execution Mode commands with ')' prefix that provide environment controls and interrogation facilities.
System Functions	Functions with ⎕ prefix that provide executable controls and inquiry capabilities regarding the environment.
System Variables	Variables shared with APL/700 to specialize processing within a workspace (index origin, print precision, comparison tolerance, and random link).
Template	Specification of name and call syntax of defined function.
Text	Any string of characters.
Transaction	Cycle consisting of user entry, APL processing (and display of output and prompt as required), and unlock of keyboard.
Type	Either character or numeric, of data object.

Term	Meaning
Value	The scalar element or array of elements of a data object, each in the domain for the type of the data object.
Variable	Data object attached to a name by assignment and used for reference.
Vector	A rank-1 datum.
Workspace	The maximum space made available by the APL installation for direct access by an application. See Active Workspace, Inactive Workspace.

APPENDIX B

WORKSPACE CONTENT SPACE CONSIDERATIONS

USE OF SPACE.

The user workspace size is limited to the maximum number of bytes established by the installation. The system function ⎕*WA* provides the amount of space remaining and the amount in use. In a clear workspace, there is some space in use for workspace management and for the user symbol table. As functions, variables and groups are created, the space remaining decreases. The space remaining is used also for temporary results of computations. The available space is augmented by release of unneeded objects: automatically for temporary results, local names, or a prior data object attached to a name on replacement; explicitly for other named objects. Since the total available space is limited, some consideration of space consumption may be required in large applications.

SYMBOL TABLE.

The symbol table is used to provide convenient reference to names, and to literal constants and comments in user-defined functions. Each symbol table entry requires 6 bytes, whether or not the entry actually refers to anything. The user can control the maximum symbol table size in a clear workspace using either:

)*SYMS* N	establish default as N symbols
)*CLEAR* N	override default to become N symbols

The user can interrogate the current symbol table size by: ⎕*NA*.

NAMES.

Each entry in the symbol table referring to a name contains the means to recover the corresponding name supplied by the user. The space required (once per name) depends on the number of characters in the name:

Characters in name	Extra bytes
1, 2 or 3	0 (stored in entry)
X, more than 3	12+6×⌈*X*÷6

VARIABLES.

Each data object has an overhead of 12 bytes. Also, each requires space to describe the structure and to contain the values of its elements. The space for structure description depends on the rank.

Rank R	Extra Bytes
Scalar 0	0
Vector 1	0
Matrix 2	6
Array 3 or more	6+6×*R*

The space for N elements, regardless of shape, depends on the type:

Type	Bytes
Boolean	6×⌈*N*÷32
Numeric, not Boolean	6×*N*
Character	6×⌈*N*÷6

FUNCTION DEFINITION.

The space for function definition occurs only once in a workspace.

Each line of a user-defined function requires 18 bytes overhead. Each local name, argument or label requires 6 bytes.

Upon initial definition, line editing, or upon fixing a variable to become a function, the internal representation of the function is a token stream. Each name, constant primitive function or operator, file operator, system command or variable, punctuation, literal or comment is a token. Each token requires 2 bytes. Each constant also requires the space for the corresponding data object. Each comment requires space for the text string.

Upon first execution of any line of a function, the internal representation of that line is converted into a process stream that provides a parenthesis-free reordering suitable for direct elaboration. The process stream is generally more compact than the token stream. The process stream representation is maintained until the line is edited.

DEFINED FUNCTION CALL.

Each dynamic instance of a function call (appearing in the state indicator) requires space for all instances of locals:

Local name	12 bytes
Result	12 bytes
Label	18 bytes
Argument	enough for copy of data object if a variable name

Thus, significant space consumption can result from having earlier instances of functions suspended or pending in the state indicator.

Reassignment to an argument changes the initial space allocation, just as with any other variable.

The space indicated for local names is the minimum requirement at function entry when they have no meaning. As they gain value as variables by assignment, or as functions by fixing, more space is required. The amount is determinable as the sum of the individual space requirements as indicated before for the various kinds of names.

FUNCTION REPRESENTATION SPACE COMPARISON.

Typical relative space requirements are indicated below, assuming most names are 3 characters or less, and few comments are included.

Representation		Typical size ratio
vector	data	1.0
canonic	data	2.7
token stream	executable function	2.4
process stream	executed function	2.3

The overhead per line for the function forms is more than the fully expanded names of the vector data representation. The appended blanks in the canonic representation become a major part; particularly if a function has a large local names list, or lines of greatly varying length.

Note that fixing a vector representation may require more space than the original, and that some space is reclaimed by first execution. The space for a comment (a string of characters) is constant in all representations (except canonic where comments that do not increase the length of the longest line take no extra space).

LOCAL AND GLOBAL NAMES.

Any name local to a defined function shields any global meaning of that name. The space the global object is also required, even though inaccessible until exit from the function shielding it.

GROUPS.

Each group name takes 12 bytes. In addition, a group with N names attached requires 6×⌈*N*÷4 bytes.

SHARED VARIABLES.

If there are any shared variable offers outstanding, 12 bytes are required. In addition each shared name takes 6 bytes.

TEMPORARY RESULTS.

Any data object created as a result of expression elaboration requires space for its elements and description as indicated for a variable above. This space is relinquished when the function for which it is an argument has been executed.

SPACE SAVING TECHNIQUES.

Clear the state indicator of unnecessary pending functions.

Expunge or erase unnecessary variables or functions.

Limit the space for unnecessary positions in the symbol table by copying into a clear workspace having only the necessary positions.

Recover the space for local variables or local functions fixed therein by exiting the function to which they are local.

Call common defined functions rather than repeat expressions contained therein.

Attach a scalar to a variable name, replacing a large named data object that is no longer needed.

Hold large inactive data objects in file components. Enough space must exist in the workspace to accept a component. After a file write of a large variable, it may be necessary to assign a scalar to that variable name to free enough space before another file component can be read, even to the same name.

Keep functions not immediately required in vector form as file components. Use a cover function that reads and fixes necessary functions from file components as needed, and expunges them when no longer needed. Exit from the cover function automatically recovers the space for such functions if their names are local.

Minimize the number of lines in a function at the expense of writing more complex expressions.

Use Boolean data objects where appropriate instead of numerics. Arithmetic functions applied to Booleans cause conversion to numeric representation. A numeric data object N known to have only values 0 and 1 can be converted to Boolean by *N*←1=*N*.

Pack several numeric values with limited domains into a single number.

Adapt processing algorithm to space available. Trade iterative processing on sub-arrays for space required for parallel processing on the entire arrays.

Avoid reduction of the result of an outer product operator where inner product will suffice.

Consider using a global variable rather than an argument to a defined function to avoid creating a copy of the argument if always called using the same variable name.

Develop parallel functions for documentation. All comments can be placed therein. A frequently used convention is to put the documentation in another function with "HOW" as a suffix to the executable function name. The documentation can be erased easily if all such names are included in a group.

APPENDIX C

REFERENCE CHARTS

Much of the material detailed in the body of this report is presented here in the form of summary reference charts. These charts are intended for review, once the complete development has been absorbed. They may also be used as a quick indication of the power of APL/700 constructs.

The subjects covered in these charts are:

- dyadic and monadic scalar primitive functions
- primitive operators on dyadic scalar primitive functions
- mixed functions
- primitive file functions
- function definition and editing actions

Also, four APL syntax summary pages are provided for quick reference.

Finally, a condensation is included of the transaction cycle, editing, and the attention conventions.

DYADIC SCALAR PRIMITIVE FUNCTIONS A ● B		● ●		MONADIC SCALAR PRIMITIVE FUNCTIONS ● B
DEFINITION OR EXAMPLE	NAME		NAME	DEFINITION OR EXAMPLE
LARGER OF A AND B ↔ A⌈B 7 ↔ 3⌈7 6.01 ↔ 6.01⌈6.01 ¯3 ↔ ¯3⌈¯7	MAXIMUM	⌈	CEILING	SMALLEST INTEGER NOT LESS THAN B ↔ ⌈B 4 ↔ ⌈3.141 ¯3 ↔ ⌈¯3.141 101 ↔ ⌈101
SMALLER OF A AND B ↔ A⌊B 3 ↔ 3⌊7 6.01 ↔ 6.01⌊6.01 ¯7 ↔ ¯3⌊¯7	MINIMUM	⌊	FLOOR	LARGEST INTEGER NOT GREATER THAN B ↔ ⌊B 3 ↔ ⌊3.141 ¯4 ↔ ⌊¯3.141 101 ↔ ⌊101
1.5 ↔ ¯2+3.5 5.5 ↔ 2+3.5 ¯1.5 ↔ 2+¯3.5	ADD	+	IDENTITY	0+B ↔ +B 3.5 ↔ +3.5 ¯3.5 ↔ +¯3.5
¯1.5 ↔ 2-3.5 1.5 ↔ 3.5-2 5.5 ↔ 2-¯3.5	SUBTRACT	-	NEGATE	0-B ↔ -B ¯3.5 ↔ -3.5 3.5 ↔ -¯3.5
5 ↔ 4×1.25 ¯3 ↔ 6×¯.5 0 ↔ 0×¯.09	MULTIPLY	×	SIGNUM	SIGN OF B: 1 ↔ ×7.2 0 ↔ ×0 ¯1 ↔ ×¯3
1.76 ↔ 3.52÷2 ¯5 ↔ 10÷¯2 4 ↔ 12÷3	DIVIDE	÷	RECIPROCATE	1÷B ↔ ÷B .5 ↔ ÷2 ¯2 ↔ ÷¯.5
3 ↔ 5|13 B-A×⌊B÷A ↔ A|B FOR A≠0 5 ↔ 0|5 B ↔ A|B FOR A=0 5 ↔ 14|5 .14 ↔ 1|3.14 4 ↔ 5|¯11 ¯1 ↔ ¯4|7	RESIDUE	|	MAGNITUDE	ABSOLUTE VALUE OF B ↔ |B 9.5 ↔ |9.5 9.5 ↔ |¯9.5 0 ↔ |0
A RAISED TO THE POWER B: 9 ↔ 3*2 1024 ↔ 2*10 2 ↔ 4*.5 ¯3 ↔ ¯27*(÷3)	POWER	*	BASE E POWER	(2.71828...)*B 4 ↔ *1.386294361... 20.0855... ↔ *3
(●B)÷●A ↔ LOGARITHM OF B FOR BASE A ↔ A●B 1.87506... ↔ 10●75 3 ↔ 2●8	LOGARITHM	●	BASE E LOGARITHM	(2.71828...)●B N ↔ *●N ↔ ●*N 1.386294361... ↔ ●4 ¯.693147... ↔ ●.5
0●0 1●0 0●1 1●1 3●4 3●3 4●3				
0 0 1 0 1 0 0	LESS	<		
1 0 1 1 1 1 0	NOT GREATER	≤		
1 0 0 1 0 1 0	EQUAL	=		
1 1 0 1 0 1 1	NOT LESS	≥		
0 1 0 0 0 0 1	GREATER	>		
0 1 1 0 1 0 1	NOT EQUAL	≠		
0●0 0●1 1●0 1●1 BOOLEAN DOMAIN (0 OR 1)		~	NOT	0 ↔ ~1 1 ↔ ~0 BOOLEAN DOMAIN (0 OR 1)
0 0 0 1	AND	∧		
0 1 1 1	OR	∨		
1 1 1 0	NAND	⍲		
1 0 0 0	NOR	⍱		
(1-B*2)*.5 ↔ 0○B ARCSIN B ↔ ¯1○B SIN B ↔ 1○B ARCCOS B ↔ ¯2○B COS B ↔ 2○B ARCTAN B ↔ ¯3○B TAN B ↔ 3○B (¯1+B*2)*.5 ↔ ¯4○B (1+B*2)*.5 ↔ 4○B ARCSINH B ↔ ¯5○B SINH B ↔ 5○B ARCCOSH B ↔ ¯6○B COSH B ↔ 6○B ARCTANH B ↔ ¯7○B TANH B ↔ 7○B	CIRCULAR	○	PI TIMES	B×3.14159... ↔ ○B 6.283185... ↔ ○2
6 ↔ 2!4 (!B)÷(!A)×!B-A FOR A≤B 0 ↔ 9!3 1 ↔ 5!5 0 ↔ A!B FOR A>B ¯10 ↔ 3!¯3 4.9346... ↔ 1.1!4.5	COMBINA- TORIAL	!	FACTORIAL	B×!B-1 ↔ !B FOR B≥1, B AN INTEGER; 1 ↔ !0 GAMMA(B+1) ↔ !B FOR NON-INTEGER B 6 ↔ !3 39916800 ↔ !11 2.68344... ↔ !2.3 3.3283... ↔ !¯2.3

PRIMITIVE OPERATORS ON DYADIC SCALAR PRIMITIVE FUNCTIONS

NAME	FORM	DEFINITION	EXAMPLES
REDUCTION	●/B	B VECTOR: SCALAR RESULT IS FORMED BY ELABORATING THE APL EXPRESSION FORMED BY PLACING ● BETWEEN THE ELEMENTS OF THE VECTOR. IF B IS AN EMPTY VECTOR THE RESULT IS THE IDENTITY ELEMENT FOR ● IF IT EXISTS. B ARRAY: RESULT IS FORMED BY REDUCING VECTORS ON THE LAST DIMENSION OF THE ARRAY. THE RESULT HAS RANK 1 LESS THAN THE RANK OF THE ARGUMENT. THE SHAPE OF THE RESULT IS THE SAME AS THE SHAPE OF THE ARGUMENT LESS THE LAST DIMENSION. B SCALAR: THE RESULT IS THE SCALAR B. B MUST BE IN THE DOMAIN OF ●.	6 ↔ +/1 2 3 1.4 ↔ -/2.3 5.6 4.7 1 ↔ ×/⍳0 ¯4.31...E68 ↔ ⌈/⍳0 1.5 4.8 7.875 ↔ ÷/3 3⍴⍳9 5 ↔ */5
	●/[K]B	LIKE ● BUT VECTORS ON THE K TH DIMENSION ARE REDUCED.	1.75 3.2 4.5 ↔ ÷/[1]3 3⍴⍳9
	●⌿B	●⌿B ↔ ●/[1]B REDUCTION ON THE FIRST DIMENSION.	1.75 3.2 4.5 ↔ ÷⌿3 3⍴⍳9 6 ↔ +⌿1 2 3
	●⌿[K]B	●⌿[K]B ↔ ●/[1+(⍴⍴B)-K]B REDUCTION ON K TH FROM LAST DIMENSION.	1.5 4.8 7.875 ↔ ÷⌿[1]3 3⍴⍳9
SCAN	●\B	B VECTOR: RESULT IS A VECTOR OF THE SAME LENGTH WHOSE I TH ELEMENT IS ●/I↑B. B ARRAY: RESULT IS FORMED BY REPLACING VECTORS ON THE LAST DIMENSION OF B BY THE ● SCAN OF THE VECTOR IN B. B SCALAR: THE RESULT IS THE SCALAR B. B MUST BE IN THE DOMAIN OF ●.	1 3 6 ↔ +\1 2 3 2.3 ¯3.3 1.4 ↔ -\2.3 5.6 4.7 1 0.5 1.5 4 0.8 4.8 ↔ ÷\3 3⍴⍳9 7 0.875 7.875 1 ↔ ∧\1
	●\[K]B	LIKE ●\ BUT VECTORS ON THE K TH DIMENSION ARE SCANNED.	1 2 3 0.25 0.4 0.5 ↔ ÷\[1]3 3⍴⍳9 1.75 3.2 4.5
	●⍀B	●B ↔ ●\[1]B SCAN ON THE FIRST DIMENSION.	1 2 3 0.25 0.4 0.5 ↔ ÷⍀3 3⍴⍳9 1.75 3.2 4.5
	●⍀[K]B	●⍀[K]B ↔●\[1+(⍴⍴B)-K]B SCAN ON THE K TH FROM LAST DIMENSION.	1 0.5 1.5 4 0.8 4.8 ↔ ÷⍀[1]3 3⍴⍳9 7 0.875 7.875
INNER PRODUCT	A●.●B	ELEMENTS OF THE RESULT ARE FORMED BY TAKING CONFORMING VECTORS ON THE LAST DIMENSION OF A AND THE FIRST DIMENSION OF B APPLYING ● BETWEEN THEM AND REDUCING THE RESULT BY ●. M1+.×M2 IS THE LINEAR ALGEBRA MATRIX PRODUCT FOR MATRICES M1 AND M2.	32 ↔ 1 2 3+.×4 5 6 1 ↔ 1 0 1∨.∧1 1 0 5 6 7 8 ↔ (2 3⍴⍳6)-.⌈3 4⍴⍳12 8 8 8 8
OUTER PRODUCT	A∘.●B	THE RESULT IS THE OPERATOR ● APPLIED BETWEEN ALL PAIRS OF ELEMENTS SELECTED FROM A AND B. THE RESULT HAS SHAPE (⍴A),⍴B.	4 5 8 10 ↔ 1 2 3∘.×4 5 12 15 0 1 1 1 ↔ 0 1∘.∨0 1

● AND ● ARE ANY DYADIC SCALAR PRIMITIVE FUNCTIONS: ⌈ ⌊ + - × ÷ | * ● < ≤ = ≥ > ≠ ∧ ∨ ⍲ ⍱ ○ !
K IS A DIMENSION NUMBER OF B: K∊⍳⍴⍴B

MIXED PRIMITIVE FUNCTIONS- 1

NAME	FORM	DEFINITION	EXAMPLE
SHAPE	⍴B	SHAPE PRODUCES A VECTOR WHICH IS THE SHAPE OF THE ARGUMENT. ,A ↔ ⍴A⍴B	,5 ↔ ⍴¯2 ¯1 0 1 2 2 3 4 ↔ ⍴2 3 4⍴⍳24 ⍳0 ↔ ⍴'A'
INTEGERS IN	⍳B	B MUST BE A NON-NEGATIVE INTEGER SCALAR. THE RESULT IS A VECTOR OF LENGTH B OF THE FIRST B INTEGERS STARTING AT THE INDEX ORIGIN. ⍳0 ↔ THE EMPTY NUMERIC VECTOR. ⍳N ↔ (⍳N-1),N IN ORIGIN 1.	1 2 3 4 5 ↔ ⍳5 ,1 ↔ ⍳1 0⍴0 ↔ ⍳0
INDEX	A⍳B	A MUST BE A VECTOR. THE RESULT IS A DATA OBJECT WITH THE SAME SHAPE AS B. EACH ELEMENT OF THE RESULT IS THE INDEX IN A OF THE THE FIRST OCCURENCE OF THE CORRESPONDING ELEMENT IN B. IF THE ELEMENT DOES NOT OCCUR IN A THE RESULT IS 1+⍴A(IN ORIGIN 1, ⍴A IN ORIGIN 0).	3 ↔ 4 7 10 22⍳10 1 2 1 3 ↔ 'ABCABCDE'⍳'ABAC' 4 2 4 1 ↔ ¯1 0 1⍳10 0 ¯16 ¯1 4 4 4 ↔ 'ABC'⍳1 2 3
DEFAULT FORMAT	⍕B	THE RESULT IS A CHARACTER DATA OBJECT WITH THE SAME SHAPE AS B EXCEPT THE LAST DIMENSION IS EXPANDED. THE RESULT IS A CHARACTER REPRESENTATION OF B.	'1 2 3' ↔ ⍕1 2 3 'APL' ↔ ⍕'APL'
FORMAT	A⍕B	SEE FORMAT CHART.	
EVALUATE	⍎B	B MUST BE A CHARACTER STRING WHICH IS A VALID APL EXPRESSION. THE RESULT OF EVALUATE IS THE RESULT PRODUCED FROM THE EVALUATION OF THE EXPRESSION IF IT PRODUCES A RESULT. IF THE EXPRESSION DOES NOT PRODUCE A RESULT EVALUATEMUST BE THE LEFTMOST FUNCTION IN THE EXPRESSION.	4 ↔ ⍎'2+2' 1 2 3 4 5 ↔ ⍎'⍳5' 'APL' ↔ ⍎'''APL''' ¯2 ¯1 0 1 2 ↔ ⍎⍕¯2 ¯1 0 1 2
MEMBERSHIP	A∊B	A DETERMINES THE SHAPE OF THE BOOLEAN RESULT. EACH ELEMENT IS 1 IF PRESENT IN B, 0 OTHERWISE. A∊B ↔ ∨/A∘.=,B	1 0 1 1 ↔ 1 2 3 1∊1 3 5 0 1 1 1 0 ↔ 'LEARN'∊'TEACHER'
SUBSET	A⊂B	THE BOOLEAN SCALAR RESULT IS 1 IF ALL UNIQUE ELEMENTS OF A ALSO APPEAR IN B, 0 OTHERWISE. A⊂B ↔ ∧/,A∊B	1 ↔ 1 2⊂3 2 1 0 ↔ 'A'⊂3 0 ↔ 1 2⊂ 3 1 ↔ 'PAOLI'⊂'PLATONIC'
SUPERSET	A⊃B	THE BOOLEAN SCALAR RESULT IS 1 IF ALL UNIQUE ELEMENTS OF B ALSO APPEAR IN A, P OTHERWISE. A⊃B ↔ ∧/,B∊A	0 ↔ 1 2⊃4 3 2 1 0 ↔ 'A'⊃3 0 ↔ 'PAOLI'⊃'PLATONIC'
UNION	A∪B	THE VECTOR RESULT IS THE UNIQUE ELEMENTS FROM A OR B IN THE ORDER OF FIRST OCCURRENCE IN (,A),,B.	1 4 ↔ 1 1∪4 1 1 3 ↔ 1 1 3 1∪⍳0 'MANGET' ↔ 'MANAGEMENT'∪''
INTERSECTION	A∩B	THE VECTOR RESULT IS THE UNIQUE ELEMENTS OCCURRING IN BOTH (,A) AND (,B) IN THE ORDER THEY FIRST OCCUR IN A.	2 3 ↔ 1 2 3∩2 3 4 'HAR' ↔ 'HARRY'∩'MARTHA'
EXCLUSION	A~B	THE VECTOR RESULT IS THE UNIQUE ELEMENTS OCCURRING IN A BUT NOT IN B, IN THE ORDER THEY FIRST OCCUR IN A.	,1 ↔ 1 2 3~ 2 3 4 'SET' ↔ 'SETTLED'~'LAND'

MIXED PRIMITIVE FUNCTIONS - 2

NAME	FORM	DEFINITION	EXAMPLE
REPRESENT	A⊤B	B SCALAR: IF A IS A VECTOR THE RESULT IS A VECTOR THE SAME LENGTH AS A. THE RESULT CONTAINS THE REPRESENTATION OF B IN THE NUMBER SYSTEM A. IF A IS AN ARRAY THEN THE RESULT IS THE REPRESENTATION OF B IN THE NUMBER SYSTEMS SPECIFIED BY VECTORS ALONG THE FIRST DIMENSION OF A. B ARRAY: THE RESULT WILL BE A DATA OBJECT WITH SHAPE (⍴A),⍴B WHERE VECTORS ALONG THE FIRST DIMENSION OF THE RESULT ARE THE REPRESENTATION OF A SCALAR IN B IN THE NUMBER SYSTEM SPECIFIED BY A VECTOR ALONG THE FIRST DIMENSION OF A. FUNCTIONS IN A MANNER SIMILAR TO OUTER PRODUCT.	1 0 1 ↔ 2 2 2⊤5 0 26 23 ↔ 24 60 60⊤1583 1 0 0 3 ↔(3 2⍴4 5)⊤17 1 2 1 1 0 0 0 1 1 0 ↔ 2 2 2⊤4 7 3 0 0 1 1 0
BASE VALUE	A⊥B	B VECTOR: IF A IS A VECTOR THEN THE RESULT IS A SCALAR WHICH IS THE BASE 10 VALUE OF THE VECTOR IN THE NUMBER SYSTEM SPECIFIED BY A. A MAY BE A SCALAR IN WHICH CASE IT IS EXTENDED TO THE LENGTH OF B. IF A IS AN ARRAY THE RESULT HAS SHAPE ¯1↓⍴A AND CONTAINS THE REPRESENTATION IN BASE 10 OF B IN THE NUMBER SYSTEM SPECIFIED BY A VECTOR ALONG THE LAST DIMENSION OF A. B ARRAY: THE RESULT IS AN ARRAY WITH SHAPE (¯1↓⍴A),1↓⍴B. THE RESULT IS SCALARS WHICH ARE THE BASE 10 REPRESENTATION OF VECTORS ALONG THE FIRST DIMENSION OF B IN THE NUMBER SYSTEMS SPECIFIED BY VECTORS ALONG THE LAST DIMENSION OF A. FUNCTIONS IN A MANNER SIMILAR TO INNER PRODUCT.	5 ↔ 2 2 2⊥1 0 1 1583 ↔ 24 60 60⊥0 26 23 15 ↔ 2⊥1 1 1 1 22 30 38 ↔ (3 2⍴5 5 7 7 9 9)⊥4 2 4 6 ↔ 2 2 2⊥3 2⍴1 1 0 1 0 0
MATRIX INVERSE	⌹B	B MUST BE A MATRIX WITH NO MORE COLUMNS THAN ROWS. THE RESULT IS THE INVERSE OR GENERALIZED INVERSE OF THE MATRIX IF IT EXISTS.	3.5 ¯1.5 0.5 ¯4 2 ¯1 ↔ ⌹3 3⍴(4⍴1),2 3 ¯2 ¯1 2 1.5 ¯0.5 0.5
MATRIX DIVIDE	A⌹B	B MUST BE A MATRIX WITH NO MORE COLUMNS THAN ROWS. A IS EITHER A VECTOR WITH LENGTH EQUAL TO THE NUMBER OF ROWS IN B OR A MATRIX WITH THE SAME NUMBER OF ROWS AS B. THE RESULT IS THE SOLUTION TO THE SYSTEM OF LINEAR EQUATIONS WITH COEFFICIENT MATRIX B AND RIGHT HAND SIDE(S) A IF IT EXISTS. WHEN B HAS MORE ROWS THAN COLUMNS THE RESULT IS A LEAST SQUARES FIT FOR THE SYSTEM.	¯1 1 ↔ 0 ¯1⌹2 2⍴1 1 2
GRADE-UP	⍋B	B MUST BE A NUMERIC VECTOR. THE RESULT IS A SET OF INDICES THAT CAN BE USED TO ORDER B IN ASCENDING ORDER.	2 5 4 1 3 ↔ ⍋8 0 9 5 0 0 0 5 8 9 ↔ 8 0 9 5 0[⍋8 0 9 5 0]
GRADE-DOWN	⍒B	B MUST BE A NUMERIC VECTOR. THE RESULT IS A SET OF INDICES THAT CAN BE USED TO ORDER B IN DESCENDING ORDER.	3 1 4 2 5 ↔ ⍒8 0 9 5 0 9 8 5 0 0 ↔ 8 0 9 5 0[⍒8 0 9 5 0]
ROLL	?B	B MUST CONTAIN POSITIVE INTEGERS. THE RESULT IS A DATA OBJECT LIKE B WITH EACH ELEMENT A RANDOM CHOICE FROM ⍳S WHERE S IS THE CORRESPONDING ELEMENT OF B.	1 1 ↔ ?1 1
DEAL	A?B	A AND B MUST BE NON-NEGATIVE INTEGERS WITH A NOT GREATER THAN B. THE RESULT IS A VECTOR OF LENGTH A THE ELEMENTS OF THE RESULT ARE A RANDOM SELECTION WITHOUT REPLACEMENT FROM ⍳B.	,1 ↔ 1?1 ⍳0 ↔ 0?10

MIXED PRIMITIVE FUNCTIONS FOR STRUCTURING - 1

THE RIGHT ARGUMENT OF ANY STRUCTURE MIXED PRIMITIVE FUNCTIONS MAY BE A CHARACTER DATA OBJECT. SINCE CATENATE AND LAMINATE JOIN TWO DATA OBJECTS, IF THE RIGHT ARGUMENT IS A CHARACTER DATA OBJECT THE LEFT ARGUMENT MUST ALSO BE ONE. ALL OTHER STRUCTURE MIXED PRIMITIVE FUNCTIONS FUNCTION IN THE SAME MANNER ON CHARACTER DATA OBJECTS AS ON NUMERIC DATA OBJECTS. FILL FOR TAKE AND EXPAND IS BLANKS IF THE RIGHT ARGUMENT IS A CHARACTER DATA OBJECT.

THE FOLLOWING VARIABLES ARE USED IN THE EXAMPLES:

```
                                 111 112 113 114
   1 2 3 4 5 ↔ V                 121 122 123 124
                                 131 132 133 134
                                                    ↔ T
 11 12 13 14                     211 212 213 214
 21 22 23 24 ↔ M                 221 222 223 224
 31 32 33 34                     231 232 233 234
```

NAME	FORM	DEFINITION	EXAMPLES
RESHAPE	AρB	THE DATA OBJECT B IS MADE INTO THE SHAPE SPECIFIED BY A. IF B HAS LESS ELEMENTS THAN ARE NEEDED THE ELEMENTS OF B ARE REUSED UNTIL ENOUGH ELEMENTS ARE OBTAINED. IF B HAS MORE ELEMENTS THAN ARE NEEDED THE EXCESS ARE IGNORED. ,A ↔ ρAρB	5 5 5 ↔ 3ρ5 ,1 ↔ 1ρV 2.5 ↔ (ι0)ρ2.5 8.6 ¯3.1 1 2 3 4 ↔ 3 2ρV 5 1
RAVEL	,B	THE DATA OBJECT B IS RESHAPED INTO A VECTOR. ,B ↔ (×/ρB)ρB	11 12 13 14 21 22 23 24 31 32 33 34 ↔ ,M 1ρ8.6 ↔ ,8.6
CATENATE	A,B	THE DATA OBJECTS A AND B ARE JOINED TOGETHER TO FORM A NEW DATA OBJECT. THE DATA OBJECTS ARE JOINED ALONG THE LAST DIMENSION. A SCALAR IS EXTENDED TO FORM A PLANE ACROSS THE DIMENSION IT IS BEING JOINED TO.	1 2 3 4 5 1 2 3 4 5 ↔ V,V 7 1 2 3 4 5 ↔ 7,V 7 11 12 13 14 8 21 22 23 24 ↔ 7 8 9,M 9 21 22 23 24 11 12 13 14 1 21 22 23 24 1 ↔ M,1 31 32 33 34 1
	A,[K]B	LIKE A,B BUT THE DATA OBJECTS ARE JOINED ON THE K TH DIMENSION.	11 12 13 14 21 22 23 24 ↔ M,[1]7 8 9 10 31 32 33 34 7 8 9 10
LAMINATE	A,[K]B	THE DATA OBJECTS A AND B ARE JOINED ALONG A NEW DIMENSION. K MUST BE NON-INTEGRAL AND BETWEEN THE NUMBERS OF THE DIMENSIONS BETWEEN WHICH THE NEW DIMENSION IS FORMED. A SCALAR IS EXTENDED TO THE SHAPE OF THE OTHER OBJECT.	1 100 2 200 ↔ 1 2 3,[1.5]100 200 300 3 300 1 2 3 4 5 8 8 8 8 8 ↔ V,[.476]8
REVERSE	⌽B	B VECTOR: THE ORDER OF THE ELEMENTS IN B IS REVERSED. B ARRAY: THE VECTORS ON THE LAST DIMENSION OF B ARE REVERSED. B SCALAR: NO ACTION OCCURS WHEN B IS A SCALAR.	5 4 3 2 1 ↔ ⌽V 14 13 12 11 24 23 22 21 ↔ ⌽M 34 33 32 31 1.5 ↔ ⌽1.5
	⌽[K]B	SAME AS ⌽B BUT VECTORS ON THE K TH DIMENSION ARE REVERSED.	31 32 33 34 21 22 23 24 ↔ ⌽[1]M 11 12 13 14
	⊖B	⊖B ↔ ⌽[1]B REVERSAL ALONG THE FIRST DIMENSION.	31 32 33 34 21 22 23 24 ↔ ⊖3 3ρM 11 12 13 14
	⊖[K]B	⊖[K]B ↔ ⌽[1+(ρρB)-K]B REVERSAL ALONG THE K TH FROM LAST DIMENSION.	14 13 12 11 24 23 22 21 ↔ ⊖[1]M 34 33 32 31

MIXED PRIMITIVE FUNCTIONS FOR STRUCTURING - 2

NAME	*FORM*	*DEFINITION*	*EXAMPLES*
ROTATE	A⌽B	*B VECTOR: THE ELEMENTS OF THE VECTOR ARE ROTATED TO THE LEFT CYCLICALLY (⍴B)|A POSITIONS.* *B ARRAY: VECTORS ON THE LAST DIMENSION OF B ARE ROTATED BY ARE THE AMOUNT SPECIFIED BY THE CORRESPONDING ELEMENT IN A. A MUST BE AN ARRAY OF RANK ONE LESS THAN THE RANK OF B AND SHAPE SAME AS B LESS THE LAST ELEMENT. A MAY BE A SCALAR IN WHICH CASE IT SPECIFIES THE ROTATION FOR ALL VECTORS.* *B SCALAR: NO OPERATION IS PERFORMED IF B IS A SCALAR.*	3 4 5 1 2 ↔ 2⌽*V* 4 5 1 2 3 ↔ ¯2⌽*V* 14 11 12 13 21 22 23 24 ↔ ¯1 0 1⌽*M* 32 33 34 31 12 13 14 11 22 23 24 21 ↔ 5⌽*M* 32 33 34 31 5 ↔ ¯8⌽5
	A⌽[K]B	*LIKE A⌽B BUT VECTORS ON THE K TH DIMENSION ARE ROTATED.*	31 12 23 34 11 22 33 14 ↔ ¯4 ¯3 ¯2 ¯1⌽[1]*M* 21 32 13 24
	A⊖B	A⊖B ↔ A⌽[1]B *ROTATION ON THE FIRST DIMENSION.*	21 22 23 24 31 32 33 34 ↔ 1⊖*M* 11 12 13 14
	A⊖[K]B	A⊖[K]B ↔ A⌽[1+(⍴⍴B)-K]B *ROTATION ON THE K TH FROM LAST DIMENSION.*	12 13 14 11 23 24 21 22 ↔ 1 2 ¯1⊖[1]*M* 34 31 32 33
TRANSPOSE	⍉B	*B SCALAR: THE RESULT IS B AS A 1 × 1 MATRIX.* *B VECTOR: THE RESULT IS B AS A COLUMN MATRIX (SHAPE ⍴B × 1).* *B ARRAY: THE RESULT IS B WITH THE DIMENSIONS REVERSED.* (⌽⍳⍴⍴B)⍉B ↔ ⍉B *FOR* 2≤⍴⍴B	1 1⍴¯6.3 ↔ ⍉¯6.3 ¯1 11 21 31 0 ↔ ⍉¯1 0 1 12 22 32 ↔ ⍉*M* 1 13 23 33 14 24 34
PERMUTE	A⍉B	*THE DIMENSIONS OF B ARE PERMUTED AS SPECIFIED BY A. THE I TH DIMENSION OF B IS THE A[I] DIMENSION OF THE RESULT. SEVERAL DIMENSIONS OF B MAY BE MAPPED INTO A SINGLE DIMENSION OF THE RESULT TO OBTAIN A DIAGONAL CROSS SECTION OF B. IF A IS THE SAME AS* ⍳⍴⍴B *THEN THE RESULT WILL BE B.*	11 21 31 1 2 3 4 5 ↔ 1⍉*V* 12 22 32 ↔ 2 1⍉*M* 13 23 33 ↔ ⍉*M* 14 24 34 11 22 33 ↔ 1 1⍉*M* 111 121 131 ↔ 1 2 1⍉*T* 212 222 232
COMPRESS	A/B	*B VECTOR: A MUST BE A LOGICAL VECTOR WHOSE LENGTH IS IS THE SAME AS THE LENGTH OF B. THE RESULT HAS LENGTH* +/A. *THE ELEMENTS OF THE RESULT ARE TAKEN FROM B EVERYWHERE A 1 APPEARS IN A. A MAY BE A SCALAR IN WHICH CASE THE RESULT IS B IF A IS* 1 *AND THE EMPTY VECTOR IF A IS* 0. *B ARRAY: VECTORS ON THE LAST DIMENSION OF B ARE COMPRESSED BY A.* *B SCALAR: B IS EXTENDED TO THE LENGTH OF THE VECTOR A AND THEN COMPRESSED BY A.*	1 2 4 ↔ 1 1 0 1 0/*V* 2 3 5 ↔ 0 1 1 0 1/*V* ⍳0 ↔ 0/*V* 1 2 3 4 5 ↔ 1/*V* 12 13 22 23 ↔ 0 1 1 0/*M* 32 33 5 5 5 5 ↔ 1 0 1 1 0 0 1/5 ¯4.5 ¯4.5 ↔ 0 1 0 0 1 0 0 0/¯4.5
	A/[K]B	*LIKE A/B BUT VECTORS ON THE K TH DIMENSION ARE COMPRESSED.*	11 12 13 14 31 32 33 34 ↔ 1 0 1/[1]*M*
	A⌿B	A⌿B ↔ A/[1]B *COMPRESS ON THE FIRST DIMENSION.*	21 22 23 24 31 32 33 34 ↔ 0 1 1⌿*M*
	A⌿[K]B	A⌿[K]B ↔ A/[1+(⍴⍴B)-K]B *COMPRESS ON THE K TH FROM LAST DIMENSION.*	11 12 21 22 ↔ 1 1 0 0⌿[1]*M* 31 32

MIXED PRIMITIVE FUNCTIONS FOR STRUCTURING - 3

NAME	*FORM*	*DEFINITION*	*EXAMPLES*
EXPAND	*A\B*	*B VECTOR: A MUST BE A LOGICAL VECTOR SUCH THAT +/A IS THE SAME AS THE LENGTH OF B. THE RESULT HAS THE SAME LENGTH AS A WHERE SUCCESSIVE ELEMENTS OF B ARE USED WHERE EACH* 1 *APPEARS IN A AND FILL IS INSERTED WHERE EACH* 0 *APPEARS.* *B ARRAY: VECTORS ON THE LAST DIMENSION OF A ARE EXPANDED BY A.* *B SCALAR: B IS EXTENDED TO LENGTH +/A AND THEN EXPANDED BY A.*	0 1 2 0 3 4 5 0 ↔ 0 1 1 0 1 1 1 0\V 11 0 12 13 0 14 21 0 22 23 0 24 ↔ 1 0 1 1 0 1\M 31 0 32 33 0 34 0 0 0 0 7 7 7 0 ↔ 0 0 0 0 1 1 1 0\7
	A\[K]B	*LIKE A\B BUT VECTORS ON THE K TH DIMENSION ARE EXPANDED.*	11 12 13 14 0 0 0 0 21 22 23 24 ↔ 1 0 1 0 1\[1]M 0 0 0 0 31 32 33 34
	A⍀B	*A⍀B ↔ A\[1]B* *EXPANSION ON THE FIRST DIMENSION.*	0 0 0 0 11 12 13 14 21 22 23 24 ↔ 0 1 1 0 1⍀M 0 0 0 0 31 32 33 34
	A⍀[K]B	*A⍀[K]B ↔ A\[1+(⍴⍴B)-K]B* *EXPANSION ON THE K TH FROM LAST DIMENSION.*	0 11 0 12 0 13 14 0 21 0 22 0 23 24 ↔ 0 1 0 1 0 1 1⍀[1]M 0 31 0 32 0 33 34
TAKE	*A↑B*	*B VECTOR: THE RESULT IS THE FIRST(LAST) |A ELEMENTS OF B IF A IS POSITIVE(NEGATIVE). IF |A IS GREATER THAN THE LENGTH OF B THEN FILL IS ADDED AT THE END(BEGINNING) OF B.* *B ARRAY: A MUST BE A VECTOR WHOSE LENGTH IS EQUAL TO THE RANK OF B. THE RESULT OF TAKE IS A CORNER OF THE ARRAY.* *B SCALAR: B WILL BE MADE INTO A ONE ELEMENT OBJECT WITH RANK THE SAME AS THE LENGTH OF A THEN THE TAKE IS DONE ON IT.*	1 2 3 ↔ 3↑V 3 4 5 ↔ ¯3↑V 1 2 3 4 5 0 0 ↔ 7↑V 0 0 1 2 3 4 5 ↔ ¯7↑V 11 12 13 21 22 23 ↔ 3 3↑M 31 32 33 0 0 0 0 0 0 0 0 0 0 11 12 13 14 0 ↔ ¯5 5↑M 21 22 23 24 0 31 32 33 34 0 0 ¯3 0 0 ↔ 2 ¯2↑¯3
DROP	*A↓B*	*B VECTOR:THE RESULT IS B WITH THE FIRST(LAST) |A ELEMENTS OF B REMOVED IF A IS POSITIVE(NEGATIVE). IF |A IS GREATER OR EQUAL TO THE LENGTH OF B THE RESULT IS AN EMPTY VECTOR.* *B ARRAY: A MUST BE A VECTOR WHOSE LENGTH IS EQUAL TO THE RANK OF B. THE RESULT OF DROP IS A CORNER OF THE ARRAY.* *B SCALAR: B WILL BE MADE INTO A ONE ELEMENT OBJECT WITH RANK THE SAME AS THE LENGTH OF A THEN THE DROP IS DONE ON IT.*	4 5 ↔ 3↓V 1 2 ↔ ¯3↓V ⍳0 ↔ 7↓V ⍳0 ↔ ¯7↓V 11 12 21 22 ↔ 0 ¯2↓M 31 32 1 1 1⍴8 ↔ 0 0 0↓8 0 1 0 1⍴¯1.75 ↔ 5 0 1 0↓¯1.75

PRIMITIVE FILE FUNCTIONS

NAME	FORM	DEFINITION
CREATE FILE	⍍F	CREATES THE FILE WITH THE NAME F. CHANGES THE PASSWORD ON F.
RENAME FILE	N⍍F	RENAMES FILE F TO BECOME N.
DESTROY FILE	⍔F	DESTROYS THE FILE F.
COMPONENT WRITE	A⍈[K]F	INSERTS A AS THE K TH COMPONENT OF F.
COMPONENT READ	⍇[K]F	RETURNS THE K TH COMPONENT OF F.
COMPONENT NULL	⊟[K]F	REPLACES THE K TH COMPONENT OF F WITH A NULL COMPONENT (DESTROYS K TH COMPONENT).
COMPONENT FIRST OUT	⍃F	IF NON-NULL, RETURNS THE FIRST COMPONENT OF F AND REMOVES IT FROM F.
COMPONENT LAST OUT	⍄F	IF NON-NULL, RETURNS THE LAST COMPONENT OF F AND REMOVES IT FROM F.
COMPONENT FIRST IN	A⍃F	APPENDS A AS NEW COMPONENT BEFORE COMPONENT POSITIONS ALREAD IN F.
COMPONENT LAST IN	A⍄F	APPENDS A AS NEW COMPONENT AFTER COMPONENT POSITIONS ALREADY IN F.
VALUE MAP	⌸F	RETURNS A BOOLEAN VECTOR WITH LENGTH THE NUMBER OF COMPONENTS IN F.
NULL MAP	⍯F	VALUE MAP RETURNS 1 IF NON-NULL. NULL MAP RETURNS 1 IF NULL.
COMPONENT TAKE	A⍐F	MODIFIES F TO BE THE A TAKE OF F. SIMILAR TO THE TAKE FUNCTION. TAKING MORE COMPONENTS THAN ARE IN THE FILE APPENDS NULL COMPONENTS TO THE FRONT OR END OF F.
COMPONENT DROP	A⍗F	MODIFIES F TO BE THE A DROP OF F. LIKE THE DROP FUNCTION.
REVERSE COMPONENTS	⌼F	REVERSES FILE COMPONENT ORDER IN F. LIKE THE REVERSE FUNCTION.
ROTATE COMPONENTS	A⌼F	MODIFIES F TO BE THE A ROTATE OF F. LIKE THE ROTATE FUNCTION
COMPRESS COMPONENTS	A⍁F	MODIFIES F TO BE THE A COMPRESS OF F. LIKE THE COMPRESS FUNCTION.
EXPAND COMPONENTS	A⍂F	MODIFIES F TO BE THE A EXPAND OF F. LIKE THE EXPAND FUNCTION.
HOLD FILE	⍓F	PLACES A HOLD ON F (PREVENTS OTHER USERS FROM USING F).
FREE FILE	⍌F	REMOVES HOLD ON F (ALLOWS OTHER USERS TO USE F).
RELEASE FILE	⊞F	RELEASE FILE FROM THIS USE.

QUERY FILE	A⌻F	RETURNS INFORMATION ABOUT F:
		A=1 - CURRENT SIZE OF FILE IN BYTES. A=5 - TIMES FILE REORGANIZED.
		A=2 - MAXIMUM SIZE OF FILE IN BYTES. A=6 - ACCOUNTS CURRENTLY USING F.
		A=3 - CURRENT NUMBER OF COMPONENTS. A=7 - TIME OF LAST MODIFICATION.
		A=4 - BOOLEAN 1 IF MODIFIED SINCE BECAME ACTIVE.

FILE USE STATUS	⌻F	RETURNS USAGE STATUS OF FILE F: 0 = FILE F DOES NOT EXIST. 1 = FILE EXISTS AND IS NOT ACTIVE. 4 = FILE IS BEING CLOSED. 2 = FILE IS ACTIVE. 5 = FILE IS HELD BY SOME ACCOUNT.

SYSTEM INTERROGATE	⌻B	RETURNS INFORMATION ABOUT THE FILE SYSTEM:
		B=1 - CURRENT NUMBER OF FILE USERS. B=3 - MAXIMUM NUMBER OF FILE USERS.
		B=2 - CURRENT NUMBER OF ACTIVE FILES. B=4 - MAXIMUM NUMBER OF ACTIVE FILES.

F IS A CHARACTER STRING CONTAINING THE NAME OF THE FILE. THE NAME MAY BE COMPOSED OF FROM 1 TO 12 ALPHANUMERIC CHARACTERS (NO UNDERSCORES) STARTING WITH A LETTER. A LOCK IN BRACKETS MAY FOLLOW THE NAME. IF A FILE ASSOCIATED WITH ANOTHER ACCOUNT IS TO BE ACCESSED THE ACCOUNT NAME IN PARENTHESIS SHOULD PREFIX THE FILE NAME. INDIVIDUAL COMPONENTS MAY HAVE ANY TYPE (CHARACTER OR NUMERIC) AND ANY SHAPE THAT FITS IN THE WORKSPACE.

FUNCTION DEFINITION AND EDITING ACTIONS

COMMAND	ACTION SYMBOL	FORM	ACTION	NEXT PROMPT
	∇	∇H	DEFINE NEW FUNCTION, WITH HEADER H; INITIATE EDITING THEREON.	[1]
OPEN	∇	∇F	INITIATE EDITING OF PREVIOUSLY DEFINED FUNCTION, F.	[Z]
OPEN (LOCKED)	⍫	⍫F	ONLY IF OWNER OF WS AND NOT COPIED	
CLOSE	∇	∇	TERMINATE FUNCTION EDITING. (MAY FOLLOW ANY COMMAND EXCEPT EDIT)	
CLOSE (LOCKED)	⍫	⍫	SO NO COPY OF WS CAN OPEN FUNCTION	
REPLACE		[A]T	TEXT OF LINE A IS REPLACED BY T. (IF A = Z, SAME AS APPEND-AFTER)	[Z]
APPEND-BEFORE	↑	[↑]T	TEXT OF NEW LINE 1 IS T.	
APPEND-AFTER	↓	[↓]T	TEXT OF NEW LAST LINE IS T.	
INSERT-BEFORE	↑	[↑A]T	TEXT OF NEW LINE, TO BE INSERTED BEFORE LINE A, IS T.	[↑A+1]
INSERT-AFTER	↓	[↓A]T	TEXT OF NEW LINE, TO BE INSERTED AFTER LINE A, IS T.	[↓A+1]
FULL-EDIT	∊	[∊A]	INITIATE EDIT OF LINE A. RULES SAME AS FOR TRANSACTION EDIT.	[Z]
PREFIX-EDIT	α	[αA]	SIMILAR TO FULL-EDIT EXCEPT SINGLE INSERTION BEFORE TEXT OF LINE A IS ASSUMED.	
DIRECT-PREFIX	α	[αA]T	THE TEXT T IS INSERTED BEFORE THE TEXT OF LINE A.	
SUFFIX-EDIT	ω	[ωA]	SIMILAR TO FULL-EDIT EXCEPT SINGLE INSERTION AFTER TEXT OF LINE A IS ASSUMED.	
DIRECT-SUFFIX	ω	[ωA]T	THE TEXT T IS INSERTED AFTER THE TEXT OF LINE A	
IMMEDIATE-EDIT	⍳	[⍳A]	UPON TERMINATION OF DEFINITION MODE, TEXT OF A BECOMES THE 'MOST RECENT APL EXPRESSION' AVAILABLE FOR EDIT.	

FUNCTION DEFINITION AND EDITING LINE GROUP ACTIONS

UNQUALIFIED: ALL LINES IN DOMAIN		QUALIFIED: LINES CONTAINING NAME X IN DOMAIN	
FORM	LINE DOMAIN	FORM	LINE DOMAIN
[∘]	0 THRU Y	[(∘X)]	0 THRU Y CONTAINING X
[∘A]	A ONLY	[(∘X)A]	A ONLY IF CONTAINING X
[A∘]	A THRU Y	[A(∘X)]	A THRU Y CONTAINING X
[A∘B]	A THRU B	[A(∘X)B]	A THRU B CONTAINING X

∘ IS ANY MULTILINE FUNCTION EDITING ACTION
A, B ARE LINE NUMBER SPECIFIERS: INTEGER, LABEL OR LABEL∓INTEGER; A≤B
X IS NAME OF LABEL, FUNCTION OR VARIABLE
Y IS NUMBER OF PRESENT LAST LINE

COMMAND	ACTION SYMBOL	ACTION	NEXT PROMPT
SET-TRACE CLEAR-TRACE	⊤ ⊥	FORM OF DISPLAY (DURING EXECUTION): F[N]K(S)V F = FUNCTION NAME N = LINE NUMBER LINE [0] TRACES K = VALUE TYPE FUNCTION RETURN. N - NUMERIC OTHER LINES TRACE B - BOOLEAN LEFTMOST VALUE, C - CHARACTER IF ANY. S = VALUE SHAPE V = VALUE	[Z]
SET-STOP CLEAR-STOP	⌈ ⌊	FORM OF DISPLAY (DURING EXECUTION): F[N]⋆ F = FUNCTION NAME N = LINE NUMBER LINE [0] STOPS BEFORE RETURN.	
SET-MONITOR CLEAR-MONITOR	∩ ∪	INITIATE COLLECTION OF STATISTICS. LINE [0] COUNTS THE NUMBER OF TIMES THE FUNCTION IS EXECUTED.	
DISPLAY-LINES	⎕	FORM OF DISPLAY: HEAD - ∇H H = HEADER BODY - [N] T N = LINE NUMBER TAIL - ∇ T = LINE TEXT	[Z]
DISPLAY-NUMBERS	?	FORM OF DISPLAY: VECTOR OF NUMBERS	
DELETE	~	DELETE THE D SELECTED LINES WITHIN DOMAIN R (DELETE ON LINE ZERO DELETES LOCAL NAMES LIST ONLY).	[Z-D]

BURROUGHS B 6700/B 7700

APL/700 SYNTAX SUMMARY

system commands

session control

)ON	acct [password]*
)OFF	[oldpword/newpword]*
)COFF	[oldpword/newpword]*
)BLOT	

terminal control

)WIDTH	30 thru 32767*
)TABS	0 thru 30*

clear-workspace control

)CLEAR	16 thru 1024*
)SYMS	16 thru 1024*
)ORIGIN	0 or 1*
)DIGITS	1 thru 12*
)SEED	0 to n*
)FUZZ	0 to 1*

library control

)FILES	
)LIB	
)LOAD	wsid
)COPY	wsid nameset*
)PCOPY	wsid nameset*
)SAVE	wsid*
)DROP	own-wsid
)WSID	name*

group control

)ATTACH	groupname nameset*
)DETACH	groupname nameset*
)GRP	groupname

run state

)SI	
)RESET	

name display

)FNS	name*
)VARS	name*
)GRPS	name*
)ERASE	nameset

wsid is (account) name [password]**

** optional field*

argument legend

A	any type
B	Boolean
C	character
D	decimal, numeric
F	character 'name'
I K	integer
L	list
M N	name
P Q	line specifier
R	result
T	text

header for defined function n

template
template local-names-list

no result	result	template
n	R ← n	niladic
n N	R ← n N	monadic
M n N	R ← M n N	dyadic

control structures

	call defined function n
n	niladic
n A	monadic
A n A	dyadic
	sequence of execution
→ I	branch
→	terminate
N:	label
( )	function precedence
;	list separator
⍝	comment

system variables

⎕CT	comparison tolerance
⎕IO	index origin
⎕PP	print precision
⎕RL	random link
⎕	evaluated in, explicit out
⍞	character in, set prompt

shared variable functions

C ⎕SVO C	shared variable offer
⎕SVO C	degree of coupling
B ⎕SVC C	shared variable control
⎕SVC C	control vector
⎕SVQ C	shared variable query
⎕SVR C	shared variable retract

system functions

function representations

	⎕CR	F	canonic represent
	⎕VR	F	vector represent
	⎕FX	C	fix

name

C	⎕NL	I	name list *
	⎕NC	C	name classification
	⎕EX	C	expunge

diagnostic

I	⎕ST	F	set trace *
I	⎕SS	F	set stop *
I	⎕SM	F	set monitor *
I	⎕RT	F	reset trace *
I	⎕RS	F	reset stop *
I	⎕RM	F	reset monitor *
I	⎕MV	F	monitor values *
	⎕QT	F	query trace
	⎕QS	F	query stop
	⎕QM	F	query monitor

execution control

	⎕DL	D	delay
	⎕ED	C	edit
B	⎕ED	C	phrase edit
	⎕ER	C	error

character set

⎕B	backspace
⎕L	linefeed
⎕R	return
⎕T	tab
⎕N	null
⎕A	alphabet
⎕D	digits
⎕AV	atomic vector

status inquiry

⎕PT	print tabs
⎕PW	print width
⎕WI	workspace-i.d.
⎕AN	account name
⎕AI	account information
⎕LC	line counter
⎕TS	time stamp
⎕UL	user load
⎕WA	working availability
⎕NA	name availability
⎕LA	library availability
⎕FA	file availability
⎕SA	shares availability
⎕NEWS	sign-on news

* dyadic - selective
monadic - inclusive

constants

'KEN''S'	character "ken's"
¯1 1.2 3.4E¯7	numeric

transaction editing

meaning of attention

initial:	enter edit cycle
embedded:	correct typing error
terminal:	display next phrase

edit control characters

/	delete
.	mark phrase

function editing actions

∇ M	define
∇ M	open
⍫ M	open (locked)
∇	close
⍫	close (locked)
[P] T	replace
[↑] T	append (before)
[↓] T	append (after)
[↑Q] T	insert (before)
[↓Q] T	insert (after)
[∊Q]	full edit
[⍺Q]	prefix edit
[⍵Q]	suffix edit
[⍳Q]	inject edit

multiline group actions ∘

⊤	set trace*
⊥	reset trace*
⌈	set stop*
⌊	reset stop*
∩	set monitor*
∪	reset monitor*
⎕	display lines*
?	display addresses*
∼	delete*

* unqualified (all lines)

[∘]	0 thru last
[P∘]	P thru last
[∘Q]	Q only
[P∘Q]	P thru Q

qualified (lines with N)

[(∘N)]	0 thru last
[P(∘N)]	P thru last
[(∘N)Q]	Q only
[P(∘N)Q]	P thru Q

identifiers

letter, underscored letter, ∆ or ⍙, followed by 0 or more of above, _, or digits.

selection and assignment

N[*L*]	*select*
N ← *A*	*replace*
N[*L*] ← *A*	*insert*
N f← *A*	*modify#*
N[*L*] *f*← *A*	*modified insert#*

# *f is scalar dyadic primitive function*

scalar primitive functions

⌊ *D*	*floor*
⌈ *D*	*ceiling*
D ⌊ *D*	*minimum*
D ⌈ *D*	*maximum*
+ *D*	*identity*
- *D*	*negate*
× *D*	*signum*
÷ *D*	*reciprocate*
| *D*	*magnitude*
D + *D*	*add*
D - *D*	*subtract*
D × *D*	*multiply*
D ÷ *D*	*divide*
D | *D*	*residue*
⋆ *D*	*base e power*
● *D*	*base e logarithm*
D ⋆ *D*	*power*
D ● *D*	*logarithm*
D < *D*	*less*
D ≤ *D*	*not greater*
A = *A*	*equal*
D ≥ *D*	*not less*
D > *D*	*greater*
A ≠ *A*	*unequal*
~ *B*	*not*
B ∧ *B*	*and*
B ∨ *B*	*or*
B ⍲ *B*	*nand*
B ⍱ *B*	*nor*
○ *D*	*pi times*
I ○ *D*	*circular*
! *D*	*factorial*
D ! *D*	*combinatorial*

mixed primitive functions - structure

⍴ *A*	*shape*
I ⍴ *A*	*reshape*
⍳ *I*	*integers*
A ⍳ *A*	*index in*
, *A*	*ravel*
	catenate / laminate
A , *A*	*last dimension*
A ,[*K*] *A*	*Kth from first dim'n*
A ,[*D*] *A*	*between dim'ns* ⌊*D*, ⌈*D*
	reverse
⌽ *A*	*last dimension*
⊖ *A*	*first dimension*
⌽[*K*] *A*	*Kth from first dim'n*
⊖[*K*] *A*	*Kth from last dim'n*
	rotate
I ⌽ *A*	*last dimension*
I ⊖ *A*	*first dimension*
I ⌽[*K*] *A*	*Kth from first dim'n*
I ⊖[*K*] *A*	*Kth from last dim'n*
⍉ *A*	*transpose dimensions*
I ⍉ *A*	*permute dimensions*
	compress
B / *A*	*last dimension*
B ⌿ *A*	*first dimension*
B /[*K*] *A*	*Kth from first dim'n*
B ⌿[*K*] *A*	*Kth from last dim'n*
	expand
B \ *A*	*last dimension*
B ⍀ *A*	*first dimension*
B \[*K*] *A*	*Kth from first dim'n*
B ⍀[*K*] *A*	*Kth from last dim'n*
I ↑ *A*	*take*
I ↓ *A*	*drop*

mixed primitive functions - sets

A ∊ *A*	*membership*
A ⊂ *A*	*subset*
A ⊃ *A*	*superset*
A ∪ *A*	*union*
A ∩ *A*	*intersection*
A ~ *A*	*exclusion*

mixed primitive functions - other

⍋ D	grade up
⍒ D	grade down
? I	roll
I ? I	deal
D ⊥ D	base value
D ⊤ D	represent
⌹ D	matrix inverse
D ⌹ D	matrix divide
⍎ C	evaluate

format primitive functions

⍕ A	implicit format
K̲ ⍕ D	numeric format
K̲	in pairs w d
w	width
d	decimal places:
	<0 floating point
	=0 integer
	>0 fixed point
C̲ ⍕ L̲	character format
L̲	expression or (list)
C̲	format: s or s;...;s
s	segment: g or g,...,g
g	group: c or r̲(c)
r̲	replicator
c	clause: p or p,...,p
p	phrase: one of
m j A w	character
m j E w.d	floating point
m l q F w.d r	fixed point
m l q I w r	integer
X w	skip forward
T n	tab to n-th column
<text>	actual text
m	phrase replicator#
j	justifier:
L	left justify in field#
w	field width
d	decimal places
l r	left, right decorators:
-o+<text>	sign selector(s)#
*<text>	background#
q	qualifiers:
L	left justify in field#
B	skip if zero#
C	insert commas#
Z	insert leading zeros#

# optional field

primitive operators

A ∘.g A	outer product#
	reduction
f/ D	last dimension
f⌿ D	first dimension
f/[K] D	k-th from first dim'n
f⌿[K] D	k-th from last dim'n
	scan
f\ D	last dimension
f⍀ D	first dimension
f\[K] D	k-th from first dim'n
f⍀[K] D	k-th from last dim'n
A̲ f.g A	inner product#

f, g are scalar dyadic primitive functions

file functions

⍍ F	create file
N ⍍ F	rename file
⍔ F	destroy file
⍈[K] F	null Kth component
A ⍈[K] F	write Kth component
⍇[K] F	read Kth component
⍃ F	first-out component
⍄ F	last-out component
A ⍃ F	first-in component
A ⍄ F	last-in component
⌼ F	reverse components
I ⌼ F	rotate components
I ⍐ F	take components
I ⍗ F	drop components
B ⍁ F	compress components
B ⍂ F	expand components
⍓ F	hold file
⍌ F	free file
⍗ F	release file
⌸ F	value component map
⍯ F	null component map
⌻ I	interrogate system
⌻ F	test file status
I ⌻ F	query file

TRANSACTION CYCLE

1. SYSTEM INITIATES CYCLE BY DISPLAYING PROMPT AND UNLOCKING KEYBOARD.
2. USER SPECIFIES TRANSACTION BY MAKING TEXT ENTRY.
3. SYSTEM COMPLETES TRANSACTION BY INTERPRETING ENTRY, DISPLAYING APPROPRIATE DATA OR ERROR MESSAGE, AND RETURNING TO STEP 1.

TYPING RULES

KEY	ACTION
CHARACTER	INSERT CHARACTER INTO TEXT AT POSITION OF CURSOR, THEN MOVE TO RIGHT ONE SPACE.
SPACE	POSITION CURSOR ONE SPACE TO RIGHT.
BACKSPACE	POSITION CURSOR ONE SPACE TO LEFT.
TAB	POSITION CURSOR RIGHTWARD TO NEXT TAB STOP.
LINEFEED	DISCARD TEXT ABOVE AND TO RIGHT OF CURSOR.
RETURN	TERMINATE USER ENTRY PORTION OF TRANSACTION.

TRANSACTION EDIT

```
1. SYSTEM EITHER (1) TYPES OUT TEXT, RETURNS, AND UNLOCKS
   KEYBOARD, OR (2) EXDENTS CURSOR, AND UNLOCKS KEYBOARD.
2. USER TYPES IN EDIT CONTROLS.
   INITIAL INPUT OF --
     ATTENTION   SYSTEM ASSUMES MODIFICATION AT END OF
                 TEXT, POSITIONS CURSOR TO COLUMN
                 IMMEDIATELY TO RIGHT OF TEXT, UNLOCKS
                 KEYBOARD AND PROCEEDS AT STEP 4.
   OTHERWISE IF INPUT UNDER CHARACTER OF TEXT IS:
     '/'         DELETE CHARACTER ABOVE;
     '.'         MARK START OF NEXT PHRASE.
3. SYSTEM TYPES OUT REVISED TEXT, STOPPING BEFORE NEXT
   INSERTION POINT, AND UNLOCKS KEYBOARD.
4. USER ADDS TO, MODIFIES, OR TERMINATES CURRENT ENTRY BY
   USUAL TYPING RULES.
   INPUT OF --
     ATTENTION   PROCEEDS AT STEP 3 IF CURSOR TO RIGHT
                 OF CURRENT TEXT FOR NEXT PHRASE.
```

ATTENTION CONVENTIONS

<table>
<tr><th rowspan="2">KEYBOARD STATE</th><th rowspan="2">ATTENTION INPUT IS</th><th colspan="4">ACTION (SEE TRANSACTION EDIT FOR OVERRIDING USES)</th></tr>
<tr><th>MODE</th><th>PROMPT</th><th>AFTER VALID ENTRY</th><th>AFTER ERRONEOUS ENTRY</th></tr>
<tr><td rowspan="5">UNLOCKED</td><td rowspan="4">INITIAL</td><td>EXECUTION</td><td>FIVE SPACES</td><td>EDIT MOST RECENT APL EXPRESSION THIS LEVEL.</td><td rowspan="4">EDIT ERRONEOUS ENTRY.</td></tr>
<tr><td>DEFINITION</td><td>[...]</td><td>EDIT MOST RECENT DEFINITION MODE ENTRY.</td></tr>
<tr><td>⎕</td><td>⎕: LF 3-SP</td><td>PROCESS EVALUATED INPUT.</td></tr>
<tr><td>⍞</td><td>USER DEFINED</td><td>PROCESS CHARACTER INPUT.</td></tr>
<tr><td>NON-INITIAL</td><td colspan="4">SYSTEM LINEFEEDS, TYPES 'v', LINEFEEDS, AND UNLOCKS KEYBOARD.
ACTION SAME AS LINEFEED FOR TYPING RULES.</td></tr>
<tr><td>LOCKED</td><td>N.A.</td><td colspan="4">SEQUENTIALLY INPUT ATTENTIONS MEAN:

DURING EXECUTION OF AN APL EXPRESSION — OTHERWISE
FIRST: SUSPEND EXECUTION AFTER LINE AND ABORT QUEUED OUTPUT. — ABORT QUEUED OUTPUT.
SECOND: ALSO KILL ACTION.</td></tr>
</table>

INDEX

Terms indexed below with section and page numbers are used in sections 1 through 9 of this manual. Cross references are indicated by (see Primary listing). They are used both for alternative entries and for some terms used in other APL manuals and texts. Some generic terms are included to provide different categorizations than are discussed in detail in the manual: in particular, alphabetic lists of character names, file editing actions, file functions, primitive functions, system commands, system functions and system variables. For each APL character, entries are included for both the function or action names in which it is used, and the character name independent of its use.

- Absolute value | 5-8
- Account name
  - for files 7-1
  - for other workspaces 3-6
  - for sign-on)*ON* 3-2
  - function ⎕*AN* 6-12
- Accounting information ⎕*AI* 6-12
- Acoustic coupler 2-6
- Across dimension 4-2
- Action specifier 8-2
- Actions, on defined functions (see Function editing actions)
- Active
  - file 7-2
  - function 8-18
  - workspace 1-8
- Add + 5-8
- Along dimension 4-2
- Alpha α 8-10
- Alphabet ⎕*A* 6-9
- Alternating
  - product ÷/ 5-23
  - sum -/ 5-23
- And ∧ 5-13
- APL
  - character set 2-2
  - keyboards 2-3
  - MCS identifier \APL 2-6
- Apostrophe (see Quote) '
- Arccos 5-14
- Arccosh 5-14
- Arch characters
  - down ∪ 5-42, 8-14
  - left ⊂ 5-42
  - right ⊃ 5-42
  - up ∩ 5-42, 8-14
- Arcsin 5-14
- Arcsinh 5-14
- Arctan 5-14
- Arctanh 5-14
- Areas (see Availabilities)
- Arguments
  - definition 1-5
  - in function header 8-1
  - left and right 4-7, 8-1
- Arrays 4-2
- Arrow characters
  - down ↓ 5-40, 7-7
  - left ← 5-4
  - right → 8-3, 8-20
  - up ↑ 5-40, 7-7
- Assignment ← 5-4
- Asterisk * 5-10
- Atomic vector ⎕*AV* 6-9
- Attach system command)*ATTACH* 3-10
- Attention ATTN
  - key 2-4
  - editing last entered expression 2-9
  - entry typing error correction 2-8
  - function suspension 2-10
  - output termination 2-10
- Availability
  - file ⎕*FA* 6-12
  - library ⎕*LA* 6-12
  - name ⎕*NA* 6-12
  - workspace ⎕*WA* 6-12
- Average 8-2
- Backslash \\ 2-6, 2-9, 5-38, 8-10
  - dash ⍀ 5-38
  - quad ⍂ 7-8
- Backspace
  - key BKSP 2-4
  - character ⎕*B* 6-9
- Base e logarithm ⍟ 5-10
- Base e power * 5-10

Base value ⊥ 5-8
Beta function 5-16
Blanks as separator 4-9
Blot system command)*BLOT* 3-3
Body of function 8-2
Boolean type 4-4
Braces 2-3, 6-10
Brackets [ ]
  dimension selector 4-8
  index list 4-8, 5-2
  prompt 8-2
Branch →
  conditions 8-4
  described 8-3
  examples 8-4
  no argument (see Terminate) 8-4
Built-in functions (see Functions, primitive)
Bytes, space measure 6-12
Calculator mode, (see Execution mode)
Call
  defined function 8-18
  recursive 8-19
Canceling a line 2-8
Canonic representation ⎕*CR* 6-4
Cap ∩ 5-42, 8-14
Caret characters
  down ∨ 5-13
  left < 5-12, 5-13
  right > 5-12, 5-13
  up ∧ 5-13
  not down ⍱ 5-13
  not left ≥ 5-12, 5-13
  not right ≤ 5-12, 5-13
  not up ⍲ 5-13
Carrier return, (see Return)
Catenate , 5-8
Ceiling ⌈ 5-7, 8-14
Change password
  account 3-2
  file ⍍ 7-3
  workspace 3-7
Character
  alphabet ⎕*A* 6-9
  atomic vector ⎕*AV* 6-11
  data displayed 4-3
  data type 4-3
  digits ⎕*D* 6-9
  error 2-10
  format ⍕ 5-62
  input ⍞ 4-11
  quote ' 4-3
  set, APL 2-2, ⎕*AV* 6-11
  string 4-4
  type mixed with numeric 4-8
  vectors 4-3

Character names
  alpha ⍺ 8-10
  apostrophe (see quote) ' 4-3
  arch,
    down ∪ 5-42, 8-14
    left ⊂ 5-42
    right ⊃ 5-42
    up ∩ 5-42, 8-14
  arrow,
    down ↓ 5-40, 7-7
    left ← 5-4
    right → 8-3,8-20
    up ↑ 5-40, 7-7
  asterisk (see star) ⋆
  backslash \ 2-6, 5-38
    dash ⍀ 5-38
  braces 6-10
  brackets [ ] 4-8, 5-2, 8-2
  box (see quad) ⎕
  cap (see arch, up) ∩
  caret,
    down ∨ 5-13
    left < 5-12, 5-13
    right > 5-12, 5-13
    up ∧ 5-13
    not down ⍱ 5-13
    not left ≤ 5-12, 5-13
    not right ≥ 5-12, 5-13
    not up ⍲ 5-13
  ceiling ⌈ 5-7, 8-14
  circle ○ 5-14
    backslash ⍉ 5-36
    dash ⊖ 5-34
    star ⍟ 5-10
    stroke ⌽ 5-34
  colon : 4-11, 8-2
  comma , 5-8
  cup (see arch, down) ∪
  dash - 5-8
  del ∇ 8-6
    stroke ⍒ 5-44
    tilde ⍫ 8-6
  delta ∆ 4-6
    stroke ⍋ 5-44
    underbar ⍙ 2-4
  diamond 6-10
  dieresis ¨ 2-3
  digits 2-2
  divide ÷ 5-8
  dollar $ 6-10
  domino ⌹ 5-50
  epsilon ∊ 5-42, 8-10
  equal = 5-12
  exclamation ! 5-16
  floor ⌊ 5-7, 8-14
  I-bar ⌶ 6-22
  iota ⍳ 5-31, 8-10

jot ∘ 5-20
    down tack ⍎ 5-54
    up arch (see lamp) ⍝
    up tack ⍕ 5-55
letter 2-2
    underscored 2-2
lamp ⍝ 4-10
log ⍟ 5-10
negative ¯ 4-3
omega ⍵ 8-10
parentheses ( ) 4-7, 8-12
    right) 3-1
period . 2-9, 4-3, 5-20, 5-26, 8-10
plus + 5-8
quad ⎕ 4-11, 8-16
    backslash ⍂ 7-8
    circle ⌼ 7-6
    del ⍔ 7-3
    delta ⍍ 7-3
    divide (see domino) ⌹
    down arrow ⍗ 7-7, 7-9
    down caret ⍌ 7-9
    equal ⌸ 7-10
    jot ⌻ 7-11
    left arrow ⍇ 7-4
    left caret ⍃ 7-5
    quote ⍞ 4-11
    right arrow ⍈ 7-4
    slash ⍁ 7-8
    tilde ⎕̃ 6-10
    up arrow ⍐ 7-7
    up caret ⍓ 7-9
    unequal ⍯ 7-10
question mark ? 5-45, 8-16
quote ' 4-3
    quad (see quad quote) ⍞
rho ⍴ 5-30
semicolon ; 4-8, 4-11, 5-55, 8-1
slash / 2-9, 5-38, 8-10
    dash ⌿ 5-38
star * 5-38
stroke | 5-8
tack,
    down ⊥ 5-46
    left 2-3, 6-10
    right 2-3, 6-10
    up ⊤ 5-48
tilde ~ 5-13, 5-42, 8-17
times × 5-8
underscore _
    alphabet 4-6
    delta ⍙ 4-6
unequal ≠ 5-12

Circle ○ 5-14
    backslash ⍉ 5-36
    dash ⊖ 5-34
    star ⍟ 5-10
    stroke ⌽ 5-34
Circular functions ○ 5-14
Clear command)*CLEAR* 3-4
Close function action ∇ 8-6
Close and lock function action ⍫ 8-6
Coercion 5-17
Colon : 4-11, 8-2
Combinations ! 5-15
Combinatorial, generalized ! 5-16
Comma , 5-8
Command, system (see System command)
Comment ⍝ 4-10
Comparison tolerance
    default)*FUZZ* 3-5
    variable ⎕*CT* 6-2
Component
    of file 7-1
    of list 4-8
Compress / ⌿ 5-38
Compress components ⍁ 7-8
Conformable arguments 5-17
Connecting with the APL/700 system 2-6
Constant 1-5, 4-5
Continue off command)*COFF* 3-2
Control structures 1-7, 4-7, 8-18
Coordinates of an array (see Dimensions)
Copy command)*COPY* 3-7
Copy, protect)*PCOPY* 3-7
Corner element 4-2
Correction of typing error 2-8
Cosh 5-14
Cosine 5-14
Create file ⍍ 7-3
Cup ∪ 5-42
Curve fitting 5-53
Dash - 5-8
Data Communications Processor (DCP) 2-1
Data entry mode 1-4
Data object 4-2
    character 4-3
    display forms 4-3
    numeric 4-4
    tests of properties 4-5
Deal ? 5-45
Deblank 5-35
Decimal point . 4-4
Default format ⍕ 5-58
Define header action ∇ 8-6

Defined function editing actions
  action specifier 8-5
  define header ∇ 8-6
  delete ~ 8-17
  display line numbers *?* 8-16
  display lines ⎕ 8-16
  full edit line ∊ 8-10
  function
    close ∇ 8-6
    close and lock ⍫ 8-8
    open ∇ 8-6
    open locked ⍫ 8-6
  inject as most recent
   expression ⍳ 8-10
  insert
    before ↑ 8-8
    after ↓ 8-8
  multiline group specifier 8-12
  prefix edit ⍺ 8-10
  replace [ ] 8-8
  reset
    monitor ∪ 8-14
    stop ⊤ 8-14
    trace ⌊ 8-14
  set
    monitor ∩ 8-14
    trace ⌈ 8-14
    stop ⊤ 8-14
  suffix edit *⍵* 8-10
Defined functions
  arguments 8-1
  body 8-2
  canonic represent ⎕*CR* 8-4
  definition 8-1
  documentation 8-21
  editing actions (see Function
   editing actions)
  editing as data 8-21
  example 8-2
  execution, (see Execution of
   defined functions)
  fix ⎕*FX* 6-4
  header 8-1
  line numbers 8-5
  line renumbering 8-5
  local names 8-1
  local names list 8-1
  list command)*FNS* 3-9
  name 8-1
  name list ⎕*NL* 6-5
  transformations 6-4
  vector represent ⎕*VR* 6-4
Definition and editing mode,
 function 8-1
Del ∇ 8-6
  stroke ⍒ 5-44
  tilde ⍫ 8-6

Delay ⎕*DL* 6-8
Delete action ~ 8-17
Delta ∆ 4-6
  stroke ⍋ 5-44
  underbar ⍙ 2-4
Destroy file ⍔ 7-3
Detach command)*DETACH* 3-10
Diagnostics 6-6, 8-14
Diagonal, selecting from
 an array 5-36
Diamond 2-3, 6-10
Dieresis ¨ 2-3, 6-10
Digits)*DIGITS* 3-5
Dimension selector 5-19, 5-29
Dimensions 4-2
Display
  array 4-3
  defined function 8-16
  empty vector 4-3
  fractional number 4-4
  line numbers action *?* 8-16
  lines action ⎕ 8-16
  matrix 4-3
  number in E notation 4-4
  significant digits
   3-5, 4-4, 6-2
  value of expression 4-3
  vector 4-3
Divide ÷ 5-8
Documentation 8-21
Dollar sign *$* 6-10
Domain
  of function argument 1-5
  of numbers 4-4
Domino ⌹ 5-50
Down
  arch ∪ 5-42, 8-14
  arrow ↓ 5-40, 8-8
  caret ∨ 5-13
  tack ⊥ 5-46, 8-14
E notation for numbers *E* 4-4
Edit
  characters . / 2-9, 8-10
  function 8-10, 8-21
  most recently entered
   expression 2-9
  system function ⎕*ED* 6-8
Elaboration of expression 4-7
Element of a data object 4-2
Empty array 4-4
Entry length 2-5
Epsilon ∊ 5-42, 8-10
Equal = 5-12
Erase
  command)*ERASE* 3-9
  file (see Destroy)
  system function ⎕*EX* 6-5
  typing error 2-8
  workspace (see Drop command)

Error reports, table of 9-4
Errors
    described 9-1
    in a defined function 9-2
    in expression entry 4-9
    reports, table 9-4
    system function ⎕*ER* 6-8
Escape
    from character input Ø 4-12
    from evaluated input → 4-12
Evaluation of expressions 4-7
Evaluate ⍎ 5-54
Evaluated input ⎕ 4-11
Exclamation ! 5-16
Execution
    controls 6-8
    mode 1-4
    state 2-1
Execution of defined functio
    active function 8-18
    call 8-18
    diagnostics 6-6, 8-14
    dynamic expunging 8-18
    global name 8-18
    instance 8-18
    local name 8-18
    monitor 6-6,8-14
    multiple instances 8-19
    pendant 8-18
    recursive calls 8-19
    result 8-18
    scope of local name 8-18
    structured program 8-18
    stop 6-6, 8-14
    suspended function 8-18, 8-20
    trace 6-6, 8-14
Expand \ ⍀ 5-38
Expand components ⍂ 7-8
Explicit output ⎕← 4-11
Exponential notation *E* 4-4
Expression
    definition 1-7
    entry 4-9
    list 4-8
    order of execution 4-7
    with a quad or
      quote-quad ⎕ ⍞ 4-11
Expunge ⎕*EX* 6-5
Factorial ! 5-16
File
    account name 7-1
    active status 7-2
    availability ⎕*FA* 6-12
    components 7-1
    inactive status 7-2
    integrity 7-2
    library names)*FILES* 3-6
    limits 7-1
    name 7-1
    password 7-1
    open 7-2
File functions
    change password ⍍ 7-3
    create file ⍍ 7-3
    compress components ⍁ 7-8
    destroy file ⍔ 7-3
    expand components ⍂ 7-7
    first component in ⍃ 7-5
    first component out ⍃ 7-5
    free ⍔ 7-9
    hold ⍍ 7-9
    interrogate file system ⌻ 7-11
    last component in ⍄ 7-5
    last component out ⍄ 7-5
    map components non-null ⌸ 7-10
    map components null ⍯ 7-10
    null component ⍈ 7-4
    query file attribute ⌻ 7-11
    read component ⍇ 7-4
    release file ⍗ 7-9
    rename file ⍍ 7-3
    reverse components ⌼ 7-6
    rotate components ⌼ 7-6
    take components ⍐ 7-7
    test file status ⌻ 7-11
    write component ⍈ 7-4
Fill 5-38, 5-40
First file component
    in ⍃ 7-5
    out ⍃ 7-5
Fix ⎕*FX* 6-4
Fixed Point number 4-4
Floor ⌊ 5-7, 8-14
Forgotten password 1-9
Formally equivalent
  expressions ←→ 5-1
Format ⍕ 5-55
Fractional numbers 4-4
Free file ⍔ 7-9
Full edit action ϵ 8-10
Function definition and editing
  mode 8-1
Functions
    defined (see Defined
      functions) 8-1
    file (see File functions) 7-1
    primitive (see Primitive
      functions) 5-1
    system (see System
      functions) 6-1
Future value 5-47

Fuzz
  command)*FUZZ* 3-5
  system function ⎕*CT* 6-2
Gamma function ! 5-16
Generalized combination ! 5-16
Generalized factorial ! 5-16
Global name 8-18
Go to (see Branch) 8-3
Grade down ⍒ 5-44
Grade up ⍋ 5-44
Graph construction 5-21
Greater than > 5-12
Greater than or equal ≥ 5-12
Group commands 3-10
  names command)*GRPS* 3-9
  content command)*GRP* 3-11
  name of 4-6
Header, function 8-1
Hexadecimal/decimal
  conversion 5-47, 5-49
Histogram construction 5-21
Hold File ⍓ 7-9
Hyperbolic functions ○ 5-14
I-bar functions ⌶ 6-22
Identifier 4-6
Identity + 5-8
Identity elements for scalar
  dyadic functions 5-28
Illegal character
  display ▯ 2-10, 6-11
Implicit output 4-11
Inactive
  file 7-2
  workspace 1-8
Index 5-2
  generator ⍳ 5-3
  list 5-2
  number 5-19, 5-29
  of (ranking) ⍳ 5-31
  sequence (row major
    order) 5-32
Index origin
  default origin)*ORIGIN* 3-4
  variable ⎕*IO* 6-2
  effect on functions 6-3
Indexing [ ] 5-2
Inject line action ⍳ 8-10
Inner product operator ⊕.⊗ 5-26
Input
  transaction 2-8
  communicators ⎕ ⍞ 4-11
Insert
  extra blanks 4-9
  line action ↑ ↓ 8-8
  value in array *A*[*L*]← 5-4
Instance of defined function 8-18
Integer
  fraction separation ⊤ 5-49
  not greater than ⌊ 5-7
  not less than ⌈ 5-7
  subtype of numeric 4-4
Integers to ⍳ 5-31
Internal character
  representation ⎕*AV* 6-9
Interpolation 5-52
Interrogate file system ⌻ 7-11
Intersection ∩ 5-42
Interrupt (see Attention)
Inverse
  hyperbolic functions ○ 5-14
  matrix ⌹ 5-50
  trigonometric function ○ 5-14
Iota ⍳ 5-31, 8-10
Jot ∘ 5-20
  down tack ⍎ 5-54
  quad ⌻ 7-11
  up arch ⍝ 4-10
  up tack ⍕ 5-55
Justify ragged array 5-25
Keyboard 2-3
Label
  name 4-6
  use of 8-2, 8-3
Laminate , 5-32
Lamp ⍝ 4-10
Last entered expression 4-9
Last file component
  in ⍄ 7-5
  out ⍄ 7-5
Least squares estimation 5-53
Leaving function definition
  mode 8-8
Left
  arch ⊂ 5-42
  arrow ← 5-4
  brace 6-10
  bracket [4-8, 5-2, 8-5
  caret < 5-12
  parenthesis (4-2, 8-12
  tack 2-3, 6-10
Length of names 4-6
Less than < 5-12
Less than or equal ≤ 5-12
Library
  area ⎕*LA* 6-12
  command)*LIB* 3-6
  of files)*FILES* 3-6
  of inactive workspaces 1-8
Line
  count ⎕*LC* 6-12
  drops 1-8
  editing α ∊ ω 8-10
  group specifier 8-12

in function definition 8-2
insertion ↑ ↓ 8-8
number 8-5
number specifier 8-5
renumbering 8-5
replacing [ ] 8-8
Linear
curve fit 5-53
equations 5-52
least squares estimation 5-53
Linefeed
key 2-5
character ⎕*L* 6-9
List
component 4-8
delimiter ; 4-8
expression 4-8
local names 8-1
Literal character constant 4-5
Load command)*LOAD* 3-7
Local
functions 8-2
label constant 8-3
left argument 8-1
name 8-2, 8-18
names list 8-1
right argument 8-1
result 8-1
variables 8-2
Local/communicate switch 2-4
Location in ⍳ 5-31
Locked
account 3-2
file ⍍ 7-3
function ⍫ 8-6
keyboard 1-3, 2-5
workspace 3-6
Locks and passwords 1-9
Log ⍟ 5-10
Logarithm ⍟ 5-10
Logical functions 5-13
Logical station number
LSN 2-6
Looping
by backward branch → 8-3
interrupt by ATTN 2-10
Main diagonal 5-37
Map file components
null ⍁ 7-10
non-null ⌸ 7-10
Margin key 2-5
Match ∧.= 5-27
Matrix
described 1-4, 9-2
display of 4-3
Matrix divide ⌹ 5-50
Matrix inverse ⌹ 5-50
Matrix multiply
inner product +.× 5-26
outer product ∘.× 5-20
scalar × 5-8
Maximum ⌈ 5-7
likelihood estimator 5-53
Membership ∊ 5-42
Merge 5-44
Message control system MCS 2-6
Minimax ⌊.⌈ 5-27
Minimum ⌊ 5-7
Minterm ∨.∧ 5-27
Mixed
primitive functions 5-29
radix 5-46, 5-48
type list 4-11, 5-55
Modes 1-4
character input 4-11
evaluated input ⎕: 4-11
execution function
definition and
editing 8-1
Modify ⊕← 5-4
Modified insert []⊕← 5-4
Monadic function defined 4-7
Monitor execution 6-6, 8-14
Monitor values ⎕*MV* 6-6
Most recently entered expression 4-9
Multiline group specifier 8-12
Multiple
spaces 4-9
linear regression 5-55
Multiply × 5-8
N-dimensional data object 4-2
Name
account 4-6
area ⎕*NA* 6-12
classification ⎕*NA* 6-5
defined function 4-6, 8-1
display 3-9
file 4-6
group 4-6
label 4-6
list ⎕*NL* 6-5
password 4-6
restrictions 4-6
shared variable 4-6
variable 4-6
uses 4-6
workspace 4-6
Name displays
files)*FILES* 3-6
functions)*FNS* 3-9
group)*GRP* 3-11
groups)*GRPS* 3-9
library of workspaces)*LIB* 3-6
list ⎕*NL* 6-5
variables)*VARS* 3-6

- Nand ⍲ 5-13
- Natural logarithm ⍟ 5-10
- Negate - 5-8
- Negative sign for number ¯ 4-3
- News
  - sign-on message 2-7
  - system function ⎕*NEWS* 6-12
- Niladic function 4-7
- No-element array 4-3
- Nonscalar arguments used with scalar functions 5-17
- Nor ⍱ 5-13
- Not ~ 5-13
- Not equal ≠ 5-12
- Not greater ≤ 5-12
- Not less ≥ 5-12
- Null
  - character ⎕*N* 6-9
  - file component ⍈ 7-4
  - file map ⌧ 7-10
- Number of users ⎕*UL* 6-12
- Numeric
  - data type 4-3
  - character representation 6-9
  - format ⍕ 5-6
  - vector 4-3
- Off commands
  - terminate)*OFF* 3-2
  - suspend, continue)*COFF* 3-2
- Omega *ω* 8-10
- On command)*ON* 3-2
- One-element array 4-4
- Open
  - file 7-2
  - function action ∇ 8-6
  - locked function action ⍫ 8-6
- Operators, primitive
  - inner product ⊕.⊗ 5-26
  - outer product ∘.⊗ 5-20
  - reduction /⊗ 5-22
  - scan \⊗ 5-24
- Or ∨ 5-13
- Order of elaboration 4-7
- Origin (see Index origin)
- Orthogonal 4-2
- Outer product operator ∘.⊕ 5-20
- Output for display
  - communicators ⎕← ⍞← 4-11
  - display form 4-3
  - mixed type 4-11
- Over (see Reduction)
- Overstruck characters 2-4
- Overtake ↑ 5-40
- Parallel elaboration 1-7, 4-7
- Parentheses ( )
  - in an expression 4-7
  - redundant 4-8
  - system command prefix) 3-1
- Password
  - for sign-on 2-6
  - forgotten 1-9
  - name 4-6
  - on account 3-2
  - on file 7-3
  - on workspace 3-7
  - security use 1-9
- Pendant function 8-18
- Period . 2-9, 4-3, 5-20, 5-26, 8-10
- Permute dimensions ⍉ 5-36
- Pi times ○ 5-14
- Plane across dimensions 4-2
- Plus + 5-8
- Polynomial 5-47
- Power * 5-10
- Precision of numbers 4-4
- Prefix edit action α 8-10
- Present value 5-47
- Primitive Functions
  - absolute value | 5-8
  - add + 5-8
  - and ∧ 5-13
  - base e logarithm ⍟ 5-10
  - base e power * 5-10
  - base value ⊥ 5-46
  - catenate , 5-32
  - ceiling ⌈ 5-7
  - circular ○ 5-14
  - combinatorial ! 5-16
  - compress / ⌿ 5-38
  - deal ? 5-45
  - divide ÷ 5-8
  - drop ↓ 5-40
  - equal = 5-12
  - evaluate ⍎ 5-54
  - expand \ ⍀ 5-38
  - factorial ! 5-16
  - floor ⌊ 5-7
  - format ⍕ 5-55
  - grade down ⍒ 5-44
  - grade up ⍋ 5-44
  - greater than > 5-12
  - greater than or equal ≥ 5-12
  - identity + 5-8
  - index of ⍳ 5-31
  - integers to ⍳ 5-31
  - intersection ∩ 5-42
  - laminate , 5-32
  - less than < 5-12

    less than or equal ≤ 5-12
    logarithm ⍟ 5-10
    magnitude | 5-8
    matrix divide ⌹ 5-50
    matrix inverse ⌹ 5-50
    maximum ⌈ 5-7
    membership ∊ 5-42
    minimum ⌊ 5-7
    multiply × 5-8
    nand ⍲ 5-13
    natural logarithm ⍟ 5-10
    negate - 5-8
    nor ⍱ 5-13
    not equal ≠ 5-12
    not greater ≤ 5-12
    not less ≥ 5-12
    or ∨ 5-13
    permute dimensions ⍉ 5-36
    pi times ○ 5-14
    power * 5-10
    ravel , 5-32
    reciprocate ÷ 5-8
    represent ⊤ 5-48
    reshape ⍴ 5-30
    residue | 5-8
    reverse ⌽ ⊖ 5-34
    roll 5-45
    rotate ⌽ ⊖ 5-34
    selection [ ] 5-2
    set exclusion ~ 5-42
    shape ⍴ 5-30
    signum × 5-8
    subset ⊂ 5-42
    superset ⊃ 5-42
    subtract - 5-8
    take ↑ 5-40
    transpose dimensions ⍉ 5-36
    unequal ≠ 5-12
    union ∪ 5-42
Primitive operators (see
  Operators, primitive)
Principal diagonal of matrix 5-37
Print precision
    default)*DIGITS* 3-5
    variable ⎕*PP* 6-2
Print tabs
    default)*TABS* 3-3
    function ⎕*PT* 6-12
Print width 3-3
    default)*WIDTH* 3-3
    variable ⎕*PW* 6-12
Procedures for terminal 2-6
Processor phase 1-3
Progressive expression
  development 4-9
Prompts 1-4
Properties of data objects 4-5
Protect copy)*PCOPY* 3-7
Protecting functions ⍫ 8-6
Quad
    display lines ⎕ 8-16
    evaluated input prompt ⎕: 4-11
    explicit output ⎕← 4-11
    overstrikes (see character names)
    quitting quad input 4-12
Qualification, line group 8-12
Query
    file attribute ⌼ 7-11
    monitors ⎕*QM* 6-6
    stop ⎕*QS* 6-6
    trace ⎕*QT* 6-6
Question mark *?* 5-45, 8-16
Quitting (see Sign-off)
Quotas
    computer use 1-7
    files, number 1-7, +/⎕*FA* 6-12
    files, space, 1-7, ⌼ 7-11
    on user account 1-7
    shared variables 1-7
    workspaces 1-7, +/⎕*LA* 6-12
Quote-quad ⍞ 4-11
Quotes in character data ' 4-3
Radians ○ 5-14
Radices 5-8
Random number
    deal *?* 5-45
    link default)*SEED* 3-5
    link variable ⎕*RL* 6-2
    roll *?* 5-45
Range of a function result 1-5
Rank
    of a data object 4-2
    determined by shape
      function 5-30
Rank-n arrays
    described 4-2
    display of 4-3
Ravel , 5-32
Read component ⍇ 7-4
Reciprocate ÷ 5-8
Recovery 1-8, 2-12
Redundant
    blanks 4-9
    parentheses 4-9
Recursive function 8-19
Reduction operator ⊕/ 5-22
Relational functions 5-12
Release file ⊞ 7-90

- Rename
  - file ⍍ 7-3
  - function 8-10
  - workspace)*WSID* 3-8
- Reordering a vector 7-3
- Repeat key 2-5
- Replace
  - function ← 5-4
  - line action 8-8
  - multiple 5-5
- Represent ⊤ 5-48
- Request for input
  - character ⍞ 4-11
  - evaluated ⎕ 4-11
    - prompt ⎕. 4-11
- Reset
  - command)*RESET* 3-12
  - monitors ⎕*RM* 6-6, ∪ 8-14
  - stop ⎕*RS* 6-6, ⌊ 8-14
  - trace ⎕*RT* 6-6, ⊥ 8-14
- Reshape ⍴ 5-30
- Residuals of curve fit 5-53
- Residue | 5-8
- Result
  - explicit 8-1, 8-18
  - value of expression 4-7
- Return
  - key RETN 2-4
  - character ⎕*R* 6-9
  - completing entry 2-8
- Reverse ⌽ ⊖ 5-34
- Reverse components ⌼ 7-6
- Rho ⍴ 5-30
- Right
  - arch ⊃ 5-42
  - arrow → 8-3
  - brace 2-3, 6-10
  - bracket] 4-8, 5-2, 8-5
  - caret > 5-12
  - parenthesis) 4-7, 8-12
  - tack 2-3, 6-10
- Right-to-left elaboration 4-7
- Roll ? 5-45
- Root (see Power)
- Rotate ⌽ ⊖ 5-34
- Rotate components ⌼ 7-6
- Row major order 5-32
- Save workspace command)*SAVE* 3-7
- Scalar
  - single character 4-3
  - number 4-3
- Scalar primitive functions
  - definition 5-6
  - extension arrays 5-17
- Scan operator ⊕\ 5-24
- Scientific notation 4-4
- Scope of local name 8-18
- Security 1-9
- Seed command)*SEED* 3-5
- Selection [ ] 5-2
- Self protections 1-8
- Semicolon ;
  - and indexing 4-8
  - local names list delimiter 8-1
  - with formatted output 5-55
  - with mixed output 4-11
- Sequence of characters 5-3, 5-32
- Set
  - character input prompt ⍞← 4-11
  - exclusion ~ 5-42
  - functions 5-42
  - monitors ⎕*SM* 6-6, ∩ 8-14
  - stop ⎕*SS* 6-6, ⌈ 8-14
  - trace ⎕*ST* 6-6, ⊤ 8-14
- Session controls 3-2
- Shape ⍴ 5-30
- Shape of data object 4-2, 4-4
- Shared variables
  - access control 6-13
  - availability ⎕*SA* 6-12
  - control ⎕*SVC* 6-18
  - coupling ⎕*SVO* 6-16
  - name 4-6
  - offer ⎕*SVO* 4-6
  - query ⎕*SVQ* 6-20
  - retract ⎕*SVR* 6-20
  - surrogate 6-13
- Shares availability ⎕*SA* 6-12
- Sharing 1-9
- Shift key 2-4
- Shriek ! 5-16
- Sign-off 2-6
  - continue)*COFF* 3-2
  - discard)*OFF* 3-2
- Sign-on 2-6
  - connect command)*ON* 3-2
  - MCS specifier \APL 2-6
- Significant digits (see Print Precision)
- Signum × 5-8
- Sine 5-14
- Single 4-4
- Sinh 5-14
- Size ×/⍴ 4-2
- Slash / 2-9, 5-38, 8-10
  - dash ⌿ 5-38
- Solidus / (see slash)
- Sorting 5-44
- Space
  - bar 2-4
  - with constant vector 4-9
  - in formatted data 5-55
  - redundant 4-9

- Special characters 2-2, 2-4
- Square root
  - normalized surds ○ 5-14
  - general (see power) ⋆ 5-11
- Squish quad ⌷ 2-10, 6-11
- Standard functions (see functions, primitive)
- Star ⋆ 5-10
- State indicator)*SI* 3-12
- Station name 2-6
- Stopping
  - function execution ATTN 2-10
  - output display ATTN 2-10
  - terminate → 8-3
  - session (see sign-off)
  - suspend 6-6, 8-11
- Storage space availability ⎕*WA* 6-12
- String 4-4
- Stroke | 5-8
- Structure of an expression 4-7
- Structured program 8-18
- Structuring an array 4-3, 5-30
- Subarray 5-2
- Subscript list 5-2
- Subset ⊂ 5-42
- Subtract - 5-8
- Sudilos \ (see backslash)
- Suffix edit action ω 8-10
- Superset ⊃ 5-42
- Surrogate 6-13
- Suspended functions
  - clearing state indicator)*RESET* 3-12, 8-20
  - correcting errors in 8-20
  - detection of 8-20
  - meaning of 8-18
  - termination → 8-20
- Switches on the terminal 2-4
- Symbol table
  - availability ⎕*NA* 6-12
  - command)*SYMS* 3-4
  - entry by replace 5-5
- Syntax
  - defined functions 8-1
  - expression list 4-8
  - expressions 4-7
  - formats 5-56
  - primitive functions 4-7
- System commands
  -)*ATTACH* 3-10
  -)*BLOT* 3-3
  -)*CLEAR* 3-4
  -)*COFF* 3-2
  -)*COPY* 3-7
  -)*DETACH* 3-10
  -)*DIGITS* 3-5
  -)*DROP* 3-8
  -)*ERASE* 3-9, 8-21
  -)*FILES* 3-6
  -)*FNS* 3-9
  -)*FUZZ* 3-5
  -)*GRP* 3-11
  -)*GRPS* 3-9
  -)*LIB* 3-6
  -)*LOAD* 3-7
  -)*OFF* 3-2
  -)*ON* 3-2
  -)*ORIGIN* 3-4
  -)*PCOPY* 3-7
  -)*RESET* 3-12, 8-20
  -)*SAVE* 3-7
  -)*SEED* 3-5
  -)*SI* 3-12
  -)*SYMS* 3-4
  -)*TABS* 3-3
  -)*VARS* 3-9
  -)*WIDTH* 3-3
  -)*WSID* 3-8
- System functions
  - accounting information ⎕*AI* 6-12
  - account name ⎕*AN* 6-12
  - alphabet ⎕*A* 6-9
  - atomic vector ⎕*AV* 6-9
  - backspace character ⎕*B* 6-9
  - canonic representation ⎕*CR* 6-4
  - delay ⎕*DL* 6-8
  - digits ⎕*D* 6-9
  - edit ⎕*ED* 6-8
  - error ⎕*ER* 6-8
  - expunge ⎕*EX* 6-5
  - file availability ⎕*FA* 6-12
  - fix ⎕*FX* 6-4
  - library availability ⎕*LA* 6-12
  - line count ⎕*LC* 6-12
  - linefeed character ⎕*L* 6-9
  - monitor values ⎕*MV* 6-6
  - name availability ⎕*NA* 6-12
  - name classification ⎕*NC* 6-5
  - name list ⎕*NL* 6-5
  - news ⎕*NEWS* 6-12
  - null character ⎕*N* 6-9
  - print tabs ⎕*PT* 6-12
  - print width ⎕*PW* 6-12
  - query monitors ⎕*QM* 6-6
  - query stop ⎕*QS* 6-6
  - query trace ⎕*QT* 6-6
  - reset monitors ⎕*RM* 6-6
  - reset stop ⎕*RS* 6-6
  - reset trace ⎕*RT* 6-6
  - return character ⎕*R* 6-9

set monitors ⎕*SM* 6-6
set stop ⎕*SS* 6-6
set trace ⎕*ST* 6-6
shared variable
control ⎕*SVC* 6-18
offer ⎕*SVO* 6-16
query ⎕*SVQ* 6-20
retract ⎕*SVR* 6-20
shares availability ⎕*SA* 6-12
tab character ⎕*T* 6-9
time stamp ⎕*TS* 6-12
user load ⎕*UL* 6-12
vector representation ⎕*VR* 6-4
working availability ⎕*WA* 6-12
workspace identity ⎕*WI* 6-12
System information ɪ 6-20
System variables 6-2
comparison tolerance ⎕*CT* 6-2
index origin ⎕*IO* 6-2
print precision ⎕*PP* 6-2
random link ⎕*RL* 6-2
Tab
key, SET/CLR 2-5
character ⎕*T* 6-9
command)*TABS* 3-3
inquiry ⎕*PT* 6-12
interval 2-7
Tables 5-21
Tack characters
down ⊥ 5-46
left 2-3, 6-10
right 2-3, 6-10
up ⊤ 5-48
Take ↑ 5-40
Take components ⍐ 7-7
Tangent 5-14
Tanh 5-14
Telephone 2-6
Template of function 8-1
Terminal
keyboard 2-3
logical station number 2-6
station name 2-6
tab interval 2-7
width of display 2-7
Terminating function execution
after a suspension)*RESET* 3-12
from function line → 8-3
Test file status ⌼ 7-11
Text (see string) 4-4, 8-11
Tilde ~ 5-13, 5-42, 8-17
Times × 5-8
Time stamp ⎕*TS* 6-12
Trace
function execution 6-6, 8-14
of matrix 5-37
Transaction
definition 1-3
entries 2-8
editing 2-8
Transformations, function 6-4
Transpose dimensions ⍉ 5-36
Triangular numbers 5-25
Trigonometic functions 5-14
Truth table 5-12
Type of data object 1-4
Typing errors 2-8
Twitch prompt 2-10
Underscore _ 4-6
delta ⍙ 4-6
Unequal ≠ 5-12
Union ∪ 5-42
Up
arch ∩ 5-42, 8-14
arrow ↑ 5-40, 8-8
caret ∧ 5-13
tack ⊤ 5-48
User
account 1-7
name ⎕*AN* 6-12
defined functions (see
defined functions) 8-1
load ⎕*UL* 6-12
phase 1-3
Value of a data object 4-2
Variable
global 8-18
list ⎕*NL* 6-5
command)*VARS* 3-9
local 8-1, 8-18
name 4-6
Vector
described 4-2
display of 4-3
empty 4-4
represent ⎕*VR* 6-4
Visual fidelity 2-5, 4-9
Width
default command)*WIDTH* 3-3
print ⎕*PW* 6-12
Working space
availability ⎕*WA* 6-12
Workspace
attributes 3-4
clear)*CLEAR* 3-4
identity ⎕*WI* 6-12
library 3-6
locking of 3-2
name 4-6
name command)*WSID* 3-16
Write component ⍈ 7-4